Ο ΜΕΤΑΓΛΩΤΤΙΣΤΗΣ ΤΗΣ CIMPLE

Πάσχος Ελευθέριος Α.Μ 4151
Τμήμα Μηχανικών Η/Υ Πανεπιστήμιο Ιωαννίνων 2021
Μάθημα: Μεταφραστές 8ο Εξάμηνο

ΠΕΡΙΕΧΌΜΕΝΑ

1)Περιγραφή της γλώσσας Cimple (Σελ. 2)
2)Λεκτική ανάλυση (Σελ. 8)
3)Συντακτική ανάλυση (Σελ. 11)
4)Σημασιολογική ανάλυση (Σελ. 13)
5)Παραγωγή ενδιάμεσου κώδικα (Σελ. 18)
6)Παραγωγή τελικού κώδικα (Σελ. 22)
7)Testing (Σελ. 30)
8)Bugs/Errors (Σελ. 38)

Οι εικόνες που χρησιμοποιούνται στο παρακάτω κείμενο καθώς και επιλεγμένοι σύνδεσμοι προέρχονται από το μάθημα του Κ.Γεώργιου Μανή στο μάθημα των Μεταφραστών Ακ.Έτος 2020-2021 Πανεπιστήμιο Ιωαννίνων τμήμα Μηχανικών Η/Υ και πληροφορικής.

1)ΠΕΡΙΓΡΑΦΗ ΤΗΣ ΓΛΩΣΣΑΣ CIMPLE

Η γλώσσα προγραμματισμού Cimple αποτελεί μια γλώσσα προγραμματισμού με τις πολύ απλές λειτουργίες μιας προγραμματιστικής γλώσσας καθώς και διάφορες καινοτόμες λειτουργίες που την ξεχωρίζουν από τις άλλες.

Αρχικά, η cimple περιλαμβάνει ένα σύνολο δεσμευμένων λέξεων, αυτές είναι οι **program declare if else while switchcase forcase incase case default not and or function procedure call return in inout input print**.

Το αλφάβητο της Cimple αποτελείται από:

• τα μικρά και κεφαλαία γράμματα της λατινικής αλφαβήτου (A,…,Z και a,…,z),
• τα αριθμητικά ψηφία (0,…,9),
• τα σύμβολα των αριθμητικών πράξεων (+, -, *, /),
• τους τελεστές συσχέτισης (<, >, =, <=, >=, <>)
• το σύμβολο ανάθεσης (:=)
• τους διαχωριστές (;, “,”, :)
• τα σύμβολα ομαδοποίησης ([,], (,) , { , })
• του τερματισμού του προγράμματος (.)
• και διαχωρισμού σχολίων (#)

Οι σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων. Οι ακέραιες σταθερές πρέπει να έχουν τιμές από –(2^32 − 1) έως 2^32 − 1.

Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Κάθε αναγνωριστικό αποτελείται από τριάντα το πολύ γράμματα. Αναγνωριστικά με περισσότερους από 30 χαρακτήρες θεωρούνται λανθασμένα.

Οι λευκοί χαρακτήρες (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια, να μην βρίσκονται μέσα σε δεσμευμένες λέξεις, αναγνωριστικά, σταθερές. Το ίδιο ισχύει και για τα σχόλια, τα οποία πρέπει να βρίσκονται ανάμεσα στα σύμβολα #.

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η Cimple είναι οι ακέραιοι αριθμοί. Η δήλωση γίνεται με την εντολή declare. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές και χωρίς να είναι αναγκαίο να βρίσκονται στην ίδια γραμμή. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης αναγνωρίζεται με το ελληνικό ερωτηματικό. Επιτρέπεται να έχουμε περισσότερες των μία συνεχόμενες χρήσεις της declare.

Η προτεραιότητα των τελεστών από τη μεγαλύτερη στη μικρότερη είναι:

• Πολλαπλασιαστικοί: *, /
• Προσθετικοί: +, -
• Σχεσιακοί: =, <, >, <>, <=, >=
• Λογικοί: not
• Λογική σύζευξη: and

• Λογική διάζευξη: or

Μορφή Προγράμματος

Κάθε πρόγραμμα ξεκινά με τη λέξη κλειδί program. Στη συνέχεια ακολουθεί ένα αναγνωριστικό (όνομα) για το πρόγραμμα αυτό και τα τρία βασικά μπλοκ του προγράμματος: οι δηλώσεις μεταβλητών (declarations), οι συναρτήσεις και διαδικασίες (subprograms), οι οποίες μπορούν και να είναι φωλιασμένες μεταξύ τους, και οι εντολές του κυρίως προγράμματος (statements). Η δομή ενός προγράμματος Cimple φαίνεται παρακάτω. Προσέξτε την τελεία στο τέλος.

```
program id
        declarations
        subprograms
        statements
.
```

Δομές της Cimple

Εκχώρηση

ID := expression

Απόφαση if

```
if (condition)
        statements1
[ else
        statements2 ]
```

Επανάληψη while

```
while (condition)
        statements
```

Επιλογή switchcase

```
switchcase
      (case (condition) statements1 ) *
      default statements2
```

Η δομή switchcase ελέγχει τις condition που βρίσκονται μετά τα case. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες statements1 (που ακολουθούν το condition). Μετά ο έλεγχος μεταβαίνει έξω από την switchcase. Αν, κατά το πέρασμα, καμία από τις case δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην default και εκτελούνται οι statements2. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την switchcase.

Επανάληψη forcase

```
forcase
   (case (condition) statements1 ) *
   default statements2
```

Η δομή επανάληψης forcase ελέγχει τις condition που βρίσκονται μετά τα case. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες statements1 (που ακολουθούν το condition). Μετά ο έλεγχος μεταβαίνει στην αρχή της forcase. Αν καμία από τις case δεν ισχύει, τότε ο έλεγχος μεταβαίνει στη default και εκτελούνται οι statements2. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την forcase.

Επανάληψη incase
incase
(case (condition) statements1) *
Η δομή επανάληψης incase ελέγχει τις condition που βρίσκονται μετά τα case, εξετάζοντας τες κατά σειρά. Για κάθε μία για τις οποίες η αντίστοιχη condition ισχύει εκτελούνται οι statements που ακολουθούν το condition. Θα εξεταστούν με τη σειρά όλες οι condition και θα εκτελεστούν όλες οι statements των οποίων οι condition ισχύουν. Αφότου εξεταστούν όλες οι case, ο έλεγχος μεταβαίνει έξω από τη δομή incase, εάν καμία από τις statements δεν έχει εκτελεστεί, ή μεταβαίνει στην αρχή της incase, αν έστω και μία από τις statements έχει εκτελεστεί.

Η επιλογή switchcase, η επανάληψη forcase καθώς και η επανάληψη incase είναι τρεις από τις δομές που χαρακτηρίζουν την Cimple σαν μια καινοτόμο γλώσσα.

Επιστροφή τιμής συνάρτησης
return (expression)

Έξοδος δεδομένων
print (expression)

Είσοδος δεδομένων
input (ID)

Κλήση διαδικασίας
call functionName(actualParameters)
όπου actualParameters οι παράμετροι με τις οποίες καλείται μια διαδικασία

Συναρτήσεις και διαδικασίες
Η Cimple υποστηρίζει συναρτήσεις και διαδικασίες. Για τις συναρτήσεις η σύνταξη είναι:
function ID (formalPars)
{
declarations
subprograms
statements
}
ενώ για τις διαδικασίες:
procedure ID (formalPars)
{
declarations
subprograms
statements
}

Η formalPars είναι η λίστα των τυπικών παραμέτρων. Οι συναρτήσεις και οι διαδικασίες μπορούν να φωλιάσουν η μία μέσα στην άλλη. Οι κανόνες εμβέλειας ακολουθούν τους κανόνες της PASCAL. Η επιστροφή της τιμής μιας συνάρτησης γίνεται με την return. Η κλήση μιας συνάρτησης, γίνεται μέσα από τις αριθμητικές παραστάσεις σαν τελούμενο. π.χ. D = a + f(in x) όπου f η συνάρτηση και x παράμετρος που περνάει με τιμή. Η κλήση μιας διαδικασίας γίνεται με την call. π.χ. call f(inout x) όπου f η διαδικασία και x παράμετρος που περνάει με αναφορά.

Μετάδοση παραμέτρων

Η Cimple υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

• με τιμή: Δηλώνεται με τη λεκτική μονάδα in. Αλλαγές στην τιμή της δεν επιστρέφονται στο πρόγραμμα που κάλεσε τη συνάρτηση

• με αναφορά: Δηλώνεται με τη λεκτική μονάδα inout. Κάθε αλλαγή στη τιμή της μεταφέρεται αμέσως στο πρόγραμμα που κάλεσε τη συνάρτηση .

Στην κλήση μίας συνάρτησης οι πραγματικοί παράμετροι συντάσσονται μετά από τις λέξεις κλειδιά in και inout, ανάλογα με το αν περνούν με τιμή ή αναφορά.

Κανόνες εμβέλειας

Καθολικές ονομάζονται οι μεταβλητές που δηλώνονται στο κυρίως πρόγραμμα και είναι προσβάσιμες σε όλους. Τοπικές είναι οι μεταβλητές που δηλώνονται σε μία συνάρτηση ή διαδικασία και είναι προσβάσιμες μόνο μέσα από τη συγκεκριμένη συνάρτηση ή διαδικασία. Κάθε συνάρτηση ή διαδικασία, εκτός των τοπικών μεταβλητών, των παραμέτρων της και των καθολικών μεταβλητών, έχει επίσης πρόσβαση και στις μεταβλητές που έχουν δηλωθεί σε συναρτήσεις ή διαδικασίες προγόνους ή και σαν παραμέτρους αυτών.

Ισχύει ο δημοφιλής κανόνας ότι, αν δύο (ή περισσότερες) μεταβλητές ή παράμετροι έχουν το ίδιο όνομα και έχουν δηλωθεί σε διαφορετικό επίπεδο φωλιάσματος, τότε οι τοπικές μεταβλητές και παράμετροι υπερκαλύπτουν τις μεταβλητές και παραμέτρους των προγόνων, οι οποίες με τη σειρά τους υπερκαλύπτουν τις καθολικές μεταβλητές.

Μία συνάρτηση ή διαδικασία έχει δικαίωμα να καλέσει τον εαυτό της και όποια συνάρτηση βρίσκεται στο ίδιο επίπεδο φωλιάσματος με αυτήν, της οποίας η δήλωση προηγείται στον κώδικα

Η γραμματική της Cimple

program : program ID block .
block : declarations subprograms statements
declarations : (declare varlist ;) *
varlist : ID (, ID)*
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]*
```

Ένα μικρό πρόγραμμα σε cimple έχει ως εξής:

```
program fibonacci
        declare x;
        function fibonacci(in x)
        {
                return (fibonacci(in x-1)+fibonacci(in x-2));
        }
        # main #
        {
                input(x);
                print(fibonacci(in x));
        }.
```

Περιγραφή του μεταγλωττιστή της Cimple

Ο μεταγλωττιστής, έχει υλοποιηθεί σε γλώσσα προγραμματισμού python, δέχεται ως όρισμα στην γραμμή εντολών ένα πρόγραμμα γραμμένο σε γλώσσα Cimple, με κατάληξη ".ci". Αν για κάποιο λόγο ο χρήστης δώσει αρχείο που δεν είναι σε γλώσσα Cimple ο μεταγλωττιστής εμφανίζει στην οθόνη το κατάλληλο διαγνωστικό μήνυμα. Στη συνέχεια, και φυσικά με την προϋπόθεση της εισόδου ενός αρχείου που τηρεί τις αρχές της Cimple θα παραχθούν δύο ακόμη αρχεία ένα καταλήξεως ".int" που θα περιέχει μέσα τις τετράδες του ενδιάμεσου κώδικα (αναλυτικές λεπτομέρειες θα δοθούν παρακάτω) και ένα ισοδύναμο πρόγραμμα με το αρχικό γραμμένο σε γλώσσα assembly ".asm" που μπορεί κάποιος να "τρέξει" στον MIPS. Τέλος, παράγεται ακόμα ένα ισοδύναμο αρχείο σε γλώσσα C ".c" όταν στον κώδικα μας δεν υπάρχει κάποιο "Subprogram"(Function/Procedure).

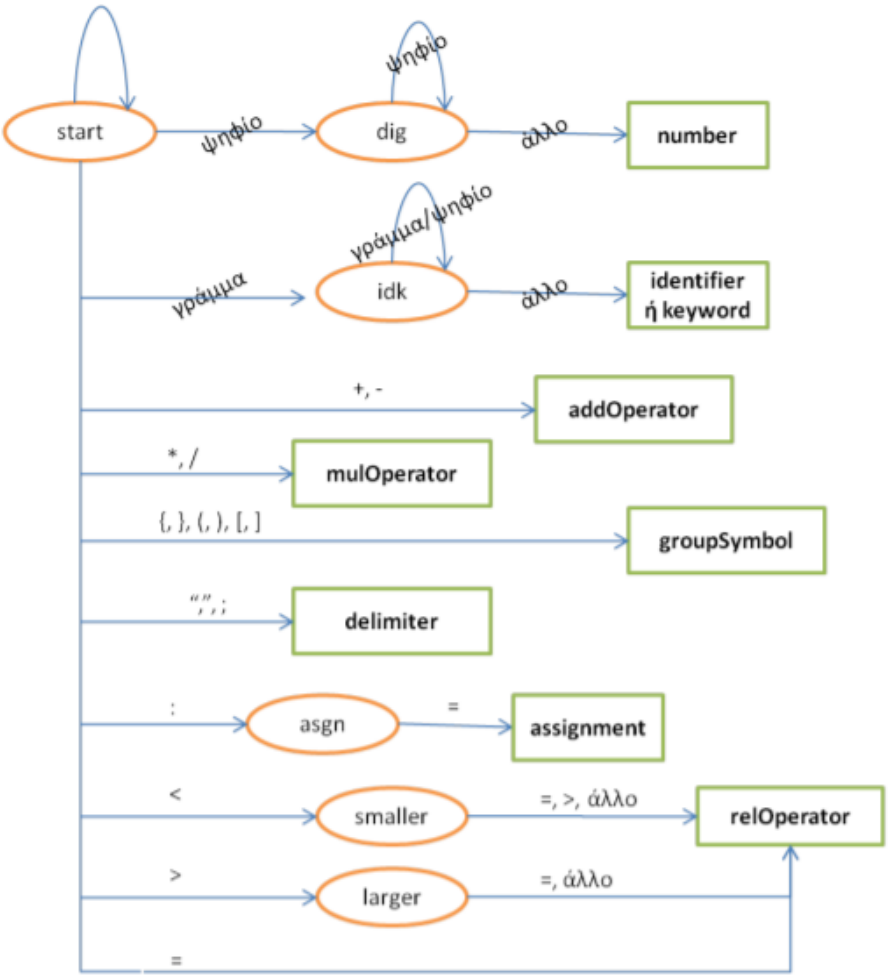
Τα στάδια μεταγλώττισης με τα οποία έχουμε ασχοληθεί είναι τα εξής:

- Λεκτική ανάλυση
- Συντακτική ανάλυση
- Σημασιολογική ανάλυση
- Παραγωγή ενδιάμεσου κώδικα
- Παραγωγή τελικού κώδικα

2)ΛΕΚΤΙΚΗ ΑΝΑΛΥΣΗ

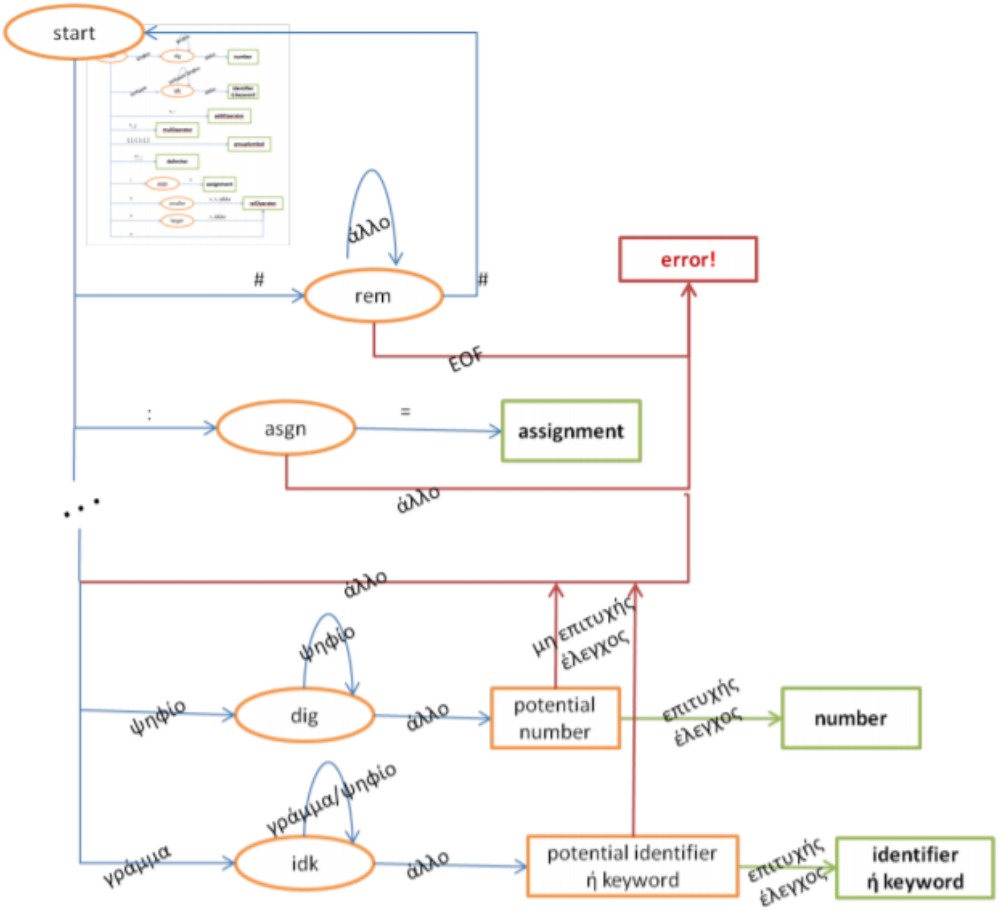
Η φάση της λεκτικής ανάλυσης αποτελείται ουσιαστικά από την υλοποίηση του λεκτικού αναλυτή συγκεκριμένα στο πρόγραμμα μας **def lex().** Συγκεκριμένα, καλείται απο τον συντακτικό αναλυτή καταναλώνει έναν έναν τους χαρακτήρες του προγράμματος που έχουμε δώσει ως είσοδο και επιστρέφει λεκτικές μονάδες.Στη δική μας περίπτωση ο **lex()** ενημερώνει κατάλληλα τις καθολικές μεταβλητές **tokenType** και **tokenString** δηλαδή το είδος της λεκτικής μονάδας και την ίδια την λεκτική μονάδα αντίστοιχα προκειμένου να χρησιμοποιηθούν από τον συντακτικό αναλυτή. Η λειτουργία πάνω στην οποία βασίζεται η υλοποίηση του **lex()** είναι το παρακάτω αυτόματο στοίβας :

Ξεκινά από μια αρχική κατάσταση με την κατανάλωση ενός χαρακτήρα πηγαίνει σε κάποια άλλη μέχρι να φτάσει σε κάποια τελική κατάσταση και να ενημερώσει κατάλληλα τις καθολικές μεταβλητές **tokenType** και **tokenString**.Ενώ στην περίπτωση λάθους που σχετίζεται με τις λεκτικές μονάδες π.χ τα σχόλια δεν έκλεισαν ποτέ ,δόθηκε μεγαλύτερος ακέραιος απο τον επιτρεπτό κτλ…(βλέπε το αυτόματο στην επόμενη σελίδα) να το ανιχνεύσει να ειδοποιήσει το χρήστη με το κατάλληλο μήνυμα και να τερματίσει η λειτουργία.



Σχήμα 2: Το αυτόματο λειτουργίας του λεκτικού αναλυτή

Κατα την λεκτική ανάλυση αναγνωρίζονται τα σφάλματα που σχετίζονται με τις λεκτικές μονάδες σύμφωνα με το παρακάτω αυτόματο:



Σχήμα 4: Διαχείριση σφαλμάτων από τον λεκτικό αναλυτή

Υλοποίηση του λεκτικού αναλυτή:

Αρχικά το αρχείο εισόδου αποθηκεύεται στην καθολική μεταβλητή myReads την οποία θα επεξεργαστούμε μεσα στον **lex()** διαχειρίζοντας την σαν έναν πίνακα που έχει μέσα κάθε χαρακτήρα του αρχείου, αυξάνοντας τον κατάλληλο counter όταν πρέπει, προκειμένου να πάρουμε τον επιθυμητό χαρακτήρα και να καταλάβουμε σε ποιά κατάσταση βρισκόμαστε ή άλλες φορές για να "κρυφοκοιτάξουμε" σε παρακάτω χαρακτήρες και να καταλάβουμε αν πρέπει να πάμε σε τελική κατάσταση και να ενημερώσουμε τις καθολικές μεταβλητές ή να συνεχίσουμε να καταναλώνουμε χαρακτήρες ή ακόμα αν πρόκειται για κάποιο σφάλμα.Μέσα στο λεκτικό αναλυτή ενημερώνεται ακόμα και η καθολική μεταβλητή **lineNo** προκειμένου

είμαστε σε θέση να δώσουμε καλύτερα διαγνωστικά μηνύματα. Είχε γίνει προσπάθεια αυτοματοποίησης όλων των ελέγχων που πρέπει να κάνει ο **lex()** όταν διαβάσει κάποιο χαρακτήρα ή και πριν, δίνοντας το κατάλληλο όρισμα στην συνάρτηση validationAndBoundaryCheck() ,έτσι ώστε να μην επαναλαμβάνουμε τα ίδια κομμάτια κώδικα πολλές φορές ωστόσο ποτέ δεν ολοκληρώθηκε στο σύνολο αλλά υλοποιήθηκε για 3 περιπτώσεις ελέγχου. Τα ”filesize” ,“id”, “num” πραγματοποιούν έλεγχο μες το **lex()** για το αν έχουμε καταναλώσει όλους τους χαρακτήρες αφου αν ο μετρητής μας ξεπεράσει το size της μεταβλητής-πίνακα που έχουμε (myReads) θα υπάρξει indexError οπότε με την κλήση της συνάρτησης αυτης με παράμετρο “filesize” προβλέπουμε το παραπάνω σφάλμα ενώ ενημερώνουμε κατάλληλα τις καθολικές μεταβλητές πως βρισκόμαστε στο τέλος του αρχείου, αν είμαστε στην περίπτωση “id” καθώς και στην “num” ελέγχουμε για το αν έχουμε ξεπεράσει το size του πίνακα και προκειμένου να αποφύγουμε το IndexErrror ενημερώνουμε τις καθολικές μεταβλητές και τερματίζουμε.

Ο **lex()** πριν διαβάσει κάποιο χαρακτήρα κάνει πάντα κλήση της παραπάνω συνάρτησης για να προβλεφθεί το IndexError. Στη συνέχεια διαβάζει έναν χαρακτήρα και αρχικά ελέγχει αν ο χαρακτήρας που διάβασε είναι η αλλαγή γραμμής προκειμένου να αυξήσει το μετρητή αλλαγής γραμμής(**lineNo**). Η κατάσταση “start” του αυτόματου μας αντιπροσωπεύεται απο την while(myChar.isspace()) : όπου εκεί μέσα όσο λαμβάνει whitespace χαρακτήρες τους προσπερνάει και ενημερώνει κατάλληλα τον μετρητή του πίνακα ή τον αριθμό γραμμής αν βρεί αλλαγή γραμμής ενώ παράλληλα έλεγχος γίνεται για να προβλεφθεί το IndexError του πίνακα. Έπειτα ,για οποιοδήποτε άλλο χαρακτήρα υπάρχει μια σειρά εντολών απόφασης που αναλαμβάνει να βρει ποιά είναι η λεκτική μονάδα που αναπαριστάται και να ενημερώσει κατάλληλα τις καθολικές μεταβλητές. Είναι, απαραίτητο να αναφερθεί ότι σε σημεία του κώδικα που προκειμένου να καταλάβουμε σε ποιά λεκτική μονάδα βρισκόμαστε πρέπει να καταναλώσουμε παραπάνω από ένα χαρακτήρες όταν καταναλώνεται ο δεύτερος χαρακτήρας γίνεται πάντα έλεγχος για το αν φτάσαμε στο τέλος του πίνακα. Ενώ, περισσότεροι έλεγχοι πραγματοποιούνται σε σημεία όπως η κατάσταση του αυτομάτου με όνομα “idk” ή στην “dig” προκειμένου να καθοριστεί εάν πρόκειται για κάποιο id ή για κάποιο ψηφίο αντίστοιχα. Αν έχουμε βρεί σαν πρώτο χαρακτήρα κάποιο γράμμα, αυξάνουμε τον μετρητή του πινακα και πηγαίνουμε στην κατάσταση “idk”

```
while (myReads[counter].isalpha() or myReads[counter].isdigit()) :
```

Εκεί αυξάνουμε συνεχώς κατα 1 τον μετρητή του πίνακα διαβάζουμε χαρακτήρες όσο ισχύει αυτη η συνθήκη πραγματοποιώντας και τον έλεγχο που προαναφέρθηκε. Όταν βγούμε από αυτή την κατάσταση μειώνουμε τον μετρητή κατα 1 για να μην χάσουμε το χαρακτήρα που μας οδήγησε στην έξοδο. Ελέγχουμε αν η λέξη που έχει παραχθεί ανήκει στις δεσμευμένες λέξεις ή άν ξεπερνάει το μέγιστο επιτρεπτό μήκος λέξης και ενημερώνουμε κατάλληλα τις καθολικές μεταβλητές ή αντίστοιχα προβάλλουμε ένα μήνυμα λάθους και τερματίζουμε. Παρόμοια είναι και η λειτουργία της κατάστασης του αυτομάτου “dig”. Αν έχουμε βρεί σαν πρώτο χαρακτήρα κάποιο ψηφίο, αυξάνουμε τον μετρητή του πινακα και πηγαίνουμε στην κατάσταση “dig”:

```
while (myReads[counter].isdigit()) :
```

Εκεί αυξάνουμε συνεχώς κατα 1 τον μετρητή του πίνακα διαβάζουμε χαρακτήρες όσο ισχύει αυτη η συνθήκη πραγματοποιώντας και τον έλεγχο του IndexError. Όταν βγούμε από αυτή την κατάσταση μειώνουμε τον μετρητή κατα 1 για να μην χάσουμε το χαρακτήρα που μας

οδήγησε στην έξοδο. Τέλος, ελέγχουμε αν ο αριθμός που σχηματίστηκε ανήκει στο επιτρεπτό όριο. Αν ναι, ενημερώνουμε τις καθολικές μεταβλητές και συνεχίζουμε αλλιώς προβάλλουμε το κατάλληλο μήνυμα σφάλματος και τερματίζουμε. Ακόμα, μια ενδιαφέρουσα κατάσταση του αυτομάτου είναι εκείνη των σχολίων. Μόλις ο **lex()** διαβάσει "#" αυξάνεται ο μετρητής κατα 1 και μπαίνουμε σε ένα infinite-loop στο οποίο διαβάζουμε χαρακτήρες δίχως να τους αποθηκεύουμε αφού δεν μας ενδιαφέρουν καθώς πρόκειται για σχόλια , μέχρι να βρούμε ξανά "#" ή να βρούμε το χαρακτήρα αλλαγής γραμμής τότε το πρόγραμμα τερματίζει καθώς αυτο σημαινει οτι τα σχόλια δεν έκλεισαν ποτέ αφού η Cimple δεν υποστηρίζει τα πολλαπλά σχόλια με τον τρόπο " #* - *#" . Άρα αν έχουν ανοίξει σχόλια πρέπει οπωσδήποτε να έχουν κλείσει πριν την αλλαγή γραμμής. Επίσης, σε όλες τις καταστάσεις που ο πρώτος χαρακτήρας μπορεί με την σειρά του να οδηγήσει σε διαφορετική κατάσταση όπως βλέπουμε στο αυτόματο(π.χ οι χαρακτήρες "<" ,">","=") έχουμε ορίσει σε αυτές μια μεταβλητή για να δούμε τον επόμενο χαρακτήρα.`nextChar = myReads[counter + 1]` Με αυτό το τρόπο κρυφοκοιτάμε τον επόμενο χαρακτήρα δίχως να τον καταναλώσουμε και αν μας οδηγεί σε κάποια από τις επόμενες καταστάσεις ενημερώνουμε τις καθολικές μεταβλητές και καταναλώνουμε 2 χαρακτήρες. Έναν για το nextChar και ένα προκειμένου στην επόμενη κλήση του **lex()** να διαβαστεί καινούργιος χαρακτήρας.Όλες οι υπόλοιπες καταστάσεις δεν έχουν κάποια ιδιαίτερη λειτουργία αν ο χαρακτήρας που διαβάστηκε ισούται με κάποιο από τα σύμβολα της cimple ενημερώνονται οι καθολικές μεταβλητές και ο μετρητής αυξάνεται κατα 1.

3)ΣΥΝΤΑΚΤΙΚΗ ΑΝΑΛΥΣΗ

Η συντακτική ανάλυση πραγματοποιείται μέσα από ένα σύνολο συναρτήσεων και έχει ώς στόχο να διαπιστωθεί εάν το πηγαίο πρόγραμμα ανήκει στην γλώσσα Cimple. Αυτό γίνεται μέσω της γραμματικής LL1 και πραγματοποιόντας αναδρομική κατάβαση. Μας έχει δοθεί ένα σύνολο από κανόνες -συναρτήσεις που πρέπει να υλοποιήσουμε. Συναρτήσεις μπορούν να καλέσουν άλλες συναρτήσεις μέχρι κάποια από αυτές να καταλήξει στο σύμβολο τερματισμού της. Εάν δεν συμβεί αυτό το πρόγραμμα τερματίζει με το κατάλληλο μήνυμα το οποίο περιέχει μέσα όλες τις απαραίτητες πληροφορίες για το σφάλμα δηλαδή τον τύπο του λάθους ,την γραμμή που συνέβει και τον χαρακτήρα στον οποίο το πρόγραμμα τερμάτισε. Οι κανόνες που ακολουθούνται βρίσκονται στην σελίδα 4 του παρόντος εγγράφου. Για παράδειγμα, για τον κανόνα **block : declarations subprograms statements** έχουμε

```
def block(name):
    declarations()
    subprograms()
    statements()
```

Ενώ για τον κανόνα **program : program ID block .** έχουμε

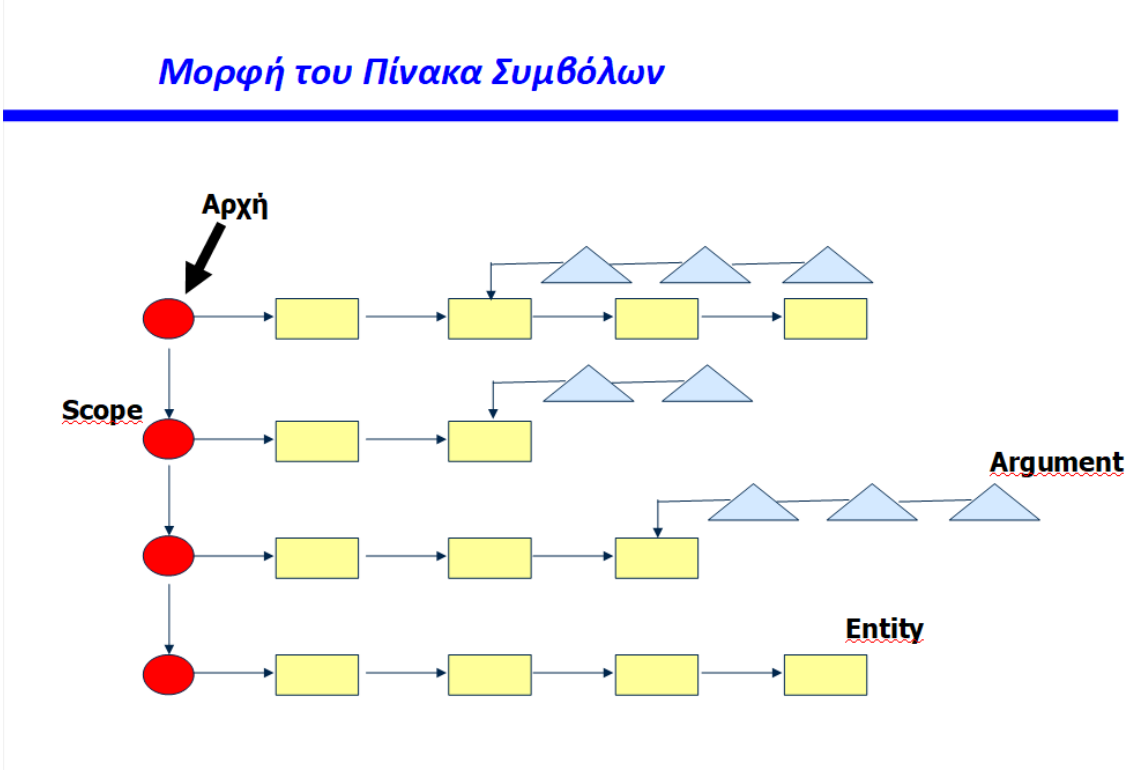
```
def program():
  lex()
  if (program_Tk == tokenType) :
     lex()
     if (id_Tk == tokenType) :
        lex()
        block()
```

```
            if(period_Tk == tokenType) :
                lex()
                if(eof_Tk == tokenType) :
                    lex()
                else :
                    print("Syntax Error in line: " + str(lineNo) + " Cimple does not support code after '.' Program crashed in : " + myChar + "\n")
            else:
                print("Syntax Error in line: " + str(lineNo) + " Period '.' expected Program crashed in : " + myChar + "\n")
                sys.exit()
        else:
            print("Syntax Error in line: " + str(lineNo) + " name of program expected Program crashed in : " + myChar + "\n")
            sys.exit()
    else:
        print("Syntax Error in line: " + str(lineNo) + " Keyword 'program' expected Program crashed in : " + myChar + "\n")
        sys.exit()
```

Και γενικότερα, όλοι οι κανόνες που υπάρχουν στην γραμματική της cimple στον κώδικα μας αποτελούν υλοποιημένες συναρτήσεις που όλες μαζί αποτελούν τον Συντακτικο Αναλυτή.

4)ΣΗΜΑΣΙΟΛΟΓΙΚΗ ΑΝΑΛΥΣΗ

Στόχος της ανάλυσης αυτής στον κώδικα μας είναι να αποτρέψουμε την χρήση κάποιου ονόματος για δευτερη φορα στο ίδιο επίπεδο, κάθε μεταβλητή η συνάρτηση που χρησιμοποιείται να έχει δηλωθεί, στα οποία θα μας βοηθήσει ο **πίνακας συμβόλων**.



Ακόμα, μέσω της σημασιολογικής ανάλυσης ελέγχεται αν κάθε συνάρτηση έχει ένα return ,αν τα return βρίσκονται μόνο σε function. Συναρτήσεις και Κλάσεις που βοηθούν στην σημασιολογική ανάλυση και ουσιαστικά στην δημιουργία του πίνακα συμβόλων είναι οι εξής:

- class Argument(): Με πεδία :
  - self.name
  - self.type
  - self.parMode

Για κάθε Argument Object, κρατάμε το ονομά του,τον τύπο του (η cimple υποστηρίζει μόνο Int) καθώς και το αν η παράμετρος περνάει με τιμή ή με αναφορά.Τα αντικείμενα αυτά δημιουργούνται μέσα στην formalparitem() και μπαίνουν στην argumentList στο ανάλογο Entity τύπου Subprogram ενώ αξίζει να σημειωθεί ότι στο ίδιο entity προστίθεται και ο τύπος του argument στο πεδίο argType που αποτελεί μια λίστα έτσι ώστε να κάνουμε εύκολα έλεγχο παρακάτω . Πρόκειται για τα γαλάζια τρίγωνα του παραπάνω σχήματος.

```
if (inOrInout==in_Tk):
  arg.parameterType = 'CV'
  scopeList[-1].entityList[-1].subprogram["argType"].append("in")
else:
  arg.parameterType = 'REF'
  scopeList[-1].entityList[-1].subprogram["argType"].append("inout")
scopeList[-1].entityList[-1].subprogram["argumentList"].append(arg)
```

- class Entity():
  - def __init__(self):
    - self.name
    - self.type
    - self.variable
    - self.subprogram
    - self.parameter
    - self.tempVar

Για κάθε Entity Object, κρατάμε ένα πεδίο με το όνομά του,τον τύπο του Entity(μπορεί να είναι “VAR”,”SUBPR”,”PARAM”,”TEMP”),τα παρακάτω πεδία διαμορφώνονται ανάλογα με τον τύπο του Entity και αποτελούν λεξικά με τις απαραίτητες πληροφορίες που χρειάζεται να έχει ο κάθε τύπος.

```
self.variable = {"type": "Int", "offset": 0}
self.subprogram = {"type": "", "startQuad": 0, "frameLength": 0, "argumentList": [],"nestingLevel":0,"argType": []}
self.parameter = {"mode": "", "offset": 0}
self.tempVar = {"type": "Int", "offset": 0}
```

Αν το Entity είναι μεταβλητή δημιουργούμε ένα Entity τύπου VAR μέσα στην συνάρτηση varlist() καθώς εκεί γίνεται η δήλωση μεταβλητών και συμπληρώνουμε με τιμές το ανάλογο λεξικό.

```
entity = Entity()
entity.type = 'VAR'
entity.name = tokenString
entity.variable["offset"] = calculateOffset()
scopeList[-1].entityList.append(entity)
```

Αν το Entity() είναι προσωρινή μεταβλητή ακολουθείται η ίδια διαδικασία με την διαφορά ότι η δημιουργία του Entity καθώς και η αρχικοποίηση του Λεξικού του πραγματοποιείται στην συνάρτηση newTemp() αφού εκεί δημιουργούνται νέες προσωρινές μεταβλητές.

```
entity = Entity()
entity.type = 'TEMP'
entity.name = tempVar
entity.tempVar["offset"] = calculateOffset()
scopeList[-1].entityList.append(entity)
```

Στην περίπτωση που το Entity() είναι κάποιο Subprogram το πεδίο “self.subprogram” δίνει τιμές μέσα στο λεξικό του στην ανάλογη συνάρτηση “Subprogram”.

```
entity = Entity()
entity.type = 'SUBPR'
entity.name = myToken
entity.subprogram["type"] = 'Function' Ενώ αν πρόκειται για Procedure 'Procedure'
entity.subprogram["nestingLevel"]=scopeList[-1].nestingLevel +1
scopeList[-1].entityList.append(entity)
```

Τέλος, αν πρόκειται για παράμετρο το ανάλογο πεδιο δέχεται τιμές μέσα στην block().
Ελεγχουμε αν είμαστε στο scope της Main. Κάνουμε entities τις παραμέτρους των subprogramms.Και τα βάζουμε στην global scopeList.

```
if scopeList[-1].nestingLevel > 0:
  for arg in scopeList[-2].entityList[-1].subprogram["argumentList"]:
      entity = Entity()
      entity.name = arg.name
      entity.type = 'PARAM'
      entity.parameter["mode"] = arg.parameterType
      entity.parameter["offset"] = calculateOffset()
      scopeList[-1].entityList.append(entity)
```

```
class Scope():
    ● def __init__(self):
        ○ self.name
        ○ self.entityList
        ○ self.nestingLevel
```

Για κάθε Scope object κρατάμε το όνομά του το βάθος φωλιάσματος του και μια λίστα από entities που περιέχονται στο scope.Όλα τα scopes είναι μέσα σε μια global scopeList
Οι συναρτήσεις και τα τμήματα κώδικα που αλληλεπιδρούν με τις παραπάνω κλάσεις και βοηθούν στην δημιουργία του πίνακα συμβόλων είναι οι εξής:

```
def addScope
        nextScope = Scope()
        nextScope.name = name
        scopeList.append(nextScope)
        if (len(scopeList)>1):
                scopeList[-1].nestingLevel = scopeList[-2].nestingLevel +1
```

Προσθέτουμε ένα νέο scope στην global scopeList και αρχικοποιούμε το βάθος φωλιασματός του ελέγχοντας πρώτα αν πρόκειται για το scope της main . Η συνάρτηση αυτή καλείται μέσα στην block().

```
def deleteScope
        del scopeList[-1]
```

Διαγράφουμε το τελευταίο scope απο την λίστα. Η συνάρτηση αυτή καλείται μέσα στην block().

```
def calculateOffset
        counter = 0
        if (scopeList[-1].entityList != []):
                for entity in (scopeList[-1].entityList):
                        if (entity.type == 'VAR' or entity.type == 'TEMP' or entity.type ==
                'PARAM'):
                        counter += 1
```

```
        offset = 12 + (counter * 4)
        return offset
```

Ουσιαστικά υπολογίζει τα bytes των Entities. Αν στο τρέχων Scope έχω τουλάχιστον ένα Entity το οποίο έχει τύπο είτε “VAR”,”TEMP”,”PARAM” όχι subprogram καθώς αυτά δεν έχουν offset,υπολογίζει το offset του Entity.

Για να δώσουμε τιμές στο startQuad του κάθε Entity τύπου Subprogram ελέγχουμε μέσα στην block() αν βρισκόμαστε μέσα στην main και αν όχι υπολογίζουμε την τετράδα από την οποία ξεκινά το subprogram.

```
if scopeList[-1].nestingLevel > 0:
  scopeList[-2].entityList[-1].subprogram["startQuad"] = nextQuad()
```

Για να υπολογίσουμε το frameLength απο κάθε subprogram μέσα στην block()

```
if(period_Tk==tokenType):
  genQuad("halt","_","_","_")
else :
  scopeList[-2].entityList[-1].subprogram["frameLength"] = calculateOffset()
```

Όταν τελειώνει ένα block και αν δεν είναι αυτό της main δίνουμε τιμή στο πεδίο frameLength...Η τιμή ενός Subprogram που μόλις τελείωσε είναι η τιμή του τελευταίου Entity που είχε +4.

Τέλος, έχει υλοποιηθεί η συνάρτηση **symbolTable**() η οποία καλείται ακριβώς πριν την διαγραφή ενός Scope μέσα στην **block**() προκειμένου να οπτικοποιήσουμε τον πίνακα συμβόλων εμφανίζοντας τα scopes,entities και arguments.

Προκειμένου να γίνει έλεγχος για το εάν στο τρέχων Scope χρησιμοποιείται ήδη ένα όνομα είτε σαν υποπρόγραμμα είτε σαν μεταβλητή επιτυγχάνεται μέσω της checkIfExistsAlready η οποία καλείται μέσα στην Varlist() αλλά και στην subprogram

```
def checkIfExistsAlready(name):
  global scopeList
  for i in scopeList[-1].entityList:
      if (i.name==name):
          print("Syntax Error in line: " + str(lineNo) + " you cant use same name twice " + name
+ "\n")
          sys.exit()
  return True
```

Προκειμένου να γίνει έλεγχος για το αν ένα όνομα υπάρχει μέσα στον πίνακα συμβόλων δηλαδή αν έχει δηλωθεί καθώς και αν έχει δηλωθεί σαν subprogram ή σαν μεταβλητή χρησιμοποιείται η συνάρτηση :

```
def searchIfExists(name):
  global scopeList
  scope=scopeList[-1]
  j = 1
```

```
  while scopeList != []:
      for i in scope.entityList:
          if (i.name==name):
              return scope,i
      j += 1
      if (j > len(scopeList)):
          break
      scope = scopeList[-j]
  if name.isdigit():
      return "digit",name
  print("not found in symbol table : " + str(name))
  print(lineNo)
  sys.exit()
```

Γίνεται χρήση της στις συναρτήσεις **assignStat** και **actualparitem** στην **callStat** και γενικά σε συναρτήσεις που δίνουμε id στις οποίες ελέγχεται και αν το όνομα αυτό έχει δηλωθεί ήδη σαν συνάρτηση:

```
scope, entity = searchIfExists(tokenString)
if (entity.type == "SUBPR"):
  print("\nSyntax Error in line: " + str(lineNo) +"  this name is defined as function \n")
  sys.exit()
```

Έλεγχος για αν κάθε συνάρτηση έχει return αλλά και αν μόνο οι συναρτήσεις έχουν return γίνεται μέσω δυο καθολικών μεταβλητών `global hasReturn,hasFunction`

Λειτουργούν σαν flags μέσα στις συναρτήσεις **returnStat**

```
hasReturn=1
if (hasFunction<=0):
    print("Syntax Error in line: " + str(lineNo) + " found return outside of function Program crashed in : " + myChar + "\n")
  sys.exit()
else:
  hasFunction-=1
```

και αντίστοιχα στην **Subprogram** όταν έχουμε μπεί στον κώδικα των Function:

```
if (function_Tk==tokenType) :
  hasFunction+=1
……
……
…...
if (hasReturn!=1):
    print("Syntax Error in line: " + str(lineNo) + " did not found return statement inside Function Program crashed in : " + myChar + "\n")
  sys.exit()
else:
  hasReturn=0
```

```
def checkSubProgramPars(name,funcPars):
  scope, entity = searchIfExists(name)
  args=entity.subprogram["argumentList"]
  if len (funcPars) != len(args):
      return False
  tempList=[]
  for i in funcPars:
      if i == "CV":
          tempList.append("in")
      elif i == "REF":
          tempList.append("inout")
  tempList2=entity.subprogram["argType"]
  for i,j in zip(tempList,tempList2):
      if i != j :
          return False
  return True
```

Η παρακάτω συνάρτηση υλοποιείται για να ελέγξουμε αν οι παράμετροι με τις οποίες καλούνται οι συναρτήσεις είναι ακριβώς αυτές με τις οποίες έχουν δηλωθεί και με τη σωστή σειρά. Καλείται μέσα στην callstat() και την idtail() και αποτελεί συνθήκη για να γίνει το `genQuad("call",myId,"_","_")` παίρνει σαν παράμετρο το όνομα της συνάρτησης η της διαδικασίας μέσω του searchIfExists έχουμε πρόσβαση στα πεδία του entity και υλοποιούμε τον έλεγχο.

```
def searchArgName(argList,arg):
  counter=0
  for i in range(0,len(argList)):
      if arg.name == argList[i]:
          counter+=1
          if counter==2:
              return True
  return False
```

Η συνάρτηση αυτή υλοποιείται για να ελέγξουμε μέσα στην formalparitem() αν κάποιο όρισμα απο αυτά με τα οποία έχει δηλωθεί η συνάρτηση χρησιμοποιείται δύο φόρές. Δέχεται σαν όρισμα μια λίστα με όλες τις παραμέτρους που έχουν δοθεί στην συνάρτηση και ένα argument object και ελέγχει αν το όνομα του argument υπάρχει 2 φορές στην λίστα.

```
def symbolTable():
```

Η συνάρτηση αυτή είναι υπεύθυνη για την εκτύπωση του πίνακα συμβόλων ακριβώς πριν την διαγραφή ενός scope ενώ παράλληλα γράφει σε ένα αρχείο “.txt” τον πίνακα συμβόλων.

5)ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες οι οποίες αποτελούνται από έναν τελεστή και τρία τελούμενα για παράδειγμα (+,a,b,t_1). Στις τετράδες αυτές υπάρχει αρίθμηση, κάθε μία έχει έναν μοναδικό αριθμό που την χαρακτηρίζει και με βάση αυτούς

τους αριθμούς κατα σειρά εκτελούνται οι εντολές εκτός και αν υπάρξει τετράδα που υποδεικνύει κάτι άλλο (π.χ τετράδα με εντολή “jump”). Μπορούμε να πάρουμε τετράδες πολλών κατηγοριών :

- op,x,y,z

Το op μπορεί να είναι ένα εκ των (+,-,*,/). Το (x,y) μπορεί να είναι όνομα μεταβλητής ή αριθμητική σταθερά και το z μπορεί να είναι όνομα μεταβλητής.Το αποτέλεσμα της τετράδας αυτής είναι να γίνει η πράξη “op” μεταξύ του x , y και το αποτέλεσμα να εκχωρηθεί στο z.

- :=,x,_,z

Τετράδα εκχώρησης. Το (x,y) μπορεί να είναι όνομα μεταβλητής ή αριθμητική σταθερά και το z μπορεί να είναι όνομα μεταβλητής. Το αποτέλεσμα της τετράδας αυτής είναι η τιμή του χ να εκχωρηθεί στο z μέσω του τελεστή εκχώρησης :=.

- jump,_,_,z

Τετράδα με τελεστή άλματος χωρίς συνθήκη.Το αποτέλεσμα της τετράδας αυτής είναι η μεταπήδηση στην τετράδα με αριθμό z

- relOp,x,y,z

Όπου relOp ένα εκ των (=, <, >, <>, <=, >=).Το αποτέλεσμα της τετράδας αυτής είναι η μεταπίδηση στην τετράδα με αριθμό z αν ισχύει η συνθήκη relOp μεταξύ των x , y.

- begin_block,name,_,_ end_block,name,_,_ halt,_,_,_

Τετράδες που υποδηλώνουν την αρχή και το τέλος μιας ενότητας. Η begin_block,name υποδηλωνει την αρχή του προγράμματος η μιας συνάρτησης με όνομα name. Η end_block,name υποδηλωνει την το τέλος του προγράμματος η μιας συνάρτησης με όνομα name. Η halt,_,_,_ υποδηλώνει το τέλος του προγράμματος και γίνεται κλήση της πάντα πριν το end_block του προγράμματος.

- par,x,m,_ call,name,_,_ ret,x,_,_

Οι τετράδες αυτές αφορούν συναρτήσεις και διαδικασίες. Η τετράδα par,x,m,_ όπου χ παράμετρος συνάρτησης και m ο τρόπος μετάδοσης : <<“CV” μετάδοση με τιμή>>, <<REF μετάδοση με αναφορά>>,<<RET επιστροφη τιμής συνάρτησης>>. call,name,_,_ υποδηλώνει κλήση συνάρτησης με όνομα name . Η ret,x,_,_ υποδηλώνει επιστροφή τιμής συνάρτησης.

Για την παραγωγή του ενδιάμεσου κώδικα χρησιμοποιήθηκαν οι εξής βοηθητικές συναρτήσεις όπως έγινε υπόδειξη στο μάθημα (Π.Ι Μεταφραστές Ακ.Ετος 2021) :

- nextquad()
  - επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί
- genquad(op, x, y, z)
  - δημιουργεί την επόμενη τετράδα (op, x, y, z)
- newtemp()
  - δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή
  - οι προσωρινές μεταβλητές είναι της μορφής

    T_1, T_2, T_3 ...

- emptylist()
  - δημιουργεί μία κενή λίστα ετικετών τετράδων
- makelist(x)
  - δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x
- merge($list_1$, $list_2$)
  - δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών $list_1$, $list_2$
- 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

Τις συναρτήσεις αυτές τις χρησιμοποιούμε μέσα στις συναρτήσεις που έχουν δημιουργηθεί για την συντακτική ανάλυση σύμφωνα με τις υποδείξεις που έχουν γίνει στο μάθημα (Π.Ι Μεταφραστές Ακ.Ετος 2021) :Παραγωγή Ενδιάμεσου Κώδικα .

Παραδείγματα:
incase **{P1}**
(case (condition)**{P2}** statements1**{P3}**) ***{P4}**
Όπου **{P1}** = w = newTemp()
startQuad = nextQuad()
genQuad(":=", "1", "_", w)
{P2}= backpatch(CPlace[0], nextQuad())
{P3}=genQuad(":=", "0", "_", w)
backpatch(CPlace[1], nextQuad())
{P4}=genQuad("=", w,"0", startQuad)
Αρχικά με την είσοδο μάς στην incase δημιουργούμε μια νέα προσωρινή μεταβλητή με την χρήση της **newTemp()** ας την πούμε **T_1** και την καταχωρούμε στο **w**. Σημειώνουμε την επόμενη τετράδα που θα δημιουργηθεί με την χρήση της **nextQuad()** την οποία χρειαζόμαστε για να δημιουργήσουμε την επανάληψη που υλοποιεί η δομή incase. Καλώντας την **genQuad()** δημιουργούμε μια τετράδα τύπου**(:=,1,_,w)** η οποία θα έχει το label που σημειώθηκε στην αρχή. Στη συνέχεια θα εκτελεστεί η **condition()** η οποία θα δημιουργήσει δυο τετράδες μια για το αν η συνθήκη ισχύει και μια για την αντίθετη περίπτωση.Εδώ κάνοντας χρήση της **backpatch()** συμπληρώνουμε την τετράδα που αντιστοιχεί στην αληθή συνθήκη να δείχνει στην επόμενη τετράδα που θα δημιουργηθεί ας πούμε 40.Κάνοντας εκ νέου χρήση της **genQuad()** δημιουργούμε την τετράδα **(:=,0,_,w)** η οποία θα έχει και αυτή label 40. Έπειτα, εκτελείται η **statements()** η οποία θα δημιουργήσει τις δικές της τετράδες. Τώρα, κάνουμε χρήση της **backpatch()** για να ενημερώσουμε τις τετράδες που επέστρεψε η condition false να δείχνουν στην επόμενη τετράδα. Τέλος με τη χρήση της **genQuad()** δημιουργούμε την τετράδα **(=,w,0,startQuad)** άρα αν θα έχει δημιουργηθεί τετράδα με w=0

σημαίνει ότι εκτελέστηκε κάποιο statement άρα μεταφερόμαστε στην αρχή της incase δηλαδή στο label που σημειώσαμε στην αρχή.

forcase **{P1}**
    (case (condition)**{P2}** statements1**{P3}**) *
    default statements2
Όπου {P1}=startQuad = nextQuad() {P2}=backpatch(CPlace[0], nextQuad())
{P3}=genQuad("jump", "_", "_",startQuad) backpatch(CPlace[1], nextQuad())

```
def forcaseStat() :
  if (forcase_Tk == tokenType):
      lex()
      startQuad = nextQuad()
      while (case_Tk == tokenType):
          lex()
          if (left_Paren_Tk == tokenType):
              lex()
              CPlace=condition()
              backpatch(CPlace[0], nextQuad())
              if (right_Paren_Tk == tokenType):
                  lex()
                  statements()
                  genQuad("jump", "_", "_",startQuad)
                  backpatch(CPlace[1], nextQuad())
              else:
                  print("Syntax Error in line: " + str(lineNo) + " right parenthesis  misisng Program
crashed in : " + myChar + "\n")
                  sys.exit()
          else:
                print("Syntax Error in line: " + str(lineNo) + " left parenthesis   misisng Program
crashed in : " + myChar + "\n")
              sys.exit()
      if (default_Tk == tokenType):
          lex()
          statements()
      else:
          print("Syntax Error in line: " + str(lineNo) + " problem with 'Default' Program crashed
in : " + myChar + "\n")
          sys.exit()
  else:
      print("Syntax Error in line: " + str(lineNo) + " \n")
      sys.exit()
```

Αρχικά σημειώνουμε την αρχή της forcase αυτό γίνεται στο {P1}.Στη συνέχεια στο CPlace=condition() παίρνουμε μία λίστα με 2 τετράδες την condition.true και την condition.false αντίστοιχα στις οποίες λείπει το τελευταίο στοιχείο της τετράδας. Πρόκειται

για τετράδες τύπου (<,a,10,_)και μια (jump,_,_,_).Αρχικά γίνεται backpatch η condition.true στην περίπτωση μας η πρώτη και προσθέτει στο τέλος της το nextquad δηλαδή που πρέπει να πάει αν ισχύει η συνθήκη backpatch(CPlace[0], nextQuad()) .Εκτελούνται τα statements αν η συνθήκη ισχύει και γίνεται επιστροφή στην αρχή της forcase και αυτός ειναι ο λόγος που σημειώσαμε την αρχή της genQuad("jump","_","_",startQuad) . Ακόμα συμπληρώνουμε και την περίπτωση που η condition είναι false backpatch(CPlace[1], nextQuad()) με τον αριθμό του επόμενου quad. Ψάχνουμε εκ νεου να βρούμε condition που ισχύουν αν κάποια συνθήκη ήταν αληθής. Αν δεν βρούμε καμιά αληθής συνθήκη θα εκτελεστούν τα default και θα μεταφερθούμε έξω από την forcase.

Και οι υπόλοιπες συναρτήσεις του συντακτικού αναλυτή έχουν τροποποιηθεί για να παράγουν τον ενδιάμεσο κώδικα ανάλογα με την παραπάνω διαδικασία ακολουθώντας τα πρότυπα του pdf του μαθήματος που αναφέρθηκε παραπάνω.

Σε αυτή τη φάση μπορούμε να ελέγξουμε αν όλα είναι σωστά στον κώδικα μας με τη χρήση δύο αρχείων ένα με κατάληξη ".int" το οποίο περιέχει όλες τις τετράδες του ενδιάμεσου κώδικα καθώς και τα label τους, και ένα με κατάληξη ".c" το οποίο είναι το πρόγραμμα που προκύπτει απο τις τετράδες του ενδιάμεσου κώδικα σε γλώσσα C. Το αρχείο αυτό παράγεται μόνο εάν το πρόγραμμα μας δέν έχει μέσα κάποιο Subprogram το οποίο διαπιστώνεται ύστερα από κατάλληλο έλεγχο με μία καθολική μεταβλητή τύπου boolean η οποία αρχικά έχει την τιμή **TRUE** και αν κληθεί η συνάρτηση subprogram παίρνει την τιμή **FALSE** με αποτέλεσμα να αποτρέπει τον κώδικα από το να παράξει το αρχείο.

6)ΠΑΡΑΓΩΓΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ

Κάθε εντολή του ενδιάμεσου κώδικα παράγει τις αντίστοιχες εντολές του τελικού κώδικα. Ο κώδικας που παράγεται μπορεί να ελεγχθεί για την λειτουργία του στον επεξεργαστή MIPS. Σε αυτή τη φάση γίνεται η απεικόνιση των μεταβλητών στην μνήμη (στοίβα) και το πέρασμα παραμέτρων και η κλήση συναρτήσεων. Ο κώδικας που παράγεται αποθηκεύεται σε ένα ".asm" αρχείο. Κύρια συνάρτηση της φάσης αυτής είναι η **genFinalCode()** η οποία καλείται μέσα στην συνάρτηση **block()** την στιγμή που έχει παραχθεί ενδιάμεσος κώδικας για μια ολόκληρη συνάρτηση ή για τον κώδικα της main.

Έχουν δημιουργηθεί τρεις βοηθητικές συναρτήσεις όπως υποδείχθηκε στο μάθημα (Π.Ι Μεταφραστές Ακ.Ετος 2021) :Παραγωγή Τελικού Κώδικα

```
def gnvlcode(name):
    global scopeList,asmFile
    scope,entity=searchIfExists(name)
    asmFile.write("lw $t0,-4($sp)\n")
    for i in range(0,(scopeList[-1].nestingLevel-scope.nestingLevel)-1):
        asmFile.write("lw $t0,-4($t0)\n")
    if (entity.type=="PARAM"):
        offset = entity.parameter["offset"]
    elif(entity.type=="VAR"):
        offset = entity.variable["offset"]
    asmFile.write("addi $t0,$t0,-"+str(offset)+"\n")
```

Η συνάρτηση αυτή είναι υπεύθυνη για να γράψει στον καταχωρητή $t0 την διεύθυνση της μη τοπικής μεταβλητής name που δίνεται ως παράμετρος. Μέσω της **searchIfExists()** βρίσκουμε το **entity** καθώς και το **nestingLevel** της συγκεκριμένης μεταβλητής. Γράφουμε στο αρχείο μας την εντολή που μας δίνει την στοίβα του γονέα asmFile.write("lw $t0,-4($sp)\n") . Επαναληπτικά, γράφουμε στο αρχείο μας ("lw $t0,-4($t0)\n") οι φορές αντιστοιχούν στο **[(τωρινό βάθος φωλιάσματος) - (βάθος φωλιάσματος του προγόνου που έχει τη μεταβλητη)]-1** γιατί για τον "πατέρα" έχουμε παράγει. Προσθέτουμε το offset της μεταβλητής name το οποίο βρήκαμε απο τον πίνακα συμβόλων στον $t0 asmFile.write("addi $t0,$t0,-"+str(offset)+"\n") και το γράγουμε στο αρχείο.

```
def loadvr(v,r):
  global scopeList,asmFile
  scope,entity=searchIfExists(v)
  if (scope=="digit"):
      asmFile.write("li $t"+str(r)+","+str(v)+"\n")
  elif(entity.type=="VAR"):
      if (scope.nestingLevel==0):
          asmFile.write("lw $t"+str(r)+",-"+str(entity.variable["offset"])+"($s0)"+"\n")
      elif(scope.nestingLevel==scopeList[-1].nestingLevel):
          asmFile.write("lw $t"+str(r)+",-"+str(entity.variable["offset"])+"($sp)"+"\n")
      else :
          gnvlcode(v)
          asmFile.write("lw $t"+str(r)+",($t0)"+"\n")
  elif(entity.type=="TEMP"):
      if (scope.nestingLevel == 0):
          asmFile.write("lw $t" + str(r) + ",-" + str(entity.tempVar["offset"]) + "($s0)" + "\n")
      elif(scope.nestingLevel==scopeList[-1].nestingLevel):
          asmFile.write("lw $t" + str(r) + ",-" + str(entity.tempVar["offset"]) + "($sp)" + "\n")
  elif(entity.type=="PARAM" and entity.parameter["mode"]=="CV"):
      if (scope.nestingLevel == scopeList[-1].nestingLevel):
          asmFile.write("lw $t" + str(r) + ",-" + str(entity.parameter["offset"]) + "($sp)" + "\n")
      elif (scope.nestingLevel < scopeList[-1].nestingLevel):
          gnvlcode(v)
          asmFile.write("lw $t" + str(r) + ",($t0)" + "\n")
  elif (entity.type == "PARAM" and entity.parameter["mode"] == "REF"):
      if (scope.nestingLevel == scopeList[-1].nestingLevel):
          asmFile.write("lw $t0,-"+ str(entity.parameter["offset"]) + "($sp)" + "\n")
          asmFile.write("lw $t"+str(r)+",($t0)"+"\n")
      elif (scope.nestingLevel < scopeList[-1].nestingLevel):
          gnvlcode(v)
          asmFile.write("lw $t0,($t0)\n")
          asmFile.write("lw $t"+str(r)+",($t0)"+"\n")
```

Μέσω της συνάρτησης αυτής γράφεται στον καταχωρητή r η τιμή της v. Ουσιαστικά, ψάχνουμε να βρούμε μέσω αυτής την τιμή της v. Αρχικά, μέσω της **searchIfExists()** βρίσκουμε το **entity** καθώς και το **scope** της. Μέσα απο μία σειρά εντολων απόφασης προσπαθούμε να διακρίνουμε την περίπτωση που βρισκόμαστε ελέγχουμε κυρίως τον τύπο του entity της μεταβλητής v.

- Στην αρχή γίνεται ένας έλεγχος για το αν είναι αριθμητική σταθερά και αν συμβαίνει αυτό προσθέτουμε στο αρχείο την εντολή **li $tr,v**
- Αν είναι καθολική μεταβλητή γράφουμε στο αρχείο την εντολή **lw $tr,-offset($s0)**
- Αν είναι τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή και έχει δηλαδή το entity το ίδιο βάθος φωλιάσματος με αυτο που αναλύεται τώρα ή προσωρινή μεταβλητή γράφουμε στο αρχείο την εντολή **lw $tr,-offset($sp)**
- Αν είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον καλούμε την **gnlvcode(v)** και γράφουμε στο αρχείο **lw $tr,($t0)**
- Αν είναι τυπική παράμετρος που περνάει με αναφορά και βαθος φωλιάσματος ίσο με το τρέχον τότε γράφουμε στο αρχείο **lw $t0,-offset($sp)** και **lw $tr,($t0)**
- Αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον καλούμε την **gnlvcode(v)** και γράφουμε στο αρχείο **lw $t0,($t0)** **lw $tr,($t0)**

```
def storerv(r,v):
  global scopeList, asmFile
  scope, entity = searchIfExists(v)
  if(entity.type=="VAR"):
     if (scope.nestingLevel==0):
        asmFile.write("sw $t"+str(r)+",-"+str(entity.variable["offset"])+"($s0)"+"\n")
     elif(scope.nestingLevel==scopeList[-1].nestingLevel):
        asmFile.write("sw $t"+str(r)+",-"+str(entity.variable["offset"])+"($sp)"+"\n")
     else :
        gnvlcode(v)
        asmFile.write("sw $t"+str(r)+",($t0)"+"\n")
  elif(entity.type=="TEMP"):
     if (scope.nestingLevel == 0):
          asmFile.write("sw $t" + str(r) + ",-" + str(entity.tempVar["offset"]) + "($s0)" +
"\n")
     elif(scope.nestingLevel==scopeList[-1].nestingLevel):
          asmFile.write("sw $t" + str(r) + ",-" + str(entity.tempVar["offset"]) + "($sp)" +
"\n")
  elif(entity.type=="PARAM" and entity.parameter["mode"]=="CV"):
     if (scope.nestingLevel == scopeList[-1].nestingLevel):
```

```
            asmFile.write("sw $t" + str(r) + ",-" + str(entity.parameter["offset"]) + "($sp)" +
"\n")
        elif (scope.nestingLevel < scopeList[-1].nestingLevel):
            gnvlcode(v)
            asmFile.write("sw $t" + str(r) + ",($t0)" + "\n")
    elif (entity.type == "PARAM" and entity.parameter["mode"] == "REF"):
        if (scope.nestingLevel == scopeList[-1].nestingLevel):
            asmFile.write("lw $t0,-"+ str(entity.parameter["offset"]) + "($sp)" + "\n")
            asmFile.write("sw $t"+str(r)+",($t0)"+"\n")
        elif (scope.nestingLevel < scopeList[-1].nestingLevel):
            gnvlcode(v)
            asmFile.write("lw $t0,($t0)\n")
            asmFile.write("sw $t"+str(r)+",($t0)"+"\n")
```

Μέσω της συνάρτησης αυτής μεταφέρονται δεδομένα από τον καταχωρητή r στην μνήμη (μεταβλητή v). Μέσα από μία σειρά εντολων απόφασης προσπαθούμε να διακρίνουμε την περίπτωση που βρισκόμαστε, ελέγχουμε τον τύπο του entity της μεταβλητής v. Μέσω της **searchIfExists()** βρίσκουμε το **entity** καθώς και το **scope** της.

- Αν είναι καθολική μεταβλητή τότε γράφουμε στο αρχείο μας την εντολή **sw $tr,-offset($s0)**
- Αν είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή γράφουμε στο αρχείο μας την εντολή **sw $tr,-offset($sp)**
- Αν είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον γράφουμε στο αρχείο μας τις εντολές **lw $t0,-offset($sp)** **sw $tr,($t0)**
- Αν είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον καλούμε την **gnlvcode(v)** και γράφουμε στο αρχείο μας την εντολή **sw $tr,($t0)**
- Αν είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον καλούμε την **gnlvcode(v)** και γράφουμε στο αρχείο μας τις εντολές **lw $t0,($t0)** **sw $tr,($t0)**

Η συνάρτηση **genFinalCode()** αποτελεί την κύρια συνάρτηση παραγωγής τελικού κώδικα. Γίνεται κλήση αυτής της συνάρτησης μέσα στην block() όταν μια συνάρτηση έχει μόλις κλείσει το block της δηλαδή έχει κληθεί `genQuad("end_block", name, "_", "_")` ωστόσο μας ενδιαφέρει να μην έχει διαγραφεί το scope για αυτό την καλούμε πρίν γίνει κλήση της **deleteScope()**. (Καθώς, οι διαφάνειες στο website που αναφερόμαστε και παραπάνω δεν έχουν ενημερωθεί ακόμα δεν μπορεί να δοθεί σύνδεσμος για το ακριβές pdf στο οποίο μπορούμε να αναφερθούμε επεξηγηματικά για τον κώδικα της **genFinalCode()**. Θα δοθεί σύνδεσμος παλαιότερου εγγράφου προκειμένου να πάρετε πληροφορίες.Παραγωγή Τελικού Κώδικα. Για την επεξήγηση της συνάρτησης οι εντολές που σας ενδιαφέρουν βρίσκονται απο την **σελίδα 19** μέχρι το τέλος. Οι διαφάνειες που αναφέρονται παρακάτω είναι από το pdf του Ακ.Ετους 2021.

Αρχικά διασχίζουμε τις μέχρι στιγμής τετράδες και γράφουμε στο αρχείο μας το Label κάθε επανάληψης.

```
def genFinalCode():
  global quadList,scopeList,finalCounter,asmFile
  for i in range(len(quadList)):
      count = quadList[i][0]
      op = quadList[i][1]
      x = quadList[i][2]
      y = quadList[i][3]
      z = quadList[i][4]
      asmFile.write("L%d: \n" % (count))
```

Εάν βρήκαμε **jump** γράφουμε στο αρχείο την παρακάτω εντολή.(Διαφάνεια 29)

```
if(op=="jump"):
  asmFile.write("b L"+str(z)+"\n")
```

Εάν έχουμε βρεί **relational operator** τύπου (=,<>,>,<,<=,>=) (Διαφάνεια 29)

```
elif (op in relOps):
  loadvr(x,1)
  loadvr(y,2)
  asmFile.write(relOps.get(op) + ",$t1,$t2, L"+str(z)+"\n")
```

Όπου relOps : `relOps = {"=" :"beq","<>":"bne",">" :"bgt","<":"blt",">=":"bge","<=":"ble"}`

Εαν έχουμε βρεί ":=" (Διαφάνεια 30)

```
elif (op== ":="):
  loadvr(x,1)
  storerv(1,z)
```

Εάν έχουμε βρεί **operator** τυπου (+,-,*,\) (Διαφάνεια 31)

```
elif (op in operators):
  loadvr(x, 1)
  loadvr(y, 2)
  asmFile.write(operators.get(op) + ",$t1,$t1,$t2"+"\n")
  storerv(1, z)
```

Όπου operators: `operators = {"+":"add","-":"sub","*" :"mul","/":"div"}`

Εάν βρώ "out" (Διαφάνεια 32)

Αρχικά γράφουμε στο αρχείο την παρακάτω εντολή .Κάνουμε load(x,1) για να πάρουμε την τιμή του χ από την μνήμη και να την δώσουμε στον $t1 και στην συνέχεια κάνοντας move $a0,$t1 την τοποθετούμε στον καταχωρητή $α0.

```
elif(op == "out"):
  asmFile.write("li $v0,1" + "\n")
  loadvr(x, 1)
  asmFile.write("move $a0,$t1" + "\n")
  asmFile.write("syscall" + "\n")
```

Εάν έχω βρεί "inp" (Διαφάνεια 32)

Μέσω της move μεταφέρουμε την τιμή του $v0 στον $t1 και κάνοντας store thn αποθηκευω στη μνήμη.

```
elif (op == "inp"):
  asmFile.write("li $v0,5" + "\n")
  asmFile.write("syscall" + "\n")
  asmFile.write("move $t1,$v0" + "\n")
  storerv(1, x)
```

Εάν βρήκα “retv”:(Διαφάνεια 33)

Αποθηκεύεται ο x στη διεύθυνση που είναι αποθηκευμένη στην 3η θέση του εγγραφήματος δραστηριοποίησης

```
elif (op == "retv"):
  loadvr(x, 1)
  asmFile.write("lw $t0,-8($sp)"+ "\n")
  asmFile.write("sw $t1,($t0)"+ "\n")
```

Εάν βρήκα “par”(Διαφάνεια 35)

Υπάρχει στην αρχή μια καθολική μεταβλητή που παίρνει τιμές boolean και μέσω αυτής βεβαιωνόμαστε πώς η διαδικασία εύρεσης του framelength θα γίνει μόνο μία φορά για κάθε subprogram. Χρησιμοποιούμε μια μεταβλητή **finalCounter** σαν μετρητή για να βρούμε το πλήθος ορισμάτων. Μέσω μιας δομής επανάληψης ψάχνουμε απο το στοιχείο “i” που βρισκόμαστε αυτή τη στιγμή προς τα κάτω στην quadList μέχρι να βρώ το “call” στην τετράδα εκείνη βρίσκεται και το όνομα του υποπρογράμματος που με ενδιαφέρει. Αφού βρώ το όνομα το περνάμε σαν παράμετρο στην συνάρτηση searchIfExists() προκειμένου να βρούμε το scope και το entity του ονόματος του υποπρογράμματος και έτσι και το frameLength. Πριν από την πρώτη παράμετρο, τοποθετούμε τον $fp να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί add $fp,$sp,framelength

```
elif (op == "par"):
  if (checkForPar == True):
     checkForPar=False
     for j in range(i,len(quadList)):
        if (quadList[j][1] == "call"):
           FuncOrProcName = str(quadList[j][2])
           break
     scope,entity=searchIfExists(FuncOrProcName)
     asmFile.write("addi $fp, $sp,"+str(entity.subprogram["frameLength"])+ "\n")
     finalCounter=0
```

Όσο είμαστε μέσα στο “par” έχουμε πάλι μια ακολουθία εντολών απόφασης

Αν εκεί το y ισούται με “CV” (Διαφάνεια 36)

```
if (y=="CV"):
  loadvr(x, 0)
  asmFile.write("sw $t0,-" + str(12 + 4 * finalCounter) + "($fp)\n")
  finalCounter+=1
```

Αν εκεί το y ισούται με “REF” (Διαφάνεια 36)

Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή addi $t0,$sp,-offset sw $t0,-(12+4i)($fp) (Διαφάνεια 37)

```
elif (y == "REF"):
  scope,entity=searchIfExists(x)
  if (scope.nestingLevel==scopeList[-1].nestingLevel):
      if (entity.type=="VAR"):
          asmFile.write("addi $t0,$sp,-"+str(entity.variable["offset"])+"\n")
          asmFile.write("sw $t0,-"+str(12+4*finalCounter)+"($fp)\n")
      elif (entity.type=="PARAM" and entity.parameter["mode"]=="CV"):
          asmFile.write("addi $t0,$sp,-" + str(entity.parameter["offset"]) + "\n")
          asmFile.write("sw $t0,-" + str(12 + 4 * finalCounter) + "($fp)\n")
```

Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά (Διαφάνεια 40)

```
      elif (entity.type == "PARAM" and entity.parameter["mode"] == "REF"):
  asmFile.write("lw $t0,-"+str(entity.parameter["offset"])+"($sp) \n")
  asmFile.write("sw $t0,-" + str(12 + 4 * finalCounter) + "($fp) \n")
```

Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά (Διαφάνεια 44)

```
elif(scope.nestingLevel<scopeList[-1].nestingLevel):
      gnvlcode(x)
      if(entity.type == "PARAM" and entity.parameter["mode"] == "REF"):
          asmFile.write("lw $t0,($t0)\n")
          asmFile.write("sw $t0,-"+str(12 + 4 * finalCounter)+"($fp)\n")
```

Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή (Διαφάνεια 42)

```
          else:
  asmFile.write("sw $t0,-" + str(12 + 4 * finalCounter) + "($fp)\n")
```

Στο τέλος της y=”REF” αυξάνουμε κατα 1 τον finalCounter

Αν το y=”RET”

Γεμίζουμε το 3ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης με τη διεύθυνση της προσωρινής μεταβλητής στην οποία θα επιστραφεί η τιμή

```
elif (y == "RET"):
  scope,entity=searchIfExists(x)
  asmFile.write("addi $t0,$sp,-"+str(entity.tempVar["offset"])+"\n")
  asmFile.write("sw $t0,-8($fp)\n")
```

Εάν βρήκα “call”

Αρχικά γεμίζουμε το 2ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης, τον σύνδεσμο προσπέλασης, με την διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα της, ώστε η κληθείσα να γνωρίζει που να κοιτάξει αν χρειαστεί

να προσπελάσει μία μεταβλητή την οποία έχει δικαίωμα να προσπελάσει, αλλά δεν της ανήκει
Η μεταβλητή checkForPar γίνεται True. Μέσω της searchIfExists() βρίσκω το scope,entity του X.
Αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο γονέα lw $t0,-4($sp) sw $t0,-4($fp) (Διαφάνεια 48)

```
elif (op == "call"):
  checkForPar=True
  scope, entity = searchIfExists(x)
  print(entity.subprogram["startQuad"])
  if (entity.subprogram["nestingLevel"]==scopeList[-1].nestingLevel):
     asmFile.write("lw $t0,-4($sp)\n")
     asmFile.write("sw $t0,-4($fp)\n")
```

Αν καλούσα και κληθείσα έχουν διαφορετικό βάθος φωλιάσματος, τότε η καλούσα είναι ο γονέας της κληθείσας sw $sp,-4($fp) (Διαφάνεια 51)

```
  elif(scopeList[-1].nestingLevel<entity.subprogram["nestingLevel"]):
      asmFile.write("sw $sp,-4($fp)\n")
```

Στη συνέχεια μεταφέρουμε τον δείκτη στοίβας στην κληθείσα addi $sp,$sp,framelength (Διαφάνεια 52)

```
asmFile.write("addi $sp,$sp,"+str(entity.subprogram["frameLength"])+"\n")
```

Καλούμε τη συνάρτηση(Διαφάνεια 52)

```
asmFile.write("jal L"+str(entity.subprogram["startQuad"])+"\n")
```

Και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα(Διαφάνεια 52)

```
asmFile.write("addi $sp,$sp,-"+str(entity.subprogram["frameLength"])+"\n")
```

Μέσα στην κληθείσα στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της την οποία έχει τοποθετήσει στον $ra η jal sw $ra,($sp) (Διαφάνεια 53)

```
elif(op=="begin_block" and scopeList[-1].nestingLevel!=0):
  asmFile.write("sw $ra,($sp)\n")
```

Στο τέλος κάθε συνάρτησης κάνουμε το αντίστροφο, 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 lw $ra,($sp) jr $ra (Διαφάνεια 53)

```
elif (op == "end_block" and scopeList[-1].nestingLevel != 0):
  asmFile.write("lw $ra,($sp)\n")
  asmFile.write("jr $ra\n")
```

Το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται, οπότε στην αρχή του προγράμματος χρειάζεται ένα άλμα που να οδηγεί στην πρώτη ετικέτα του κυρίως προγράμματος **j Lmain** η οποία πρέπει να δημιουργείτε όταν ξεκινά η μετάφραση της main. Μέσω τών .seek γράφουμε την ετικέτα της main στην αρχή του προγράμματος και στην συνέχεια επαναφέρουμε το αρχείο να γράφει στο τέλος.(Διαφάνεια 54)

```
elif (op == "begin_block" and scopeList[-1].nestingLevel == 0):
  asmFile.seek(0, os.SEEK_SET)
  asmFile.write("j L"+str(count)+"\n")
```

```
  asmFile.seek(0, os.SEEK_END)
```
Στη συνέχεια πρέπει να κατεβάσουμε τον $sp κατά framelength της main addi $sp,$sp,framelength (Διαφάνεια 54)
```
asmFile.write("add $sp,$sp,"+str(calculateOffset())+"\n")
```
Να σημειώσουμε στον $s0 το εγγράφημα δραστηριοποίησης της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές move $s0,$sp (Διαφάνεια 54)
```
asmFile.write("move $s0,$sp\n")
```
Στό τέλος της genFinalCode αδειάζουμε την quadList προκειμένου να μην ξαναγράψουμε τις ίδιες τετράδες.

7)TESTING

Το κυρίως πρόγραμμα για έλεγχο του compiler μας είναι το **finalTest.ci**

```
program finalTest
  declare temp1, temp2, temp3;
  procedure ultimateProcedure(in inPar, inout inOutPar1, inout inOutPar2)
    declare temp4, temp5, temp6,temp7;
    function testFunc()
      declare returnVar;
      {
      inOutPar1:=10;
      inOutPar2:=20;
      returnVar:=inOutPar1+inOutPar2;
      return(returnVar)
      }
    {  inPar:=testFunc();
      inOutPar1:=inPar+2;
      inOutPar2:=inOutPar1+4;
      temp4:=inOutPar2;
      temp6:=9;
      temp7:=45;
      print(inPar);
      if(inPar>4 or temp4 > inPar){
        forcase
            case (temp6<20) temp6:=temp6+10;
            default temp6:=temp6+5;
             ;
      }
      else{
        switchcase
             case(temp7<=0)
             {
               temp7:=temp6-2;
```

```
            }
            case(temp6>0)
            {
              print(temp7+3);
              temp7:=temp6+2;
            }
           default print(temp6);;
    };
  }
{  temp1:=55;
   temp2:=45;
  if(not [temp1>temp2]){
      temp3:=50;
    }
    else{
      temp3:=50;
    };
  call ultimateProcedure(in temp1, inout temp2, inout temp3);
  temp3:=47;
  print(temp3);
}.
```

Οι τετράδες του ενδιάμεσου κώδικα:

```
1: begin_block testFunc _ _
2: := 10 _ inOutPar1
3: := 20 _ inOutPar2
4: + inOutPar1 inOutPar2 T_1
5: := T_1 _ returnVar
6: retv returnVar _ _
7: end_block testFunc _ _
8: begin_block ultimateProcedure _ _
9: par T_2 RET _
10: call testFunc _ _
11: := T_2 _ inPar
12: + inPar 2 T_3
13: := T_3 _ inOutPar1
14: + inOutPar1 4 T_4
15: := T_4 _ inOutPar2
16: := inOutPar2 _ temp4
17: := 9 _ temp6
18: := 45 _ temp7
19: out inPar _ _
20: > inPar 4 24
21: jump _ _ 22
22: > temp4 inPar 24
23: jump _ _ 32
24: < temp6 20 26
25: jump _ _ 29
26: + temp6 10 T_5
27: := T_5 _ temp6
28: jump _ _ 24
29: + temp6 5 T_6
30: := T_6 _ temp6
31: jump _ _ 45
32: <= temp7 0 34
33: jump _ _ 37
34: - temp6 2 T_7
35: := T_7 _ temp7
36: jump _ _ 45
37: > temp6 0 39
38: jump _ _ 44
39: + temp7 3 T_8
40: out T_8 _ _
41: + temp6 2 T_9
42: := T_9 _ temp7
43: jump _ _ 45
44: out temp6 _ _
45: end_block ultimateProcedure _ _
46: begin_block finalTest _ _
47: := 55 _ temp1
48: := 45 _ temp2
49: > temp1 temp2 53
50: jump _ _ 51
51: := 50 _ temp3
52: jump _ _ 54
53: := 50 _ temp3
54: par temp1 CV _
55: par temp2 REF _
56: par temp3 REF _
57: call ultimateProcedure _ _
58: := 47 _ temp3
59: out temp3 _ _
60: halt _ _ _
61: end_block finalTest _ _
```

Ο πίνακας συμβόλων που παράγει

==========
==========
Current Scope: testFunc with nestingLevel:2
Current Entities:
Entity: returnVar type: VAR offset: 12
Entity: T_1 type: TEMP offset:16
Current Scope: ultimateProcedure with nestingLevel:1
Current Entities:
Entity: inPar type: PARAM mode: CV offset: 12
Entity: inOutPar1 type: PARAM mode: REF offset: 16
Entity: inOutPar2 type: PARAM mode: REF offset: 20
Entity: temp4 type: VAR offset: 24
Entity: temp5 type: VAR offset: 28
Entity: temp6 type: VAR offset: 32
Entity: temp7 type: VAR offset: 36
Entity: testFunc type: SUBPRSubprogram :Function startQuad :1 frameLength :20
Function Arguments:
Current Scope: finalTest with nestingLevel:0
Current Entities:
Entity: temp1 type: VAR offset: 12
Entity: temp2 type: VAR offset: 16
Entity: temp3 type: VAR offset: 20
Entity: ultimateProcedure type: SUBPR Subprogram: Procedure startQuad: 0 frameLength :0
Procedure Arguments:
Argument : inPar type :Int parameterType: CV
Argument : inOutPar1 type :Int parameterType: REF
Argument : inOutPar2 type :Int parameterType: REF
==========
==========
Scope deleted.
==========
==========
Current Scope: ultimateProcedure with nestingLevel:1
Current Entities:
Entity: inPar type: PARAM mode: CV offset: 12
Entity: inOutPar1 type: PARAM mode: REF offset: 16
Entity: inOutPar2 type: PARAM mode: REF offset: 20
Entity: temp4 type: VAR offset: 24
Entity: temp5 type: VAR offset: 28
Entity: temp6 type: VAR offset: 32
Entity: temp7 type: VAR offset: 36
Entity: testFunc type: SUBPRSubprogram :Function startQuad :1 frameLength :20
Function Arguments:

Entity: T_2 type: TEMP offset:40
Entity: T_3 type: TEMP offset:44
Entity: T_4 type: TEMP offset:48
Entity: T_5 type: TEMP offset:52
Entity: T_6 type: TEMP offset:56
Entity: T_7 type: TEMP offset:60
Entity: T_8 type: TEMP offset:64
Entity: T_9 type: TEMP offset:68
Current Scope: finalTest with nestingLevel:0
Current Entities:
Entity: temp1 type: VAR offset: 12
Entity: temp2 type: VAR offset: 16
Entity: temp3 type: VAR offset: 20
Entity: ultimateProcedure type: SUBPR Subprogram: Procedure startQuad: 8 frameLength :72
Procedure Arguments:
Argument : inPar type :Int parameterType: CV
Argument : inOutPar1 type :Int parameterType: REF
Argument : inOutPar2 type :Int parameterType: REF
==========
==========
Scope deleted.
==========
==========
Current Scope: finalTest with nestingLevel:0
Current Entities:
Entity: temp1 type: VAR offset: 12
Entity: temp2 type: VAR offset: 16
Entity: temp3 type: VAR offset: 20
Entity: ultimateProcedure type: SUBPR Subprogram: Procedure startQuad: 8 frameLength :72
Procedure Arguments:
Argument : inPar type :Int parameterType: CV
Argument : inOutPar1 type :Int parameterType: REF
Argument : inOutPar2 type :Int parameterType: REF
==========
==========
Scope deleted.

Process finished with exit code 0

Μπορούμε να παρατηρήσουμε αυτό που αναφέρθηκε στην σημασιολογική ανάλυση σε ότι αφορά το Framelength μιας συνάρτησης.(Όταν τελειώνει ένα block και αν δεν είναι αυτό της

main δίνουμε τιμή στο πεδίο frameLength...Η τιμή ενός Subprogram που μόλις τελείωσε είναι η τιμή του τελευταίου Entity που είχε +4.) Πράγματι,βλέπουμε ότι η returnVar σαν entity ηταν το τελευταίο στο scope της testFunc με offset 16 ενώ στο απο κάτω scope αυτο της ultimateProcedure() η testFunc σαν entity πλέον έχει framelength 20 δηλαδή όσο το τελευταίο entity στο scope της 4.

●Τεστ για το αν υπάρχει return μέσα σε function. Αν απο το αρχείο μας στην γραμμη 11 όπου βρίσκεται το return το αφαιρέσουμε θα πάρουμε το παρακάτω μήνυμα

Syntax Error in line: 13 did not found return statement inside Function Program crashed in : { Ο συντακτικός αναλυτής ψάχνει για return μέχρι να τελείωσει το block και μόλις καταλάβει ότι ανοίγει νέο block χωρίς να έχει βρεί return στο block της συνάρτησης μας ειδοποιεί και τερματίζει.

●Τεστ για το αν υπάρχει return μέσα σε procedure η σε κομμάτι του κυρίως προγράμματος. Αρκεί να προσθέσουμε ενα return statement σε κάποιο από τα σημεία που αναφέρθηκαν. Για παράδειγμα στο αρχείο μας προσθέτουμε πρώτα ένα return στο block της procedure πριν κλείσει το block της γραμμη 39, και λαμβάνουμε το παρακάτω μήνυμα :

Syntax Error in line: 39 found return outside of function Program crashed in : n

Το ίδιο θα συμβεί αν προσθέσουμε και ένα return μέσα στο block της main για παράδειγμα στη γραμμή 50 πριν το print :

Syntax Error in line: 50 found return outside of function Program crashed in : n

Μόλις ολοκληρωθεί η λεκτική μονάδα return και δεν βρίσκεται μέσα σε function το πρόγραμμα βγάζει τα παραπάνω μηνύματα και τερματίζει.

●Τεστ για το αν κάθε μεταβλητή ή συνάρτηση ή διαδικασία που έχει δηλωθεί να μην έχει δηλωθεί πάνω από μία φορά στο βάθος φωλιάσματος στο οποίο βρίσκεται. Όταν γίνονται τα declarations σε οποιοδήποτε βάθος φωλιάσματος δηλώσουμε το ίδιο όνομα μεταβλητής , για παράδειγμα στην ultimateProcedure() δηλωθεί 2 φορές η temp4 στη γραμμή θα πάρουμε το παρακάτω μήνυμα και το πρόγραμμα θα τερματίσει :

Syntax Error in line: 4 you cant use same name twice temp4

Επίσης αν πάμε να δηλώσουμε κάποιο subprogram στο ίδιο βάθος φωλιάσματος με το ίδιο όνομα με κάποιο άλλο θα συμβεί κάτι αντίστοιχο για παράδειγμα δημιουργώ μέσα στην στην ultimateProcedure() ακόμα μία συνάρτηση testFunc() θα πάρω το αντίστοιχο μήνυμα και το πρόγραμμα θα τερματίσει :

Syntax Error in line: 13 you cant use same name twice testFunc

●Τεστ για το αν κάθε μεταβλητή, συνάρτηση ή διαδικασία που χρησιμοποιείται έχει δηλωθεί και μάλιστα με τον τρόπο που χρησιμοποιείται (σαν μεταβλητή ή σαν συνάρτηση). Αρκεί μέσα σε κάποιο block να χρησιμοποιήσουμε μια μεταλητή που δεν έχει γίνει declare για παράδειγμα αναθέτω την τιμή 5 στην μεταβλητή tempVar γραμμή 42 :

Syntax Error in line: 42 not found in symbol table : tempVar Program crashed in : r

Λαμβάνουμε διαγνωστικό μήνυμα ότι η μεταβλητή αυτή δεν υπάρχει στον πίνακα συμβόλων και κατ επέκταση δεν έχει δηλωθεί και τερματίζει η λειτουργία.

Επίσης εάν δηλώσω ένα όνομα που έχει δηλωθεί σαν συνάρτηση για παράδειγμα στα declaratios της ultimateFunc (γραμμή 4) δηλώσω την μεταβλητή testFunc θα πάρουμε το παρακάτω μήνυμα και το πρόγραμμα θα τερματίσει :

Syntax Error in line: 5 you cant use same name twice testFunc

●Τεστ για το αν οι παράμετροι με τις οποίες καλούνται οι συναρτήσεις είναι ακριβώς αυτές με τις οποίες έχουν δηλωθεί και με τη σωστή σειρά. Αν για πάραδειγμα, αλλάξουμε την σειρά των ορισμάτων της ultimate Procedure (γραμμη 47) ή αν προσθέσουμε/αφαιρέσουμε κάποιο όρισμα θα πάρουμε το παρακάτω διαγνωστικό μήνυμα και θα τερματίσουμε.

Syntax Error in line: 47 parameters dosent much in Procedure: ultimateProcedure

Επίσης, αν προσθέσω κάποιο όρισμα στην κλήση της testFunc (γραμμή 13) επίσης θα πάρουμε διαγνωστικό μήνυμα και θα τερματίσουμε.

Syntax Error in line: 13 parameters dosent much in function: testFunc

●Τεστ για το αν οι παράμετροι με τις οποίες έχει δηλωθεί το υποπρόγραμμα έχουν διαφορετικά ονόματα. Αρκεί να δώσουμε ένα ίδιο όνομα μεταβλητής στην δήλωση του υποπρογράμματος μας (γραμμή 3) π.χ να αλλάξουμε το inOutPar2 σε inOutPar1 το πρόγραμμα θα μας δώσει το παρακάτω μήνυμα και θα τερματίσει.

Syntax Error in line: 3 You cant use same parameter name : inOutPar1

●Τεστ για το αν μια συνάρτηση χρησιμοποιείται έξω από το block στο οποίο έχει δηλωθεί. Αρκεί να καλέσουμε την testFunc() στο block της main. Για παράδειγμα, την καλώ στη (γραμμή 40) temp1:=testFunc(); το πρόγραμμα παράγει διαγνωστικό μήνυμα και τερματίζει.

Syntax Error in line: 40 not found in symbol table : testFunc Program crashed in :)

●Τεστ για το αν μια διαδικασία γίνεται ανάθεση σε κάποια μεταβλητή. Αρκεί να αναθέσουμε σε μια μεταβλητή στην main την διαδικασία ultimateProcedure() για παράδειγμα στη (γραμμή 41) να προσθέσουμε την εντολή temp1:=ultimateProcedure(in temp1, inout temp4, inout temp3); το πρόγραμμα μας δίνει ανάλογο διαγνωστικό μήνυμα και τερματίζει :

Syntax Error in line: 41 You cant assign a procedure in a variable :ultimateProcedure

●Αν μία μεταβλητή έχει δηλωθεί σε μικρότερο βάθος φωλιάσματος όταν βρισκόμαστε σε μεγαλύτερο μπορούμε να την δούμε και να την επεξεργαστούμε σαν καθολικη μεταβλητή.Όταν θα επανέλθουμε στο βάθος φωλιάσματος της μεταβλητής θα έχει την κανονική της τιμή.Μπορούμε να το καταλάβουμε αυτό αν τροποποιήσουμε λίγο το αρχείο μας

```
program finalTest
  declare temp1, temp2, temp3,temp10;
  procedure ultimateProcedure(in inPar, inout inOutPar1, inout inOutPar2)
    declare temp4, temp5, temp6,temp7;
    function testFunc()
      declare returnVar;
      {
      inOutPar1:=10;
      inOutPar2:=20;
      returnVar:=inOutPar1+inOutPar2;
      return(returnVar)
      }
    {  inPar:=testFunc();
      inOutPar1:=inPar+2;
      inOutPar2:=inOutPar1+4;
      temp4:=inOutPar2;
      temp6:=9;
```

```
    temp7:=45;
    print(temp10);
    temp10:=160;
    print(inPar);
    if(inPar>4 and temp4 > inPar){
      forcase
          case (temp6<20) temp6:=temp6+10;
          default temp6:=temp6+5;
          ;
    }
    else{
      switchcase
           case(temp7<=0)
           {
             temp7:=temp6-2;
           }
           case(temp6>0)
           {
             print(temp7+3);
             temp7:=temp6+2;
           }
         default print(temp6);;
    };
  }
{  temp1:=55;
   temp10:=100;
   temp2:=45;
  if(not [temp1>temp2]){
      temp3:=50;
    }
    else{
      temp3:=50;
    };
  call ultimateProcedure(in temp1, inout temp2, inout temp3);
  print(temp10);
  temp3:=47;
}.
```

Η μεταβλητή temp10 παίρνει στην main την τιμή 100 μέσα στην ultimateProcedure την κάνουμε print και έπειτα της δίνουμε την τιμή 160 και στο τέλος της main την ξανακάνουμε print.Βλέπουμε ότι η τιμή της στο τέλος είναι 160 .
Ωστόσο αν στο βάθος φωλιάσματος της ultimateProcedure δηλώσουμε μια μεταβλητή temp10 είναι εκείνη η μεταβλητή που επεξεργαζόμαστε όποτε δεν πρόκειται για πρόβλημα.

Αν χρησιμοποιούσαμε την μεταβλητή temp10 ακόμα και μέσα στην testFunc θα άλλαζε η μεταβλητλη της ultimateProcedure και όχι της main γιατί έχει πιο κοντινό βάθος φωλιάσματος με αυτό της testFunc.
(Η διαπίστωση αυτή έγινε τρέχοντας το αρχείο asm που παράγεται στον Mips παρατηρώντας τα print).
●Τεστ για το αν το file που δίνουμε στον compiler έχει κατάληξη ".ci". Αρκεί να δώσουμε ένα οποιοδήποτε άλλο file οποιασδήποτε κατάληξης.
I need a '.ci' file,I am a Cimple compiler
Process finished with exit code 1
●Τεστ για το αν μπορούμε να δώσουμε αριθμό μεγαλύτερο από το επιτρεπόμενο όριο της Cimple. Αρκεί να πάμε σε κάποια ανάθεση στο αρχείο μας και να δώσουμε έναν αριθμό μεγαλύτερο του 4294967295.0.Για παράδειγμα στη γραμμή 8 του αρχείου δίνουμε στην μεταβλητή inOutPar1 την τιμή 4294967296 και παίρνουμε το παρακάτω μήνυμα και τερματίζουμε :
Syntax Error in line: 8 Forbidden Number
●Τεστ για το επιτρεπόμενο όριο χαρακτήρων μιάς μεταβλητής.Αρκεί να δηλώσω μια μεταβλητή με πάνω απο 30 χαρακτήρες για παράδειγμα την "qwertyuioplkjhgfdsazxcvbnmmnbvc" με 31 χαρακτήρες στα declarations της main (γραμμή 2) **Syntax Error in line: 2 Unavailable id**
●Έχει γίνει έλεγχος για το αν παράγεται αρχειο σε γλωσσα C όπως μπορούμε να διαπιστώσουμε κανένα αρχείο δεν παράγεται αν τρέξουμε το πρόγραμμά μας καθώς έχει μέσα Subprogram. Ενώ αν του δώσουμε το παρακάτω πρόγραμμα :

```
program average
  declare x,sum,average,count;
  #main#
  {
    input(x);
    sum :=0;
    count:=0;
    while(x > 1)
    {
      sum := sum + x;
      count:= count+1;

    };
    average := sum/count;
    print(average);
  }.
```

Παράγεται το αρχείο :

```c
#include <stdio.h>

int main()
{
    int x,sum,average,count,T_1,T_2,T_3;
    L_1:  //(begin_block,average,_,_)
    L_2: scanf("x= %d",&x);
    L_3: sum=0; //(:=,0,_,sum)
    L_4: count=0; //(:=,0,_,count)
    L_5: if (x>1) goto L_7; //(>,x,1,7)
    L_6: goto L_12; //(jump,_,_,12)
    L_7: T_1=sum+x; //(+,sum,x,T_1)
    L_8: sum=T_1; //(:=,T_1,_,sum)
    L_9: T_2=count+1; //(+,count,1,T_2)
    L_10: count=T_2; //(:=,T_2,_,count)
    L_11: goto L_5; //(jump,_,_,5)
    L_12: T_3=sum/count; //(/,sum,count,T_3)
    L_13: average=T_3; //(:=,T_3,_,average)
    L_14: printf("average= %d", average);
    L_15: //(halt,_,_,_)
}
```

Ενώ από δίπλα μπορουμε να παρατηρήσουμε και τις τεράδες του ενδιάμεσου κώδικα.

8)BUGS/ERRORS

●Ενώ κατα την φάση του ενδιάμεσου κώδικα είχε παραδοθεί κώδικας που μπορούσε να αναγνωρίσει το αρνητικό πρόσημο και να το διαχειριστεί κατάλληλα μέσα στην συνάρτηση expression() τώρα αν δώσουμε αρνητικό πρόσημο θα πάρουμε το παρακάτω μήνυμα :
Syntax Error in line: not found in symbol table : -1 Program crashed in : {
Επειδή δεν έγινε διαχείριση του error αυτού το κομμάτι εκείνο του κώδικα έχει μπεί σε σχόλια και ακόμα και αν δώσουμε αρνητικό πρόσημο το δεχόμαστε σαν θετικό προκειμένου να συνεχίσει να λειτουργεί το πρόγραμμά μας.