ΑΝΑΦΟΡΑ ΓΙΑ ΤΟ PROJECT CIMPLE COMPILER

- 1η Φάση: Αποτελείται από το Λεκτικό και το Συντακτικό αναλυτή.
- 2η Φάση: Αποτελείται από τον Ενδιάμεσο Κώδικα.
- 3η Φάση: Αποτελείται από τον Πίνακα Συμβόλων και τον Τελικό Κώδικα.

Θα δούμε αναλυτικά την κάθε φάση και την διασύνδεση μεταξύ τους για να επιτύχουν τον τελικό σκοπό μας δηλαδή, την μετάφραση ενός προγράμματος γλώσσας Cimple.

Επιπλέον γίνεται η υλοποίηση μερικών test αρχείων σε γλώσσα Cimple για τον έλεγχο της ορθής λειτουργίας του Compiler.

Περιεχόμενα:

ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Ο Λεκτικός Αναλυτής(def lex()) λειτουργεί σαν αυτόματο καταστάσεων που ξεκινώντας από μια αρχική κατάσταση διαβάζει κάθε χαρακτήρα και πηγαίνει σταδιακά σε κάποια άλλη μέχρι να βρει μια τελική κατάσταση.

Το αυτόματο καταστάσεων αναγνωρίζει:

- 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 ([,], (,) , { , })
- Το σύμβολο τερματισμού του προγράμματος (.)
- Το σύμβολο διαχωρισμού σχολίων (#)
- Λάθη
- Τους λευκούς χαρακτήρες (tab, space, return)οι οποίοι 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
- Τις δεσμευμένες λέξεις της Cimple (program, declare, if, else, while, switchcase, forcase, incase, case, default, not, and, or, function, procedure, call, return, in, inout, input, print)

Κύρια λειτουργία του λεκτικού αναλυτή είναι να διαβάζει το πρόγραμμα που του δίνει ο χρήστης γράμμα – γράμμα και να επεξεργάζεται τις λεκτικές μονάδες του.

Επιστρέφει στον συντακτικό αναλυτή:

- τη λεκτική μονάδα
- έναν ακέραιο που χαρακτηρίζει τη λεκτική μονάδα

Τις λεκτικές μονάδες τις αρχικοποιούμε σαν αριθμούς στον κώδικά μας όπως φαίνεται και στην παρακάτω φωτογραφία δίνοντας τους έτσι ένα χαρακτηριστικό id.

```
4
5   wordid = 0
6   programid = 1
7   ifid = 2
8   switchcaseid = 3
9   notid = 4
10  functionid = 5
11  inputid = 6
12  declareid = 7
13  elseid = 8
14  forcaseid = 9
15  andid = 10
16  procedureid = 11
17  printid = 12
18  whileid = 13
19  incaseid = 14
20  orid = 15
21  callid = 16
22  caseid = 17
23  returnid = 18
24  defaultid = 19
25  inid = 20
26  inoutid = 21
27  finalNumbid = 22
28  plusid = 23
29  minusid = 24
30  multiid = 25
31  divideid = 26
32  equalsid = 27
33  lessOrEqualid = 28
34  notEqualid = 29
35  lessThanid = 30
36  moreOrEqualid = 31
37  moreThanid = 32
38  assignid = 33
39  commaid = 34
40  questionamarkid = 35
41  rightParenthesisid = 36
42  leftParenthesisid = 37
43  Doubleapostrofid = 38
44  # rightDoubleapostrofid = 39
45  leftSquareParenthesisid = 40
46  rightSquareParenthesisid = 41
47  leftWaveParenthesisid = 42
48  rightWaveParenthesisid = 43
49  eofid = 44
50  beginid = 45
51  endid = 46
```

Τέλος, ο Λεκτικός Αναλυτής καλέιται ως συνάρτηση μέσα στο Συντακτικό Αναλυτή και θα περιγράψουμε παρακάτω πως.

ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Ο συνολικός σκοπός του Συντακτικού Αναλυτή είναι να διαπιστώνει εάν το πρόγραμμα σε γλώσσα Cimple που δώθηκε σαν είσοδος από τον χρήστη ανήκει στη γλώσσα ή όχι.Στη συνέχεια, δημιουργεί το κατάλληλο «περιβάλλον» μέσα από το οποίο αργότερα θα κληθούν οι σημαντικές ρουτίνες. Υπάρχουν πολλοί τρόποι για να κατασκευαστεί ένας συντακτικός αναλυτής. Θα προτιμήσουμε τη συντακτική ανάλυση με αναδρομική κατάβαση. Βασίζεται σε γραμματική LL(1). Η γραμματική LL(1) αναγνωρίζει από αριστερά στα δεξιά, την αριστερότερη δυνατή παραγωγή και όταν βρίσκεται σε δίλλημα ποιον κανόνα να ακολουθήσει της αρκεί να κοιτάξει το αμέσως επόμενο σύμβολο στην συμβολοσειρά εισόδου. Για κάθε έναν από τους κανόνες της γραμματικής, φτιάχνουμε και ένα αντίστοιχο υποπρόγραμμα. Όταν αναγνωριστεί και η τελευταία λέξη του πηγαίου προγράμματος, τότε η συντακτική ανάλυση έχει στεφτεί με επιτυχία.

Ο Συντακτικός Αναλυτής(def yacc()) χρησιμοποιεί επαναλαμβανόμενα τον Λεκτικό Αναλυτή για να παίρνει τα δεδομένα που χρειάζεται. Όλη τη γραμματική την έχουμε υλοποιήσει σε υποπρογράμματα(συναρτήσεις), όπου σε καθε μία από αυτές χρησημοποιούμε τις 2 global μεταβλητές(token0 = αριθμός της λεκτικής μονάδας, token1 = λεκτική μονάδα) οι οποίες είναι τα αποτελέσματα της κλήσης του Λεκτικού Αναλυτή από τον Συντακτικό. Σύμφωνα με τα δεδομένα που δέχεται ο Συντακτικός από τον Λεκτικό Αναλυτή, μπορεί να αποφασίσει ποια κλήση υποπρογράμματος θα ακολουθήσει. Σε περίπτωση που υπάρχει λάθος συντακτικό στο αρχείο Cimple, το πρόγραμμά μας, μέσω του Συντακτικού Αναλυτή, έχει την ικανότητα να υποδείξει που βρίσκεται το λάθος αυτό και να εξηγήσει τον λόγο για τον οποίο είναι λάθος.

Παρακάτω φαίνεται η δομή της γραμματικής που υλοποιεί ο Συντακτικός Αναλυτής (yacc()) και η εξήγηση της λειτουργίας του με παραδείγματα πάνω στον κώδικά μας.

```
# "program" is the starting symbol
program        :    program ID block .

# a block with declarations, subprogram and statements
block          :    declarations subprograms statements

# declaration of variables, zero or more "declare" allowed
declarations   :    ( declare varlist ; )*

# a list of variables following the declaration keyword
varlist        :    ID  ( , ID )*
               |    ε
```

```
# zero or more subprograms allowed
subprograms    :    ( subprogram )*

# a subprogram is a function or a procedure,
# followed by parameters and block
subprogram     :    function ID ( formalparlist ) block
               |    procedure ID ( formalparlist ) block

# list of formal parameters
formalparlist :    formalparitem  ( , formalparitem )*
               |    ε

# a formal parameter ("in": by value, "inout" by reference)
formalparitem :    in ID
               |    inout ID

# one or more statements
statements     :    statement ;
               |    { statement  ( ; statement )* }

# one statement
statement      :    assignStat
               |    ifStat
               |    whileStat
               |    switchcaseStat
               |    forcaseStat
               |    incaseStat
               |    callStat
               |    returnStat
               |    inputStat
               |    printStat
               |    ε

# assignment statement
assignStat     :    ID := expression

# if statement
ifStat         :    if ( condition ) statements elsepart

elsepart       :    else statements
               |    ε


# while statement
whileStat      :    while ( condition ) statements

# switch statement
switchcaseStat:    switchcase
                      ( case ( condition ) statements )*
                      default statements

# forcase statement
forcaseStat    :    forcase
                       ( case ( condition ) statements )*
                       default statements

# incase statement
incaseStat     :    incase
                        ( case ( condition ) statements )*

# return statement
returnStat     :    return( expression )

# call statement
callStat       :    call ID( actualparlist )

# print statement
printStat      :    print( expression )

# input statement
inputStat      :    input( ID )

# list of actual parameters
actualparlist :    actualparitem ( , actualparitem )*
              |    ε

# an actual parameter ("in": by value, "inout" by reference)
actualparitem :    in expression
              |    inout ID

# boolean expression
condition      :    boolterm ( or boolterm )*
```

```
# term in boolean expression
boolterm        :    boolfactor ( and boolfactor )*

# factor in boolean expression
boolfactor      :    not [ condition ]
                |    [ condition ]
                |    expression REL_OP expression

# arithmetic expression
expression      :    optionalSign term ( ADD_OP term )*

# term in arithmetic expression
term            :    factor  ( MUL_OP factor )*

# factor in arithmetic expression
factor          :    INTEGER
                |    ( expression )
                |    ID idtail

# follows a function of procedure (parethnesis and parameters)
idtail          :    ( actualparlist )
                |    ε

# sumbols "+" and "-" (are optional)
optionalSign    :    ADD_OP
                |    ε


# lexer rules: relational, arithentic operations, integers and ids
REL_OP          :    = | <= | >=  | > | < | <>
                ;
ADD_OP          :    + | -

MUL_OP          :    * | /

INTEGER         :    [0-9]+

ID              :    [a-zA-Z][a-zA-Z0-9]*
```

Όλες οι πρασινισμένες λέξεις είναι οι λεκτικές μονάδες(token1) που παίρνουμε από το Λεκτικό Αναλυτή ενώ, οι μαύρισμένες είναι υποπρογράμματα. Επίσης, το ‘ε’ συμβολίζει το ‘κενό’, η κάθετως ‘|’ συμβολίζει το ‘ή’(or) και το αστεράκι ‘*’ συμβολίζει την επανάληψη δηλαδή, μπορούμε να επαναλαμβάνουμε ατελείωτα τη λειτουργία αυτή.

Παραδείγματα υποπρογραμμάτων:

```python
# " program " is the starting symbol
def program():
    global token0
    global token1
    if (token0 == programid):
        token0, token1 = lex()
        if (token0 == wordid):
            name = token1
            token0, token1 = lex()
            block()
            return
        else:
            print("Error: program name expected,in line:", lineCounter)
            sys.exit()
    else:
        print("Error: the keyword 'program' was expected,in line:", lineCounter)
        sys.exit()
    return
```

Το υποπρόγραμμα program καλείται με το που ξεκινήσει η μετάφραση του αρχείου Cimple και καλεστεί ο λεκτικός αναλυτης για πρώτη φορά. Εφόσον διαβαστεί από το Λεκτικό η λέξη program, γίνεται από τον συντακτικό η επόμενη κλήση του Λεκτικού, μέσα από την program, για να διαβαστεί η επόμενη λεκτική μονάδα. Το επόμενο ‘διάβασμα’ γίνεται από την block κ.ο.κ.

```
# a block with declarations , subprogram and statements
def block():
    global token0
    global token1
    declarations()
    subprograms()
    statements()
    return
```

```
# declaration of variables , zero or more " declare " allowed
def declarations():
    global token0
    global token1
    while (token0 == declareid):
        token0, token1 = lex()
        varlist()
        if (token0 == questionamarkid):
            token0, token1 = lex()
            return
        else:
            print("Error: the keyword ';' was expected after the varlist,in line:", lineCounter)
            sys.exit()
    return
```

Από εδώ και πέρα συνεχίζεται το δέντρο καλεσμάτων υποπρογραμμάτων μέχρι να βρεθεί λεκτική μονάδα που υποδεικνύει τέλος προγράμματος ή να βρεθεί λάθος στην συντακτική ανάλυση όπου τερματίζει το πρόγραμμα με το αντίστοιχο μήνυμα.

ΕΝΔΙΑΜΕΣΟΣ ΚΩΔΙΚΑΣ

Ο Ενδιάμεσος Κώδικας είναι ένα σύνολο από τετράδες της μορφής op, x, y, z , οι οποίες αποτελούνται από έναν τελεστή και τρία (3) τελούμενα . Εφαρμόζεται ο τελεστής op στα τελούμενα x και y και το αποτέλεσμα τοποθετείται στο τελούμενο z ,για παράδειγμα : (+, a, b, c) αντιστοιχεί στην πράξη c=a+b.

Ακόμα, για μεταπήδηση χωρίς όρους στη θέση z χρησιμοποιούμε την τετράδα jump, _, _, z . begin_block, name, _, _ , end_block, name, _, _ και name halt , _, _, _ αντίστοιχα ,για αρχή , τέλος υποπρογράμματος και τέλος προγράμματος. Τέλος, για κλήση συνάρτησης χρησιμοποιούμε το call, name, _, _ και για τις παραμέτρους συνάρτησης την εντολή par, x, m, _ όπου x την μεταβλητή και m τον τρόπο μετάδοσης(CV : με τιμή, REF: με αναφορά RET: επιστροφή τιμής συνάρτησης).

Για την πραγματοποίηση του Ενδιάμεσου Κώδικα, κάνουμε χρήση των παρακάτω βοηθητικών συναρτήσεων.

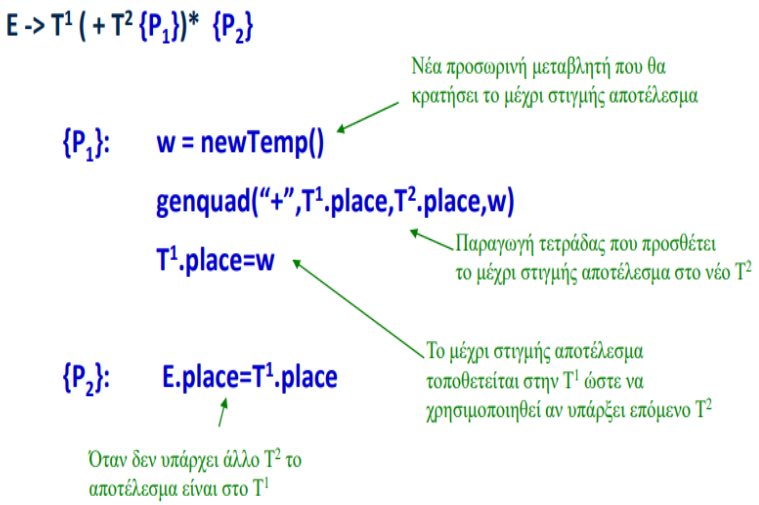
1. nextquad() : επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί.
2. genquad(op, x, y, z) : δημιουργεί την επόμενη τετράδα (op, x, y, z).
3. newtemp() : δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή , οι οποία είναι της μορφής T_1, T_2, T_3 …
4. emptylist() : δημιουργεί μία κενή λίστα ετικετών τετράδων.
5. makelist(x) : δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x.
6. merge(list1 , list2) : δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών list1 , list2.
7. 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. Για την σωστή υλοποίηση της backpatch, έχουμε δημιουργήσει μια global μεταβλητή quadList που περιέχει όλες τις τετράδες που δημιουργούνται. Την quadList την γεμίζουμε εντός της συνάρτησης genquad(). Κάθε τετράδα που δημιουργείται στο πρόγραμμά μας προστίθεται στην λίστα.

Ο Ενδιάμεσος Κώδικας τυπώνεται μέσα σε ένα αρχείο που παράγει ο κώδικάς μας με την κατάληξη .int και σε ένα αρχείο με την κατάληξη .c εάν δεν περιέχει συναρτήσεις(functions and procedures).

ΛΕΙΤΟΥΡΓΙΑ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

```
<program>              ::=   program name
                             <program_block> (name)
<program_block> (name) ::=   <declarations>
                             <subprograms>
                             genquad("begin_block",name,"_","_")
                             < block>
                             genquad("halt","_","_","_")
                             genquad("end_block",name,"_","_")
<subprograms>          ::=   function id <formalpars>
                             {  genquad("begin_block",id,"_","_")
                             <block>
                             genquad("end_block",id,"_","_")
                             }
```

Εικόνα 1

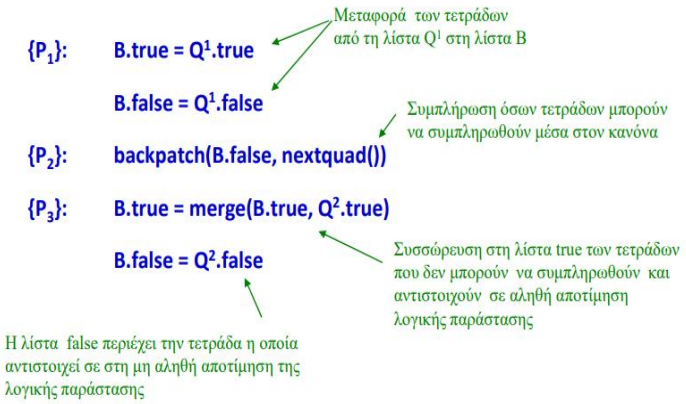


Εικόνα 2

Αρχή, τέλος προγράμματος και αριθμητικές παραστάσεις.

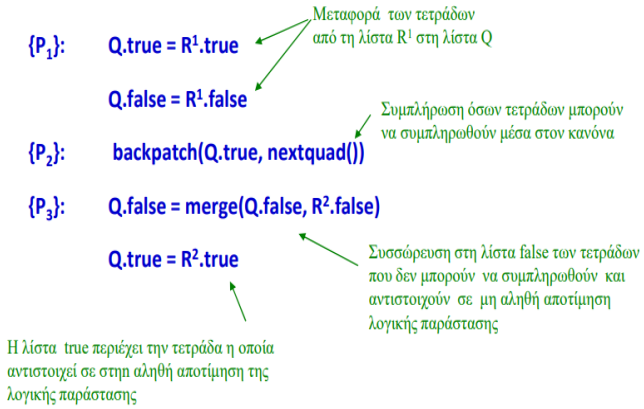
Εικόνα 1η :
Αρχικά ότι βλέπουμε στην φωτογραφία αριστερά ανάμεσα σε <> είναι υλοποιημένη συνάρτηση στο πρόγραμμα μας. Σε περίπτωση που έχουμε υποπρόγραμμα (function or procedure) στο πρόγραμμα, μετά την δήλωση των μεταβλητών, δημιουργούμε τετράδα που υποδεικνύει την αρχή του υποπρογράμματος και ύστερα από την υλοποίηση του εσωτερικού του, δημιουργούμε και την τετράδα του τέλους του. Μόλις τελειώσουν όλα τα υποπρογράμματα, δημιουργούμε την τετράδα της "main" του προγράμματος (name) . Με το που βρεθεί η λεκτική μονάδα(".”= END OF FILE), δημιουργούμε τις 2 τετράδες που σημάνουν τέλος προγράμματος.
Εικόνα 2η :
Σε περίπτωση αριθμητικής παράστασης με περισσότερα από 2 τελούμενα κάνουμε χρήση της newtemp() για δημιουργία προσωρινής μεταβλητής. Έπειτα, δημιουργούμε τετράδα "βοηθητική" με την νέα μεταβλητή, και εν τέλει αν δεν έχουμε άλλες παραστάσεις, το αποτέλεσμα της προσωρινής μεταβλητής και της T1, έχει την τιμή που πρέπει να πάρει το E. Το ίδιο κάνουμε και σε περιπτώσεις (-,*,/) αντί του αθροίσματος.

B -> Q^1 {P_1} (or {P_2} Q^2 {P_3})*



Εικόνα 3

Q -> R^1 {P_1} (and {P_2} R^2 {P_3})*

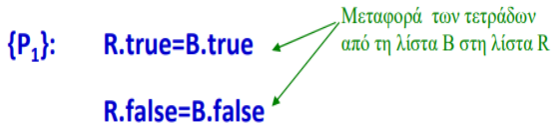


Εικόνα 4

condition() , boolterm()

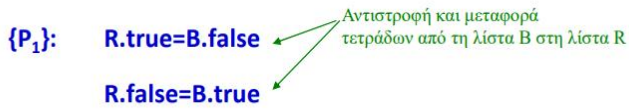
Εικόνα $3^η$,$4^η$: Ο σκοπός της condition() είναι να επιστρέψει 2 λίστες(Btrue, Bfalse), με τα αναγνωριστικά τετράδων για τις περιπτώσεις που είναι αληθής και ψευδής η τετράδα. Σε περίπτωση “or” (συνάρτηση condition()) ή “and”(συνάρτηση boolterm()) οι 2 λίστες κάνουν merge(ενώνουν) τα αναγνωριστικά τετράδων τους αναλόγως εάν είναι αληθής ή ψευδής το condition().

R -> (B) {P_1}



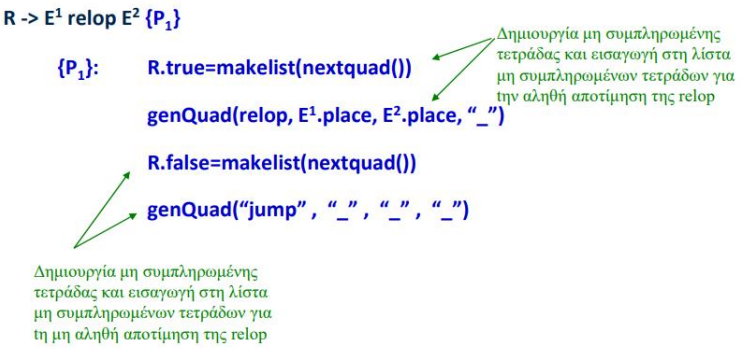
Εικόνα 5

R -> not (B) {P_1}



Εικόνα 6

R -> E^1 relop E^2 {P_1}



Εικόνα 7

boolfactor()

Εικόνα 5η:

Στην πρώτη περίπτωση της boolfactor() επιστρέφουμε τα αποτελέσματα της condition() ως έχουν.

Εικόνα 6η:

Στην δεύτερη περίπτωση της boolfactor(), δηλαδή στην περίπτωση που βρούμε την λεκτική μονάδα "not", αντιστρέφουμε τα αποτελέσματα της condition() λόγω της άρνησης του not, και τα επιστρέφουμε.

Εικόνα 7η:

Στην τρίτη και τελευταία περίπτωση της boolfactor(), δημιουργούμε δυο μη συμπληρωμένες τετράδες για την αληθή και την ψευδή αποτίμηση της relop, δημιουργούμε την τετράδα με τα το σύμβολο του relop και τα δύο τελούμενα. Φτιάχνουμε και την τετράδα όμως του "jump" , καθώς στην ψευδή περίπτωση πρέπει να πηδήξουμε φωλιασμένο κώδικα που δεν θα εκτελεστεί ποτέ. Τέλος επιστρέφουμε τα R.true και R.false, τα οποία είναι λίστες που περιέχουν το αναγνωριστικό της επόμενης τετράδας.

Κλήση συνάρτησης:

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

Εικόνα 8

```
S -> return (E) {P1}

{P1}:   genquad("retv",E.place,"_","_")
```

Εικόνα 9

```
S ->  id := E {P1};

{P1} :   genQuad(":=",E.place,"_",id)
```

Εικόνα 10

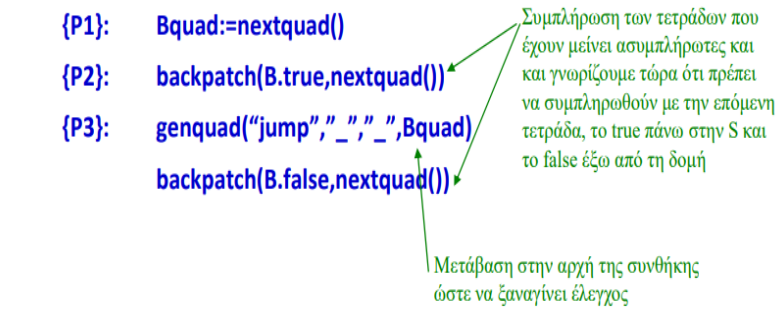
callStat(), returnStat(), assignStat()

Εικόνα 8η :
Σε αυτήν την εικόνα βλέπουμε την λειτουργία της call(κλήση συνάρτησης). Αρχικά φτιάχνουμε τετράδες για τα ορίσματα της συνάρτησης. Στην συνέχεια φτιάχνουμε μια νέα προσωρινή μεταβλητή (και την τετράδα της) και της αναθέτουμε το αποτέλεσμα της κλήσης της συνάρτησης. Εν τέλει φτιάχνουμε τετράδα και για να υποδείξουμε ότι κλήθηκε η συγκεκριμένη συνάρτηση με το όνομα της ως 1ο τελούμενο. Τις δημιουργίες αυτών των τετράδων τις χειριζόμαστε στην callStat() ,στην boolfactor() αλλά και στην actualparitem() που διαχειρίζεται τα ορίσματα.

Εικόνα 9η :
Εδώ, στην περίπτωση που βρούμε την λεκτική μονάδα "return", στην συνάρτηση returnStat() δημιουργούμε την τετράδα με το χαρακτηριστικό "retv" και πρώτο τελούμενο το αποτέλεσμα που παίρνουμε από τη κλήση της expression() για το περιεχόμενο της return.

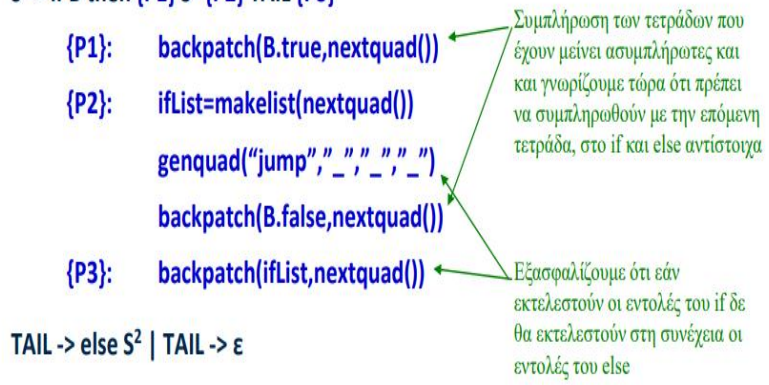
Εικόνα 10η :
Στην περίπτωση της ανάθεσης (:=), στην συνάρτηση assignStat() δημιουργούμε την τετράδα που βλέπουμε σε αυτήν την εικόνα, και σε αυτήν την περίπτωση ως 1ο τελούμενο έχουμε το αποτέλεσμα που παίρνουμε από την κλήση της expression() αλλά και το id(όνομα της μεταβλητής που δέχεται άλλη τιμή).

S -> while {P1} B do {P2} S^1 {P3}



Εικόνα 11

S -> if B then {P1} S^1 {P2} TAIL {P3}



TAIL -> else S^2 | TAIL -> ε

Εικόνα 12

whileStat(), ifStat()

Εικόνα 11η: -whileStat()- Αρχικά, στο P1 κρατάμε την επόμενη τετράδα. Ελέγχουμε τα condition() που υπάρχουν «μέσα στην παρένθεση» . Στο P2 , συμπληρώνουμε , με την χρήση της backpatch(), τις τετράδες που τα αναγνωριστικά τους ταυτίζονται με αυτά που επιστράφηκαν από την χρήση της condition() στις αληθείς περιπτώσεις. Έπειτα στο P3, δημιουργούμε τετράδα "jump", με το αναγνωριστικό που κρατήσαμε στο P1, και τέλος συμπληρώνουμε ξάνα με backpatch() αλλά για τις ψευδείς περιπτώσεις της condition().

Εικόνα 12η: -ifStat()- Αρχικά, στο P1 ελέγχουμε τα condition() που υπάρχουν «μέσα στην παρένθεση». Συμπληρώνουμε , με την χρήση της backpatch(), τις τετράδες που τα αναγνωριστικά τους ταυτίζονται με αυτά που επιστράφηκαν από την χρήση της condition() στις αληθείς περιπτώσεις. Στο P2, συνεχίζουμε με την δημιουργία μιας κενής λίστας και τοποθετούμε εκεί το αναγνωριστικό της επόμενης τετράδας, δημιουργούμε τετράδα για το jump της ψευδής περίπτωσης. Στην συνέχεια, κάνουμε backpatch() για την συμπλήρωση των ψευδών αποτελεσμάτων της condition(). Εν τέλει, στο P3 για την περίπτωση του else, κάνουμε backpatch() για την συμπλήρωση της τετράδας που είχαμε κρατήσει αρχικά στο P1. Αυτό το κάνουμε γιατί στην περίπτωση που δεν εκτελεστούν οι εντολές του if , θα πρέπει να εκτελεστούν οι εντολές του else.

```
S -> switch {P1}
       (  (cond): {P2} S1 break {P3} )*
       default: S2 {P4}

       {P1} :   exitlist = emptylist()
       {P2} :   backpatch(cond.true,nextquad())
       {P3} :   e = makelist(nextquad())
                genquad('jump', '_', '_', '_')
                mergelist(exitlist,e)
                backpatch(cond.false,nextquad())
       {P4} :   backpatch(exitlist,nextquad())
```

Εικόνα 13

```
S -> forcase  {P1}
            ( when (condition) do {P2}
                    sequence {P3}
                    end do ) *
      endforcase

      {P1}:    p1Quad=nextquad()
      {P2}:    backpatch(cond.true,nextquad())
      {P3}:    genquad("jump", "_", "_",p1quad)
               backpatch(cond.false,nextquad())
```

Εικόνα 14

switchcaseStat(), forcaseStat()

Εικόνα 13η:
Αρχικά στο P1 της switchcaseStat(), δημιουργούμε μια κενή τετράδα. Στο P2, συμπληρώνουμε , με την χρήση της backpatch(), τις τετράδες που τα αναγνωριστικά τους ταυτίζονται με αυτά που επιστράφηκαν από την χρήση της condition() στις αληθείς περιπτώσεις. Έπειτα στο P3, φτιάχνουμε μια λίστα κενή και τοποθετούμε την επόμενη τετράδα, δημιουργούμε την τετράδα του jump χωρίς την ένδειξη της τετράδας που πρέπει να γίνει το jump. Ακόμα, ενώνουμε την αρχική κενή λίστα του P1 με την λίστα της επόμενης τετράδας και κάνουμε backpatch() τις ψευδείς περιπτώσεις της condition(). Τέλος στο P4, εκτελούμε την "default" εντολή, συμπληρώνοντας με την backpatch() την αρχική κενή λίστα του P1.

Εικόνα 14η:
Αρχικά στο P1 της forcaseStat(), κρατάμε το αναγνωριστικό της επόμενης τετράδας. Μετά , στο P2, συμπληρώνουμε , με την χρήση της backpatch(), τις τετράδες που τα αναγνωριστικά τους ταυτίζονται με αυτά που επιστράφηκαν από την χρήση της condition() στις αληθείς περιπτώσεις. Εν τέλει στο P3, δημιουργούμε τετράδα "jump" που δείχνει πήδημα στο αναγνωριστικό που κρατήσαμε στο P1 και συμπληρώνουμε με backpatch() τις ψευδές περιπτώσεις της condition().

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

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

Εικόνα 15

```
S ->    incase {P1}
        ( when (condition) do {P2}
                sequence {P3}
                end do ) *
        endincase {P4}
{P1}:   w=newTemp()
        p1Quad=nextquad()
        genquad(":=",1,"_",w)
{P2}:   backpatch(cond.true,nextquad())
         genquad(":=",0,"_",w)
{P3}:   backpatch(cond.false,nextquad())
{P4}:   genquad("=", w,0,p1quad)
```

Εικόνα 16

inputStat(), printStat(), incaseStat()

Εικόνα 15η:
Εδώ, στην περίπτωση που βρούμε την λεκτική μονάδα "input" ή "print", στις συναρτήσεις inputStat() και printStat(), δημιουργούμε την τετράδα με το χαρακτηριστικό "inp" και "out" αντίστοιχα. Το πρώτο τελούμενο στην περίπτωση της input είναι το όνομα της μεταβλητής που εισάγεται στο πρόγραμμα. Το πρώτο τελούμενο στην περίπτωση της print είναι το αποτέλεσμα που παίρνουμε από τη κλήση της expression() για το περιεχόμενο της.

Εικόνα 16η:
Στην περίπτωση της συνάρτησης incaseStat() , αρχικά στο P1, δημιουργούμε νέα προσωρινή μεταβλητή, κρατάμε το αναγνωριστικό της επόμενης τετράδας και φτιάχνουμε τετράδα ανάθεσης της νέας μεταβλητής με 1ο τελούμενο τον αριθμό 1. Έπειτα στην P2, αφού έχουμε μπει στις περιπτώσεις "case", εκτελούμε την condition() για το περιεχόμενο των παρενθέσεων μετά το "case", συμπληρώνουμε με την backpatch() τις αληθείς περιπτώσεις της condition() και δημιουργούμε τετράδα , ξανά με την προσωρινή μεταβλητή που δημιουργήσαμε στο P1, αλλά αντί του 1 τον αριθμό 0. Στην συνέχεια στο P3, συμπληρώνουμε τις ψευδείς περιπτώσεις της condition() με την χρήση της backpatch(). Εν τέλει στο P4, δημιουργούμε την τετράδα με το αναγνωριστικό τετράδας που κρατήσαμε στο P1 την αρχική προσωρινή μεταβλητή και τον αριθμό 0.

ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

Ο Πίνακας Συμβόλων είναι υπεύθυνος για την ανάγνωση των μεταβλητών, των σταθερών και των προσωρινών μεταβλητών μιας συνάρτησης του προγράμματος που δίνεται από τον χρήστη. Για την υλοποίηση του Πίνακα Συμβόλων χρησιμοποιούμε τα παρακάτω:

1. Class Scope(): Κλάση που υποδεικνύει το βάθος φωλιάσματος και κρατάει δεδομένα για όλα τα περιεχόμενα του. Παίρνει παραμέτρους μια λίστα από Entities, το nestingLevel και το framelength.
2. Class Entity(): Κλάση που αντιπροσωπεύει μια μεταβλητή, μια συνάρτηση, μια σταθερά, μια παράμετρο ή μια προσωρινή μεταβλητή. Για κάθε περίπτωση έχουμε και διαφορετικά ορίσματα στην κλάση.
3. Global scopeList: Λίστα με scope, χρήσιμο για την διαχείριση των δεδομένων του scope.
4. Global nestingLevel: Το βάθος φωλιάσματος.
5. Global listofallscopes: Λίστα που περιέχει όλα τα scopes.

Σημαντικές έννοιες:

1. Int framelength: Μήκος εγγραφήματος δραστηριοποίησης.
2. Int startquad: Ετικέτα της πρώτης τετράδας του κώδικα της συνάρτησης.
3. Int offset: Απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης ή απόσταση από την κορυφή της στοίβας.
4. List argument: Λίστα με τις παραμέτρους.
5. Int parmode: Τρόπος περάσματος παραμέτρων.
6. Int startquad: Αναγνωριστική ετικέτα τετράδας.

Ενέργειες Πίνακα Συμβόλων

Προσθήκη νέου Scope: όταν ξεκινάμε τη μετάφραση μιας νέας συνάρτησης

Δείγματα, της εφαρμογής της προσθήκης νέου Scope, στον κώδικα μας:

```
def program():
    global token0
    global token1
    global scopeList
    global nestingLevel
    global listofallscopes
    if (token0 == programid):
        token0, token1 = lex()
        if (token0 == wordid):
            name = token1
            token0, token1 = lex()
            scope1 = Scope([], 0, 12)
            scopeList.append(scope1)
            listofallscopes.insert(0, scopeList[0])
            block(name)
            genquad('halt', '_', '_', '_')
            genquad('end_block', name, '_', '_')
            return
```

```
403     def subprogram():
404         global token0
405         global token1
406         global ccounter
407         global scopeList
408         global nestingLevel
409         global listofallscopes
410         if (token0 == functionid):
411             ccounter=ccounter+1
412             token0, token1 = lex()
413             if (token0 == wordid):
414                 name = token1
415                 func = Entity(functionid, name, 0)
416                 scopeList[-1].addentity(func)
417                 token0, token1 = lex()
418                 if (token0 == leftParenthesisid):
419                     token0, token1 = lex()
420                     nestingLevel = nestingLevel+1
421                     scope = Scope([], nestingLevel, 12)
422                     scopeList.append(scope)
423                     formalparlist()
```

```
elif (token0 == procedureid):
    token0, token1 = lex()
    ccounter=ccounter+1
    if (token0 == wordid):
        name = token1
        proc = Entity(procedureid, name, 0)
        scopeList[-1].addentity(proc)
        token0, token1 = lex()
        if (token0 == leftParenthesisid):
            token0, token1 = lex()
            nestingLevel = nestingLevel + 1
            scope = Scope([], nestingLevel, 12)
            scopeList.append(scope)
            formalparlist()
```

Διαγραφή Scope: όταν τελειώνουμε τη μετάφραση μιας συνάρτησης-με τη διαγραφή διαγράφουμε την εγγραφή(record) του Scope και όλες τις λίστες με τα Entities και τα Arguments(παραμέτρους) που εξαρτώνται από αυτήν.

Δείγματα, της εφαρμογής της διαγραφής Scope, στον κώδικα μας:

Οι δύο αυτές περιπτώσεις βρίσκονται εντός της συνάρτησης subprogram().

```
424                 if (token0 == rightParenthesisid):
425                     token0, token1 = lex()
426                     #genquad('begin_block', name, '_', '_')
427                     block(name)
428                     framelength = scopeList[-1].getframelength()
429                     listofallscopes.append(scopeList[-1])
430                     del scopeList[-1]
431                     nestingLevel = nestingLevel-1
432                     scopeList[-1].changeentityframelength(framelength)
433                     genquad('end_block', name, '_', '_')
```

```
457                 if (token0 == rightParenthesisid):
458                     token0, token1 = lex()
459                     genquad('begin_block', name, '_', '_')
460                     block(name)
461                     framelength = scopeList[-1].getframelength()
462                     listofallscopes.append(scopeList[-1])
463                     del scopeList[-1]
464                     nestingLevel = nestingLevel - 1
465                     scopeList[-1].changeentityframelength(framelength)
466                     genquad('end_block', name, '_', '_')
```

Προσθήκη νέου Entity:

- όταν συναντάμε δήλωση μεταβλητής.
- όταν δημιουργείται νέα προσωρινή μεταβλητή.
- όταν συναντάμε δήλωση νέας συνάρτησης.
- όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης.

Προσθήκη νέου Entity στον κώδικά μας, πραγματοποιείται στις ακόλουθες συναρτήσεις: subprogram(), varlist(), formalparitem(), incaseStat(), expression(), term(), factor().

Προσθήκη νέου Argument: όταν συναντάμε δήλωση τυπικής παραμέτρου συνάρτησης.

Δείγμα, της εφαρμογής της προσθήκης νέου Argument, στον κώδικα μας:

```
    # a formal parameter (" in ": by value , " inout " by reference )
    def formalparitem():
        global token0
        global token1
        global scopeList
        if (token0 == inid):
            token0, token1 = lex()
            if (token0 == wordid):
                name = token1
                inpar = Entity(inid, name, 12)
                inpar.parmode = 'cv'
                scopeList[-1].addentity(inpar)
                wantednum = len(scopeList) -2
                scopeList[wantednum].entitylist[-1].arglist.append('in')
                token0, token1 = lex()
                return
            else:
                print("Error: expected word after 'in' in line:", lineCounter)
                sys.exit()
        elif (token0 == inoutid):
            token0, token1 = lex()
            if (token0 == wordid):
                name = token1
                inoutpar = Entity(inoutid, name, 12)
                inoutpar.parmode = 'ref'
                scopeList[-1].addentity(inoutpar)
                wantednum = len(scopeList) - 2
                scopeList[wantednum].entitylist[-1].arglist.append('inout')
                token0, token1 = lex()
```

Η προσθήκη γίνεται με την εντολή arglist.append(x). x=’in’ ή ‘inout’.

Αναζήτηση Entity: 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 με το ζητούμενο όνομα υπάρχει πάνω από ένα Entity τότε επιστρέφουμε το πρώτο που θα συναντήσουμε.

Δείγμα, της εφαρμογής της αναζήτησης Entity, στον κώδικα μας:

```
def block(namee):
    global token0
    global token1
    global quadList
    global listofallscopes
    flag = False
    declarations()
    subprograms()
    genquad('begin_block', namee, '_', '_')
    for i in range(len(listofallscopes)):
        for y in range(len(listofallscopes[i].entitylist)):
            if listofallscopes[i].entitylist[y].name == namee and (listofallscopes[i].entitylist[y].type == functionid or listofallscopes[i].entitylist[y].type == procedureid):
                listofallscopes[i].entitylist[y].startquad = nextquad()-1
                flag = True
                break
        if flag == True:
            break

    statements()
    return
```

Εδώ επίσης βλέπουμε και τον υπολογισμό της startquad μεταβλητής ενός function ή procedure Entity.

Η χρήση του Boolean flag, γίνεται για να τελειώσει η λούπα, όταν βρεθεί το αντίστοιχο ζητούμενο Entity.

Παρακάτω θα εξηγήσουμε, τον τρόπο σωστού υπολογισμού του offset και του framelength.

Αρχικά, σε κάθε περίπτωση ορισμού νέου Entity, περνάμε σαν παράμετρο το offset 12. Μόνο στον ορισμό Entity function ή procedure το αρχικοποιούμε 0, καθώς δεν το χρειαζόμαστε. Έπειτα, ο υπολογισμός του offset γίνεται αυτόματα όταν προσθέτουμε ένα Entity σε ένα Scope με την εντολή addentity. Ας δούμε πώς.

```
def addentity(self, entity):
counter = 0
    if entity.gettype() != functionid or entity.gettype() != procedureid:
        for i in range(len(self.entitylist)):
            if self.entitylist[i].type==functionid or self.entitylist[i].type== procedureid:
                counter = counter +1
        entity.addoffset(len(self.entitylist)*4)
        entity.addoffset(-(counter*4))
        self.framelength = self.framelength + 4
    for i in range(len(self.entitylist)):
        if self.entitylist[i].name == entity.name and self.entitylist[i].type != entity.type:
            print("Error: Two or more entities have the same name but they are of different types.")
            sys.exit()
        if (self.entitylist[i].name == entity.name and self.entitylist[i].type == entity.type) and (self.entitylist[i].type == procedureid or self.entitylist[i].type == function
            print("Error: Two or more entities have the same name and they are both function/procedure.")
            sys.exit()
self.entitylist.append(entity)
```

Αν το Entity δεν είναι συνάρτηση, υπολογίζουμε τα Entities της λίστας και κρατάμε τον αριθμό στον counter. Μέσω της addoffset(), η οποία προσθέτει τον αριθμό της παραμέτρου στο ήδη υπάρχον offset, προσθέτουμε αρχικά το μήκος της λίστας πολλαπλασιαζόμενο με το 4(“όλα τα offset”) και αφαιρούμε το counter πολλαπλασιαζόμενο με το 4 (“offset που δεν μετράνε”).

Στην Scope, για τον σωστό υπολογισμό του framelength(“sp”), όταν ορίζουμε ένα Scope αρχικοποιούμε το framelength 12. Έπειτα , σε κάθε εκτέλεση της addentity() που δείξαμε παραπάνω, δηλαδή κάθε φορά που προσθέτουμε ένα Entity στο Scope αυτόματα προστίθεται ο αριθμός 4 στο ήδη υπάρχον framelength.

Στην περίπτωση των συναρτήσεων, καταρχάς όταν βρούμε συνάρτηση κάνουμε νέο scope και όταν τελειώσει το διαγράφουμε, το framelength είναι ίσο με το framelength του Scope που μόλις διαγράφηκε. Μπορείτε να δείτε στην φωτογραφία ακριβώς πως το υπολογίζουμε. Κρατάμε το framelength του Scope ακριβώς πριν το διαγράψουμε και με την εφαρμογή της changeentityframelength(), η οποία αλλάζει το framelength του Entity που βρίσκεται στην τελευταία θέση της entityList του Scope και το κάνει ίσο με την παράμετρο, υπολογίζουμε το framelength σωστά.

```
    if (token0 == rightParenthesisid):
        token0, token1 = lex()
        #genquad('begin_block', name, '_', '_')
        block(name)
        framelength = scopeList[-1].getframelength()
        listofallscopes.append(scopeList[-1])
        del scopeList[-1]
        nestingLevel = nestingLevel-1
        scopeList[-1].changeentityframelength(framelength)
        genquad('end_block', name, '_', '_')
```

ΔΙΑΔΙΚΑΣΙΑ ΛΕΙΤΟΥΡΓΙΑΣ ΤΟΥ COMPILER

Η διαδικασία λειτουργίας του Compiler αποτελείται από 5 βασικά στάδια μετάφρασης του προγράμματος από γλώσσα Cimple σε γλώσσα μηχανής Assembly. Το αρχείο test2.ci είναι ένα πρόγραμμα σε γλώσσα Cimple που υλοποιήθηκε από εμάς για τον έλεγχο της ορθής λειτουργίας του Compiler μας. Παρακάτω μπορείτε να δείτε το περιεχόμενο του test2.ci.

```
program test
    # declarations #
    declare a,b,c;

    # function #
    function f(in a)
    {
        tetragwno:=a*a;
        return(tetragwno)
    }
    function mo(in a,in b,in c)
    {
        average := (a+b+c)/3;
        return(average)
    }
    # main #
    {
        input(a);
        input(b);
        input(c);
        forcase
            case(a>10)
            {
                print(a)
            }
            default
            {
                print(b);
                print(c)
            };
        print(mo(in f(in a),in f(in b),in f(in c)))
    }.
```

ΛΕΚΤΙΚΟΣ ΚΑΙ ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Με την εκκίνηση της μετάφρασης, ξεκινάει η διαδικασία της λεκτικής και της συντακτικής ανάλυσης. Πρώτα, το πρόγραμμά μας καλεί τον Συντακτικό Αναλυτή ο οποίος την πρώτη φορά που θα καλεστεί, καλεί τον Λεκτικό Αναλυτή για να διαβαστεί η πρώτη λεκτική μονάδα. Έπειτα γίνεται μια αλληλεπίδραση του Συντακτικού και του Λεκτικού Αναλυτή για την ολοκλήρωση της συντακτικής ανάλυσης του προγράμματος Cimple. Γνωρίζοντας την Γραμματική της Cimple (Σελίδες 4,5,6) , μπορούμε να δούμε τα βήματα που κάνει ο Συντακτικός Αναλυτής μας. Θα ξεκινήσει από την συνάρτηση program(), η οποία καλεί την block(name) με “name” το όνομα του προγράμματος(test). Στην συνέχεια η block(name) με την σειρά της θα κάνει κλήση της declarations() που είναι υπεύθυνη για το όρισμα των μεταβλητών(a,b,c). Επίσης η block(name), μόλις τελειώσει η declarations() , καλεί την subprograms() και στην περίπτωσή μας που υπάρχει function ξανακαλείται η block(f) ξεχωριστά για την function f κ.ο.κ. Αυτή η διαδικασία συνεχίζεται μέχρι ο λεκτικός αναλυτής να διαβάσει την λεκτική μονάδα που υποδεικνύει το τέλος του προγράμματος.

ΕΝΔΙΑΜΕΣΟΣ ΚΩΔΙΚΑΣ

```
0. (begin_block, f, _, _)
1. (*, a, a, T_0)
2. (:=, T_0, _, tetragwno)
3. (retv, tetragwno, _, _)
4. (end_block, f, _, _)
5. (begin_block, mo, _, _)
6. (+, a, b, T_1)
7. (+, T_1, c, T_2)
8. (/, T_2, 3, T_3)
9. (:=, T_3, _, average)
10. (retv, average, _, _)
11. (end_block, mo, _, _)
12. (begin_block, test, _, _)
13. (inp, a, _, _)
14. (inp, b, _, _)
15. (inp, c, _, _)
16. (>, a, 10, 18)
17. (jump, _, _, 20)
18. (out, a, _, _)
19. (jump, _, _, 16)
20. (out, b, _, _)
21. (out, c, _, _)
22. (par, a, CV, _)
23. (par, T_4, RET, _)
24. (call, f, _, _)
25. (par, T_4, CV, _)
26. (par, b, CV, _)
27. (par, T_5, RET, _)
28. (call, f, _, _)
29. (par, T_5, CV, _)
30. (par, c, CV, _)
31. (par, T_6, RET, _)
32. (call, f, _, _)
33. (par, T_6, CV, _)
34. (par, T_7, RET, _)
35. (call, mo, _, _)
36. (out, T_7, _, _)
37. (halt, _, _, _)
38. (end_block, test, _, _)
```

Παραγωγή Ενδιάμεσου Κώδικα

Στο διπλανό σχήμα βλέπουμε το αρχείο που δημιουργείται από την κλήση της συνάρτησης writeToIntFile() για τον έλεγχο της ορθής δημιουργίας των τετράδων(quads). Στο αρχείο αυτό είναι τυπωμένες όλες οι τετράδες που παράχθηκαν από τον Ενδιάμεσο Κώδικα. Το .int αρχείο ξεκινά με την ανάγνωση της function f όπου στην συνάρτηση subprogram() θα κάνει genquad(begin_block,name,_,_) με name το f, πριν μπει στην συνάρτηση statements(). Η statements() διαβάζει wordid, πηγαίνει στην statement() και αυτή με την σειρά της λόγω wordid μπαίνει στην assignStat(). Η assignStat μπαίνει στην expression() και η expression κάνει newtemp() και genquad με την νεα μεταβλητή και τις απαραίτητες πράξεις που χρειάζονται ώστε να πραγματοποιηθεί η ανάθεση με 2 μεταβλητές. Γυρνάμε πίσω στην assignStat() και αυτή κάνει genquad με τις 2 μεταβλητές της ανάθεσης και το σύμβολο της. Γυρνάμε από την statement στην statements και ξαναμπαίνουμε στην statement αυτήν την φορά με λεκτική μονάδα “return”. Κατευθυνόμαστε στην συνάρτηση returnStat() και εκεί δημιουργούμε την τετράδα με το χαρακτηριστικό retv και το όνομα της μεταβλητής που θα επιστραφεί. Η ίδια διαδικασία επαναλαμβάνεται για την επόμενη συνάρτηση (function mo). Αφού τελειώσουν οι συναρτήσεις, και επιστρέψουμε από την subprogram() στην αρχική block(test), γίνεται genquad(begin_block,test,_,_). Προχωράμε στην statements(), διαβάζουμε την λεκτική μονάδα “input” , συνεχίζουμε στην statement() και έπειτα στην inputStat(). Εκεί, δημιουργεί την τετράδα (‘inp’,name,_,_) με name το όνομα της μεταβλητής που βρίσκεται ανάμεσα στις παρενθέσεις. Οι τετράδες στο διπλανό σχήμα με ετικέτα 16-21 αφορούν την forcase. Αφού έχουμε ‘;’ Μετά τις 3 εντολές input, συνεχίζουμε και βρισκόμαστε στην while της statements(). Εκεί θα διαβάσουμε την λεκτική μονάδα της forcase, και θα κατευθυνθούμε στην statement και στην forcaseStat(). Μπαίνοντας σε αυτήν την συνάρτηση, μπαίνουμε στο while λόγω του διαβάσματος του ‘case’ και εκεί καλούμε την condition() για να δούμε αν ισχύει η σύγκριση. Από την condition() πηγαίνουμε στην boolterm() και από αυτήν στην boolfactor() όπου μπαίνουμε στην else της και καλούμε την expression(), την rel_op() και ξανά την expression(). Από αυτές τις κλήσεις κρατάμε τις επιστροφές τους που είναι τα (a,>,10).

```
0. (begin_block, f, _, _)
1. (*, a, a, T_0)
2. (:=, T_0, _, tetragwno)
3. (retv, tetragwno, _, _)
4. (end_block, f, _, _)
5. (begin_block, mo, _, _)
6. (+, a, b, T_1)
7. (+, T_1, c, T_2)
8. (/, T_2, 3, T_3)
9. (:=, T_3, _, average)
10. (retv, average, _, _)
11. (end_block, mo, _, _)
12. (begin_block, test, _, _)
13. (inp, a, _, _)
14. (inp, b, _, _)
15. (inp, c, _, _)
16. (>, a, 10, 18)
17. (jump, _, _, 20)
18. (out, a, _, _)
19. (jump, _, _, 16)
20. (out, b, _, _)
21. (out, c, _, _)
22. (par, a, CV, _)
23. (par, T_4, RET, _)
24. (call, f, _, _)
25. (par, T_4, CV, _)
26. (par, b, CV, _)
27. (par, T_5, RET, _)
28. (call, f, _, _)
29. (par, T_5, CV, _)
30. (par, c, CV, _)
31. (par, T_6, RET, _)
32. (call, f, _, _)
33. (par, T_6, CV, _)
34. (par, T_7, RET, _)
35. (call, mo, _, _)
36. (out, T_7, _, _)
37. (halt, _, _, _)
38. (end_block, test, _, _)
```

Παραγωγή Ενδιάμεσου Κώδικα (Συνέχεια)

Από την condition() πηγαίνουμε στην boolterm() και από αυτήν στην boolfactor() όπου μπαίνουμε στην else της και καλούμε την expression(), την rel_op() και ξανά την expression(). Από αυτές τις κλήσεις κρατάμε τις επιστροφές τους που είναι τα (a,>,10). Δημιουργούμε την τετράδα genquad(>,a,10,_) και την genquad('jump',_,_,_) και επιστρέφουμε στην 2 λίστες την Rtrue και την Rfalse που περιέχουν τα αναγνωριστικά των 2 αυτών τετράδων αντίστοιχα. Αυτές επιστρέφονται στην boolterm(), από εκεί στην condition() και από εκεί πίσω στην forcaseStat(). Διαβάζουμε το κλείσιμο της παρένθεσης, κάνουμε backpatch(Rtrue,nextquad()) για να συμπληρώσουμε τα αναγνωριστικά που θα μεταβούν οι αληθείς συγκρίσεις. Συνεχίζουμε με την κλήση της statements() για το περιεχόμενο των αληθών συγκρίσεων , από την statements() διαβάζοντας το 'print' πηγαίνουμε στην statement() , στην printStat() και εκεί κάνουμε genquad(out,a,_,_). Γυρίζουμε πίσω στην forcaseStat() και κάνουμε backpatch(Rfalse,nextquad()) για να συμπληρώσουμε το αναγνωριστικά ετικετών που θα μεταβούν οι ψευδείς συγκρίσεις. Στην συνέχεια, γυρίζουμε στο while που ελέγχει αν ξαναέχουμε case , βγαίνουμε και μπαίνουμε στην περίπτωση της λεκτικής μονάδας default. Εκεί, ξαναμπαίνουμε στην statements() , στην statement() και δημιουργούνται οι 2 τετράδες για το 'print'. Βγαίνοντας από την forcaseStat() , συνεχίζουμε διαβάζοντας την λεκτική μονάδα print και μπαίνουμε στην printStat(). Σε αυτήν την συνάρτηση, μπαίνουμε στην expression() και από την expression() μπαίνοντας στην term() και ύστερα στην factor() και στην idtail(). Εκεί , δημιουργεί νέα μεταβλητή και δημιουργεί τις τετράδες (par,a,CV,_) , (par,T_4,RET,_) και (call,f,_,_). Ξαναεκτελεί την expression(), γίνεται το ίδιο και το ίδιο έως ότου φτάσει στο σημείο να έχει μια μεταβλητή, στην περίπτωσή μας T_7 και μπορεί να την κάνει print. Έτσι δημιουργεί την τετράδα (out,T_7,_,_) εντός της printStat(). Εν τέλει, γυρίζει πίσω στο αρχίκο program, εκεί όπου δημιουργεί τις 2 τελικές τετράδες (halt,_,_,_) και (end_block,test,_,_). Εκεί, τελειώνει το πρόγραμμα.

Η διαδικασία του ενδιάμεσου κώδικα που ακολουθήσαμε παραπάνω για την forcase είναι ίδια για κάθε άλλη περίπτωση που ελέγχει κάποιο condition.(ifStat, whileStat, switchcaseStat, incaseStat)

ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

Σχετικά με το κομμάτι του Πίνακα Συμβόλων που αναλύσαμε(Σελίδες 16-21) , παρακάτω φαίνεται, με την κλήση της συνάρτησης writeToSaFile(), ένα αρχείο scopes.txt με τις πληροφορίες για όλα τα Scopes και τα Entities.

```
Scope0
Entity: 0 Name: a, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 12
Entity: 1 Name: b, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 16
Entity: 2 Name: c, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 20
Entity: 3 Name: f, Type: 5, StartQuad: 0, parMode: , Framelength: 20, Offset: 12
Arglist: ['in']
Entity: 4 Name: mo, Type: 5, StartQuad: 5, parMode: , Framelength: 36, Offset: 12
Arglist: ['in', 'in', 'in']
Entity: 5 Name: T_4, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 24
Entity: 6 Name: T_5, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 28
Entity: 7 Name: T_6, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 32
Entity: 8 Name: T_7, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 36

Scope1
Entity: 0 Name: a, Type: 20, StartQuad: -1, parMode: cv, Framelength: 0, Offset: 12
Entity: 1 Name: T_0, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 16

Scope2
Entity: 0 Name: a, Type: 20, StartQuad: -1, parMode: cv, Framelength: 0, Offset: 12
Entity: 1 Name: b, Type: 20, StartQuad: -1, parMode: cv, Framelength: 0, Offset: 16
Entity: 2 Name: c, Type: 20, StartQuad: -1, parMode: cv, Framelength: 0, Offset: 20
Entity: 3 Name: T_1, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 24
Entity: 4 Name: T_2, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 28
Entity: 5 Name: T_3, Type: 0, StartQuad: -1, parMode: , Framelength: 0, Offset: 32
```

ΕΠΙΠΡΟΣΘΕΤΟ TEST ΑΡΧΕΙΟ

Ακόμη, εάν στον κώδικα .ci δεν υπάρχει function ή procedure , δημιουργούμε ένα αρχείο quadlist.c μέσω της συνάρτησης writeToCFile(), που περιέχει τον κώδικα μεταφρασμένο στην γλώσσα C. Ακολουθεί ένα παράδειγμα.

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

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

```
#include <stdio.h>

int main()
{
int: x,sum,average,T_0,T_1,T_2;
L_0: //(begin_block, average, _, _)
L_1:scanf("%d",&x); //(inp, x, _, _)
L_2:sum=0; //(:=, 0, _, sum)
L_3:count=0; //(:=, 0, _, count)
L_4:if(x > 1) goto L_6; //(>, x, 1, 6)
L_5:goto L_11; //(jump, _, _, 11)
L_6:T_0=sum+x; //(+, sum, x, T_0)
L_7:sum=T_0; //(:=, T_0, _, sum)
L_8:T_1=count+1; //(+, count, 1, T_1)
L_9:count=T_1; //(:=, T_1, _, count)
L_10:goto L_4; //(jump, _, _, 4)
L_11:T_2=sum/count; //(/, sum, count, T_2)
L_12:average=T_2; //(:=, T_2, _, average)
L_13:printf("%d",average); //(out, average, _, _)
L_14:{}
L_15: //(end_block, average, _, _)
}
```

ΚΩΔΙΚΑΣ ΤΟΥ COMPILER

Μπορείτε να βρείτε τον κώδικα του προγράμματος μας, κλικάροντας τον σύνδεσμο .