# Εργαστηριακή Άσκηση:
# Γλώσσα προγραμματισμού cpy

ΖΑΧΡΑ - ΕΛΙΣΑΒΕΤ ΑΛΛΑ ΓΚΑΜΠΟΥ (5052)

ΜΑΡΙΑ ΚΑΤΩΛΗ (5083)

Μεταφραστές

Καθηγητής:

ΓΕΩΡΓΙΟΣ ΜΑΝΗΣ


Εαρινό Εξάμηνο 2023-24

## Φάση 1η: Λεκτική Ανάλυση

Ο λεκτικός αναλυτής καλείται ως συνάρτηση από τον συντακτικό αναλυτή,διαβάζει γράμμα – γράμμα το αρχικό πρόγραμμα και κάθε φορά που καλείται επιστρέφει την επόμενη λεκτική μονάδα ( Token).

Επιστρέφει στον συντακτικό αναλυτή έναν ακέραιο που χαρακτηρίζει την λεκτική μονάδα καθώς και την ίδια την λεκτική μονάδα.

**Το αλφάβητο** της γλώσσας CPY για την οποία καλούμαστε να υλοποιήσουμε τον μεταφραστή , 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 («(»,«)», «#{»,«#}») • και σχολίων («##»).

**Οι δεσμευμένες λέξεις** είναι οι εξής :

- main
- #int
- If
- while
- print
- return
- input
- and
- def
- global
- elif
- int
- or
- #def
- else
- not

Οι λέξεις αυτές δεν μπορούν να χρησιμοποιηθούν ως μεταβλητές.

Παρακάτω φαίνεται το definition των λεκτικών μονάδων (Tokens) στην αρχή του κώδικα μας:

:

```
##ZACHRA ELISAVET ALLA GKAMPOU, 5052
##MARIA KATOLI, 5083

import os
import sys #gia na mporesoume na diavasoume to file apo to command line

#TOKENS
DIGITS='0123456789'
CAP_LETTERS='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LOW_LETTERS='abcdefghijklmnopqrstuvwxyz'
TT_PLUS =100
TT_MINUS =101
TT_MUL=102
TT_DIV=103
TT_MOD=104
TT_LESS_THAN=105
TT_MORE_THAN=106
TT_EQUALS=107
TT_LESS_EQUAL=108
TT_MORE_EQUAL=109
TT_NOT_EQUAL=110
TT_ASSIGN=111
TT_COMMA=112
TT_COLON=113
TT_LPAREN=114
TT_RPAREN=115
TT_LBRACKET=116
TT_RBRACKET=117
TT_COMMENT=118
TT_MAIN=119
INT_DECLARE=120 #einai to #int
TT_IF=121
TT_WHILE=122
TT_PRINT=123
TT_RETURN=124
TT_INPUT=125
TT_AND=126
TT_DEF=127
TT_GLOBAL=128
TT_ELIF=129
TT_INT=130
TT_OR=131
TT_DEFINE=132 #einai to #def
TT_ELSE=133
TT_NOT=134
TT_ERROR=135
TT_EOF=136
```

```
E1=-1 #ANAGNWRISTIKO/DESMEUMEMH LEKSH
E2=-2 #AKERAIA STATHERA
TT_GROUP=139
TT_ID=140
```

**<u>Η κλάση Token:</u>**

*(Manis Handbook chapter 4 'Λεκτικός Αναλυτής' , σελ. 41)*

```
class Token:
    def __init__(self, recognized_string, family, line_number):
        self.recognized_string = recognized_string
        self.family = family
        self.line_number = line_number

    def __str__(self):
        if self.recognized_string:
            return f'{self.recognized_string} family:"{self.family}", line: {self.line_number}'
        return f'family:"{self.family}", line: {self.line_number}'
```

Η κλάση Token χρησιμοποιείται για την δημιουργία αντικειμένων που αναπαριστούν λεκτικές μονάδες. Ένα αντικείμενο της κλάσης Token περιέχει την πληροφορία που μεταφέρει ο λεκτικός αναλυτής στον συντακτικό αναλυτή

Τα properties της κλάσεις είναι :

- recognized string : Η συμβολοσειρά που αναγνωρίστηκε.
- family : Η οικογένεια στην οποία ανήκει το token
- line number : η γραμμή του κώδικα στην οποία βρέθηκε το token.

Η κλάση Token αποτελείται από τον constructor '__init__' καθώς και από την μέθοδο '__str__' που είναι υπεύθυνη για την μορφή εκτύπωσης ενός αντικειμένου της κλάσης Token.

Η δεύτερη κλάση στο διάγραμμα κλάσεων είναι αυτή που δημιουργεί το αντικείμενο του λεκτικού αναλυτή ,**η κλάση Lex.**

```
class Lex:
    def __init__(self, token ,file_name):
        self.token=token
        self.file_name=file_name
        self.current_char=None
        self.current_line=1
```

*(Manis Handbook chapter 4 'Λεκτικός Αναλυτής' ,σελ 48).*

Τα πεδία της κλάσης Lex είναι τα εξής :

- token : αποτελεί τη συσχέτιση με την κλάση Token,περιέχει τη λεκτική μονάδα που επιστρέφεται ως αποτέλεσμα στον συντακτικό αναλυτή
- current_line: περιέχει τη γραμμή στην οποία βρισκόμαστε.
- file_name: αποθηκεύει το όνομα του αρχείου όπως θα το δώσει ο προγραμματιστής στη γραμμή εντολών.
- current_char: χρησιμοποιείται για να αποθηκεύει τον τρέχον χαρακτήρα που διαβάζεται από το αρχείο

## Μέθοδος lex()

Η κύρια λειτουργικότητα του λεκτικού αναλυτή υλοποιείται στην μέθοδο **lex()** ,η οποία είναι υπεύθυνη για την ανάγνωση χαρακτήρων από το αρχείο και την αναγνώριση τους σύμφωνα με τον πίνακα καταστάσεων pinakas, ο οποίος αποτελεί ένα αυτόματο.

```
    def lex(self):
        recognized_string=''
        family=''
        in_comment=False
        pinakas=[[0,1,2,TT_PLUS,TT_MINUS,TT_MUL,3,TT_MOD,TT_LPAREN,TT_ERROR,TT_RBRACKET,TT_ERROR,TT_COMMA,T
                [E2,1,TT_ERROR,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2,E2],
                [E1,2,2,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1,E1],
                [TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_DIV,TT_ERROR,TT_ERROR,TT_ERROR,TT_
                [TT_ERROR,TT_ERROR,2,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_LBRA(
                [TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,T
                [TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,
                [TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,
                [TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,T
```

*Συνέχεια σκρινσοτ(οριζόντια) :*

```
,TT_COLON,4,5,6,7,8,TT_EOF,TT_ERROR],
```

```
TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR],
RACKET,TT_RBRACKET,TT_ERROR,TT_ERROR,TT_COMMENT,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR],
N,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_EQUALS,TT_ASSIGN,TT_ASSIGN,TT_ASSIGN,TT_AS
AN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN,
AN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN,
R,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_NOT_EQUAL,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR,TT_ERROR]]
```

```
,
TT_ASASIGN,TT_ASSIGN],
THAN,TT_LESS_EQUAL,TT_LESS_THAN,TT_LESS_THAN,TT_LESS_THAN],
THAN,TT_MORE_EQUAL,TT_MORE_THAN,TT_MORE_THAN,TT_MORE_THAN],
OR]]
```

Το in_comment θα χρησιμοποιηθεί για την διαχείριση των σχολίων ,η μεταβλητή recognized_string χρησιμοποιείται για την αποθήκευση των χαρακτήρων που διαβάζονται από το αρχείο ενώ η μεταβλητή family χρησιμοποιείται για την κατηγοριοποίηση κάθε recognized_string σε μια συγκεκριμένη οικογένεια (κατηγορία).

**Pinakas:**

Κάθε γραμμή του πίνακα pinakas αντιπροσωπεύει μια κατάσταση(state) και κάθε στήλη αντιπροσωπεύει ένα είδος εισόδου(putIn). Η τιμή που βρίσκεται στη διασταύρωση μιας γραμμής και μιας στήλης δείχνει την επόμενη κατάσταση στην οποία θα μεταβεί το αυτόματο όταν βρίσκεται στην τρέχουσα κατάσταση και διαβάσει συγκεκριμένο σύμβολο εισόδου.

```
putIn=0
state=0
while state>=0 and state<100:
    self.current_char=f1.read(1)
    if not self.current_char:
        f1.close()
        break
    if self.current_char=='#' and not in_comment:
        next_char=f1.read(1)
        if next_char=='#':
            in_comment=True
            recognized_string=''
            continue
        else:
            f1.seek(f1.tell()-1)
    if in_comment:
        if self.current_char=='#':
            next_char=f1.read(1)
            if next_char=='#':
                in_comment=False
            continue
        else:
            continue
    if self.current_char in ' \t' :
        putIn=0
        family='whitespace'
    elif self.current_char=='\n':
        putIn=0
        family=''
    elif self.current_char in DIGITS:
        putIn=1
    elif self.current_char in CAP_LETTERS:
        putIn=2
    elif self.current_char in LOW_LETTERS:
        putIn=2
    elif self.current_char=='+':
        putIn=3
        family='addOperator'
    elif self.current_char=='-':
        putIn=4
        family='addOperator'
    elif self.current_char=='*':
        putIn=5
        family='mulOperator'
```

```
elif self.current_char=='(':
    putIn=8
    family='groupSymbol'
elif self.current_char==')':
    putIn=9
    family='groupSymbol'
elif self.current_char=='{':
    putIn=10
    family='groupSymbol'
elif self.current_char=='}':
    putIn=11
    family='groupSymbol'
elif self.current_char==',':
    putIn=12
    family='delimiters'
elif self.current_char==':':
    putIn=13
    family='delimiters'
elif self.current_char=='#':
    putIn=14
elif self.current_char=='=':
    putIn=15
elif self.current_char=='<':
    putIn=16
elif self.current_char=='>':
    putIn=17
elif self.current_char=='!':
    putIn=18
elif not self.current_char:
    putIn=19
    family='Error:EOF'
else:
    putIn=20
    family='Error:Invalid Char'
recognized_string+=self.current_char

state=pinakas[state][putIn]
```

Το state αρχικοποιείται στο 0 καθώς αντιπροσωπεύει την αρχική κατάσταση του αυτομάτου ενώ το putIn αρχικοποιείται στο 0 γιατί δεν εχει αναγνωσθεί ακόμη κάποιος χαρακτήρας. Εντός του βρόχου while η κατάσταση state αλλάζει με βάση τους χαρακτήρες(putIn) που διαβάζονται και του πίνακα pinakas.Ο βρόχος συνεχίζεται όσο το state>=0 && state<100 καθώς ο πίνακας έχει ορισμένες τελικές καταστάσεις με αριθμούς μεγαλύτερους του 100,όπως η οριστικοποίηση των Tokens ,και μικρότερους του 0 ( E1,E2).Αν διαβαστούν αριθμοί ,γράμματα, σύμβολα ,λευκοί χαρακτήρες κλπ. το state αλλάζει βάση του pinaka.Αν έχουμε φτάσει στο τέλος του αρχείου επιστρέφεται το σφάλμα(EOF) ενώ αν διαβαστεί μη έγκυρος χαρακτήρας επιστρέφεται το ανάλογο σφάλμα Invalid Char.

Προσθέτουμε το current char στο recognized_string και τελικά κρατάμε μόνο τους 30 πρώτους χαρακτήρες καθώς ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα.

```
recognized_string=recognized_string[0:30] ##only take into consideration the first 30 letters
```

Στον κώδικα μας η συνάρτηση next_token() είναι η lex().Επίσης η διαχείριση των error γίνεται μέσα στην ίδια την lex().

<u>Διαχείριση Σχολίων</u>

Η διαχείριση σχολίων πραγματοποιείται με την βοήθεια της μεταβλητής in_comment.

Αν ο τρέχον χαρακτήρας είναι #(υπενθυμίζεται ότι στην γλώσσα μας τα σχόλια βρίσκονται ανάμεσα σε ##..##) και δεν βρισκόμαστε μέσα σε σχόλιο διαβάζουμε τον επόμενο χαρακτήρα και αν είναι # , υποδεικνύεται σχόλιο. Θέτουμε την in_comment σε True , ‘αδειάζουμε’ το

recognized string (recongnized_string='')καθώς δεν θέλουμε να αποθηκεύουμε το περιεχόμενο των σχολίων και πηγαίνουμε στον επόμενο χαρακτήρα(continue) , ενώ αν ο επόμενος χαρακτήρας δεν είναι # ,επιστρέφουμε στην θέση που βρισκόμασταν προηγουμένως και συνεχίζουμε την ανάγνωση. Αν βρισκόμαστε ήδη σε σχόλιο ,ελέγχουμε πάλι για ## , και αν το βρούμε σημαίνει ότι βρισκόμαστε στο τέλος του σχολίου και αρά in_comment=False.

Αφού ολοκληρωθεί το σχόλιο, το πρόγραμμα πηδάει στην επόμενη επανάληψη του βρόχου while, χωρίς να εκτελέσει τον υπόλοιπο κώδικα σε αυτήν την επανάληψη. Αν ο τρέχοντας χαρακτήρας δεν είναι #, τότε συνεχίζουμε την ανάγνωση του επόμενου χαρακτήρα.

```
if (state == TT_LESS_THAN or state == TT_MORE_THAN or state == E1 or state == E2 or state == TT_ASSIGN):
    recognized_string=recognized_string[:-1]
    current_position=f1.tell()
    if (current_position>1):
        f1.seek(current_position-1)
        current_position=f1.tell()
```

Όταν φτάνουμε στις καταστάσεις  TT_LESS_THAN, TT_MORE_THAN, E1 ή E2 χρειάζεται να αφαιρέσουμε τον τελευταίο χαρακτήρα από το recognized_string.Αυτό συμβαίνει καθώς για αν φτάσουμε σε αυτές τις τελικές καταστάσεις έχουμε διαβάσει έναν χαρακτήρα που ανήκει σε επόμενη λεκτική μονάδα(*Manis Handbook chapter 4,σελ.43)* .Συνεπώς, αφαιρούμε αυτόν τον χαρακτήρα  για να έχουμε την σωστή αναπαράσταση και έπειτα η θέση ανάγνωσης στο αρχείο επιστρέφεται στην προηγούμενη θέση, ώστε να διαβάσουμε τον χαρακτήρα που ακολουθεί τον τελευταίο αφαιρεθέντα. Αυτό γίνεται με τη χρήση της f1.seek(current_position - 1). Στη συνέχεια, η θέση ανάγνωσης αποθηκεύεται στη μεταβλητή current_position .

```
for c in recognized_string:
    if c=='\n':
        self.current_line+=1

recognized_string=recognized_string.strip()
```

Το σημείο αυτό του κώδικα είναι υπεύθυνο για τον υπολογισμού του line number.Το recognized_string γίνεται strip() καθώς αυτό εξασφαλίζει ότι οι αλλαγές γραμμής που πιθανόν να περιέχονται στο τέλος του recognized_string δεν θα προκαλέσουν λανθασμένη μέτρηση των γραμμών.

```
if state>=100:
    if (state==TT_COMMENT):
        family='comment'
    if (state==TT_ASSIGN):
        family='assignment'
    if (state==TT_LESS_THAN or state==TT_MORE_THAN or state==TT_EQUALS or state==TT_LESS_EQUAL or state==TT
        family='relOperator'
    if state==TT_ERROR:
        family='Error'
elif state==E1:
    if recognized_string in ('main','if','while','print','return','input','and','#def','global','elif','#ir
        family='keyword'
    else:
        family='identifier'
elif state==E2:
    if int(recognized_string)<-32767 or int(recognized_string)>32767:
        family='Error: invalid number'
    else:
        family='number'


return Token(recognized_string,family,self.current_line)
```

Συνέχεια σκρινσοτ οριζόντια :

```
e==TT_MORE_EQUAL or state==TT_NOT_EQUAL


,'#int','or','else','not','def','int'):
```

Το τελευταίο μέρος του κώδικα είναι υπεύθυνο για την ανάθεση ‘οικογένειας’ σε διάφορες λεκτικές μονάδες .

Αν το state=E1( αναγνωριστικό ή δεσμευμένη λέξη) γίνεται ο ανάλογος διαχωρισμός με έλεγχο αν το recognized_string είναι κάποια δεσμευμένη λέξη.

Αν το state=E2(ακέραια σταθερά) γίνεται έλεγχος για το αν ο recognized_string είναι εντός του επιτρεπτού φάσματος τιμών . Αν ναι, αναγνωρίζεται ως number αλλιώς ως σφάλμα μη έγκυρου αριθμού.

## Συντακτική Ανάλυση

Κατά τη συντακτική ανάλυση ελέγχεται εάν η ακολουθία των λεκτικών μονάδων που σχηματίζεται από τον λεκτικό αναλυτή, αποτελεί μία νόμιμη ακολουθία ,με βάση τη γραμματική της γλώσσας .Όποια ακολουθία δεν αναγνωρίζεται από τη γραμματική, αποτελεί μη νόμιμο κώδικα και οδηγεί στον εντοπισμό συντακτικού σφάλματος. Έτσι, ο συντακτικός αναλυτής παίρνει ως είσοδο μία ακολουθία από λεκτικές μονάδες.

```
class Syntax:
    def __init__(self, lexer, generated_program, symbolTable):
        self.lexer=Lex(token,file_name)
        self.generated_program=generated_program
        self.symbolTable=symbolTable
        self.current_token=None
```

Η κλάση Syntax αναλαμβάνει την υλοποίηση του συντακτικού αναλυτή. Αρχικοποιείται με έναν αντικείμενο Lex που είναι υπεύθυνο για τη δημιουργία των tokens από το αρχείο εισόδου. Αποτελείται από τον constructor __init__()  και τα properties είναι τα εξης :

- lexer: αντικείμενο της κλάσης lex
- *generated_program : Είναι μέρος επόμενης φάσης*
- *SymbolTable  : είναι μέρος επόμενης φάσης*

```
def get_token(self):
    self.current_token=self.lexer.lex()
    return self.current_token

def syntax_analyzer(self):
    global token
    token=self.get_token()
    self.program()
    print("No errors found; successful compilation")

def error(self,message):
    global token
    print(message + " on line: " + str(token.line_number))
```

Η μέθοδος get_token() διαβάζει την επόμενη λεκτική μονάδα μέσω της μεθόδου lex() του λεκτικού αναλυτή,την αποθηκεύει στο πεδίο current_token του συντακτικού αναλυτή και το επιστρέφει.

Η μέθοδος syntax_analyzer() είναι η μέθοδος που ξεκινά τη διαδικασία ανάλυσης . Αρχίζει παίρνοντας το πρώτο token και καλώντας τη βασική συντακτική δομή program().

Η μέθοδος error() κάνει print το message ,οπού σε κάθε περίπτωση αποτελεί ένα διαφορετικό μήνυμα λάθος , ανάλογα με το εκάστοτε error και την γραμμή όπου συναντήθηκε .

Στην συνέχεια αυτό που πρέπει να υλοποιήσουμε είναι η γραμματική που μας έχει δοθεί σε κώδικα Python.

Συνεπώς ας δούμε θεωρητικά ποια είναι η λογική πίσω από την λογική αυτή(από γραμματική σε κώδικα).

Υλοποίηση συναρτήσεων με αναδρομική κατάβαση:

Ας αναλύσουμε τη μεθοδολογία της τεχνικής αυτής σε βήματα. Βασισμένοι πάντα στους κανόνες γραμματικής που μας έχουν δοθεί:

1. Δίνουμε στη συνάρτηση το ίδιο όνομα με τον αντίστοιχο κανόνα.
2. Γράφουμε το κώδικα που φαίνεται στο δεξί μέλος του κανόνα:
    a. Όταν καλείται μια διαφορετική γραμματική δομή, καλούμε τη συνάρτηση που την υλοποιεί.
    b. Όταν χρειάζεται να ελέγξουμε αν ακολουθεί μια λεκτική μονάδα, χρησιμοποιούμε την μέθοδο get_token() του συντακτικού αναλυτή και ελέγχουμε αν ισούται με το current_token που επιστρέφεται.

**Την πρακτική που χρησιμοποιήσαμε για τις υλοποιήσεις των κανόνων που χρησιμοποιούν Kleene star θα εξηγήσουμε παρακάτω με χρήση παραδειγμάτων του κώδικά μας.*

Ακολουθούν στιγμιότυπα τόσο της δοθείσας γραμματικής όσο και της αντίστοιχης υλοποίησης μας σε κώδικα Python.

```
program         :   declarations functions main
                ;
```

```
def program(self):
    self.declarations()
    self.functions()
    self.main()
```

Όπως φαίνεται ,καλούνται στην συνάρτηση program() οι συναρτήσεις declarations(),functions(),main() που έχουμε υλοποιήσει για τις αντίστοιχες γραμματικές.

```
declarations    :   ( '#int' id_list )*
                ;

id_list         :   ID ( ',' ID )*
                ;
```

```
def declarations(self):
    global token
    while token.recognized_string=='#int':
        token=self.get_token()
        self.id_list()

def id_list(self):
    global token
    if token.family=='identifier':
        token=self.get_token()
        while token.recognized_string==',':
            token=self.get_token()
            if token.family=='identifier':
                token=self.get_token()
            else:
                self.error("Error: no ID found after ','")
```

Οι παραπάνω κανόνες παρατηρούμε ότι χρησιμοποιούν το Kleene star, οπότε θα εξηγήσουμε τη λογική της υλοποίησής τους σε κώδικα. Ο πρώτος κανόνας έχει μία μικρή διαφορά από τον δεύτερο, αλλά δεν υπάρχουν μεγάλες διαφορές.

Ο κανόνας declarations τοποθετεί το #int μέσα στην παρένθεση του Kleene star, ενώ ο κανόνας id_list τοποθετεί το ID έξω από αυτή την παρένθεση. Άρα, όταν φτιάξαμε την συνάρτηση declarations() ελέγχαμε εξ' αρχής αν το token ισούται με τη λέξη κλειδί "#int" μέσα σε ένα while loop, διότι είναι αναγκαίο πριν την αρχικοποίηση μιας ακολουθίας μεταβλητών να βρίσκεται μπροστά. Αντίθετα, όταν φτιάξαμε την συνάρτηση id_list() ελέγχουμε αρχικά άμα υπάρχει λεκτική μονάδα τύπου "identifier" και αν υπάρχει κόμμα μετά από αυτή τότε αρχίζει το while loop. Ωστόσο, δεν είναι αναγκαίο να μπει στο loop.

Ομοίως με την declarations() υλοποιείται η glob_decl().

Ομοίως με την id_list() υλοποιούνται οι παρακάτω:

- expression()
- term()

- actual_par_list()
- condition()
- bool_term()

Επιπλέον, ας προσθέσουμε και την υλοποίηση της functions(), η οποία χρησιμοποιεί τον κανόνα function σε Kleene star:

```
functions       :   function*
                ;
```

```
def functions():
    global token
    while token.recognized_string=='def':
        self.function()
```

Εδώ ελέγχεται αν η λεκτική μονάδα ισούται με το “def” γιατί αποτελεί την πρώτη λεκτική μονάδα που εντοπίζεται όσο εκτελείται η συνάρτηση του κανόνα function.

Θα δείξουμε τι εννοούμε με αυτό παραθέτοντας τον κώδικα της function() με μια σύντομη επεξήγηση της διαδικασίας που ακολουθεί.

```
function        :   'def' ID '(' id_list ')' ':'
                    '#{'
                        declarations
                        functions
                        glob_decl
                        code_block
                    '#}'
                ;
```

```
def function():
    global token
    if token.recognized_string=='def':
        token=self.get_token()
        if token.family=='identifier':
            token=self.get_token()
            if token.recognized_string=='(':
                token=self.get_token()
                self.id_list()
                if token.recognized_string==')':
                    token=self.get_token()
                    if token.recognized_string==':':
                        token=self.get_token()
                        if token.recognized_string=='#{':
                            self.declarations()
                            self.functions()
                            self.glob_decl()
                            self.code_block()
                            if self.recognized_string=='#}':
                                token=self.get_token()
                            else:
                                self.error("Error: opened the function but never closed it, missing '#}'")
                        else:
                            self.error("Error: never opened function, missing '#{'")
                    else:
                        self.error("Error: empty function after ':'")
                else:
                    self.error("Error: never closed ID_LIST, missing ')'")
            else:
                self.error("Error: parentheses missing from function declaration")
        else:
            self.error("Error: missing function identifier")
```

Στην συνάρτηση function() υλοποιείται ο κώδικας που αντιστοιχεί σε έναν συνδυασμό των δύο πρακτικών που αναλύσαμε στη θεωρία της αναδρομικής κατάβασης. Ξεκινάει πρώτα με έλεγχο ισότητας της τρέχουσας λεκτικής μονάδας με τη λέξη κλειδί 'def' (όπως αναφέρθηκε και παραπάνω). Αν η συνθήκη ισχύει, καταναλώνουμε την επόμενη λεκτική μονάδα και ακολουθείται αυτή η λογική μέχρι να βρεθεί η παρένθεση (*token.recognized_string==')'*) όπου εκεί καταναλώνεται η επόμενη λεκτική μονάδα. Στη συνέχεια πρέπει να κληθεί η id_list() (που αποτελεί διαφορετική γραμματική δομή), η οποία ελέγχει άμα υπάρχουν αναγνωριστικά ως παράμετροι στην δήλωση της συνάρτησης. Τη στιγμή που θα τελειώσει με την συνάρτηση id_list(), θα ελέγξει αν οι επόμενες λεκτικές μονάδες τηρούν τη σειρά που δίνεται στη γραμματική μας, ώσπου θα φτάσει να ελέγξει το μπλοκ κώδικα της συνάρτησης, το οποίο αποτελείται από τις συναρτήσεις declarations(), functions(), glob_decl() και code_block(). Τα errors που βρίσκονται στο else κομμάτι των διαφορετικών ifs είναι τοποθετημένα στα σημεία που θεωρήσαμε εμείς φρόνιμο να εκτυπώνει σφάλμα ο μεταφραστής μας.

## Φάση 2η: Ενδιάμεσος κώδικας

Εφόσον έχουμε τελειώσει την ανάπτυξη του συντακτικού αναλυτή και έχουμε βεβαιωθεί ότι όλα λειτουργούν όπως πρέπει, είμαστε έτοιμοι να προχωρήσουμε στο επόμενο κομμάτι της υλοποίησης του μεταφραστή μας, τον ενδιάμεσο κώδικα.

Ο ενδιάμεσος κώδικας είναι μία ενδιάμεση αναπαράσταση της αρχικής γλώσσας πριν φτάσει στην τελική της μορφή.

Ένα πρόγραμμα συμβολισμένο στην ενδιάμεση γλώσσα αποτελείται από μία σειρά από τετράδες, οι οποίες είναι αριθμημένες έτσι ώστε σε κάθε τετράδα να μπορούμε να αναφερθούμε χρησιμοποιώντας τον αριθμό της ως ετικέτα.

*(Manis Handbook Chapter 6 'Ενδιάμεσος κώδικας', σελ. 63)*

Για να επιτύχουμε την αναπαράσταση του κώδικά μας σε τετράδες, έπρεπε αρχικά να υλοποιήσουμε ορισμένες βοηθητικές συναρτήσεις οι οποίες αναφέρονται στο εγχειρίδιο *(Manis Handbook Chapter 6 'Ενδιάμεσος κώδικας', σελ. 68)*. Παρακάτω παραθέτουμε τις κλάσεις **Quad**, **QuadList**, **QuadPointer** και **QuadPointerList** που φτιάξαμε για να μας βοηθήσουν στην υλοποίηση των συναρτήσεων αυτών:

**Quad:**

```
class Quad:
    def __init__(self, label, op, op1, op2, op3):
        self.label=QuadPointer(label)
        self.op=op
        self.op1=op1
        self.op2=op2
        self.op3=op3

    def __str__(self):
        return f'{self.label.label}: "{self.op},{self.op1},{self.op2},{self.op3}"'
```

Η χρήση της κλάσης αυτής είναι να κρατάει σε ένα αντικείμενο μια τετράδα με τα πεδία:

- **label:** κρατάει τον αριθμό της ετικέτας της τετράδας
- **op:** κρατάει την λειτουργία που πραγματοποιεί αυτή η τετράδα
- **op1, op2, op3:** τους τρεις operators της τετράδας

Φτιάξαμε και μία συνάρτηση η οποία τυπώνει τα περιεχόμενα της τετράδας με την επιθυμητή μορφοποίηση.

**QuadList:**

```
class QuadList:
    def __init__(self, programList, quadPointers, quad_counter):
        self.programList=programList
        self.quadPointers=QuadPointerList([])
        self.quad_counter=quad_counter

    def __str__(self):
        return f'QuadList(Quads: {self.programList}\nNumber of quads: {self.quad_counter}\n")'

    def backpatch(self, labelList, quadLabel):
        for l in labelList:
            for q in self.programList:
                if l==q.label:
                    q.op3=quadLabel

    def genQuad(self, operator, operand1, operand2, operand3):
        global quadLabel
        newQuadPointer=QuadPointer(quadLabel)
        newQuad=Quad(newQuadPointer,operator,operand1,operand2,operand3)
        self.programList.append(newQuad)
        self.quadPointers.append(newQuadPointer)
        self.quad_counter+=1
        quadLabel+=1

    def deleteLastQuad(self):
        return self.programList.pop(-1)

    def makeList(self, label):
        return [label]

    def emptyList(self):
        return []

    def nextQuad(self):
        return quadLabel

    def printQuads(self, filename):
        with open(filename, 'w') as f:
            for q in self.programList:
                f.write(str(q) + '\n')
```

Η **QuadList** είναι η κλάση που υλοποιεί τις περισσότερες από τις βοηθητικές συναρτήσεις που προαναφέραμε. Ένα αντικείμενο τύπου **QuadList** έχει τρία πεδία:

- **programList:** μια λίστα που κρατάει όλες τις τετράδες που δημιουργήθηκαν
- **quadPointers:** ουσιαστικά είναι μια λίστα που κρατάει τις ετικέτες των τετράδων που δημιουργήθηκαν. Πρακτικά είναι ένα αντικείμενο QuadPointerList, που θα εξηγήσουμε παρακάτω
- **quad_counter:** κρατάει τον συνολικό αριθμό των τετράδων

Στη συνέχεια, θα εξηγήσουμε τι κάνει η κάθε μία από τις βοηθητικές συναρτήσεις που υλοποιούνται σε αυτή την κλάση:

- **backpatch():** η συνάρτηση αυτή παίρνει ως παραμέτρους μια λίστα με ετικέτες **labelList** και μια ετικέτα **quadLabel** και ψάχνει μέσα στις ήδη υπάρχοντες τετράδες ποιες από αυτές έχουν τις ετικέτες που είναι μέσα στη labelList και αντικαθιστά τον τρίτο operator τους με την quadLabel.
- **genQuad():** η συνάρτηση αυτή παίρνει ως παραμέτρους το **operation** και τους **operators**, φτιάχνει ένα νέο αντικείμενο QuadPointer -ουσιαστικά μια ετικέτα- με την global μεταβλητή quadLabel και στη συνέχεια δημιουργεί μια νέα τετράδα με αυτή την ετικέτα. Τέλος, την προσθέτει στη λίστα με τις τετράδες του QuadList, προσθέτει

την ετικέτα της στην quadPointers του QuadList και αυξάνει τον quad_counter και το quadLabel.

- **makeList():** παίρνει ως παράμετρο την ετικέτα μιας τετράδας και την επιστρέφει ως μοναδικό στοιχείο μέσα σε μία λίστα.
- **emptyList():** αδειάζει μία λίστα.
- **nextQuad():** επιστρέφει την ετικέτα της επόμενης τετράδας, η οποία ουσιαστικά είναι η τρέχουσα, εφόσον αυξάνεται στη **genQuad().**
- **printQuads():** τυπώνει τις τετράδες σε ένα αρχείο (συγκεκριμένα το “quads.sym”).

Προσθέσαμε και μία ακόμη συνάρτηση την **deleteLastQuad()**, η οποία αφαιρεί και επιστρέφει την τελευταία τετράδα από την λίστα με τις υπάρχουσες τετράδες. Θα εξηγήσουμε αργότερα γιατί χρειάστηκε αυτή η συνάρτηση.

Υλοποιείται μια μέθοδος **__str__()** που τυπώνει τις τετράδες και τον συνολικό αριθμό τους.

**QuadPointer:**

```
class QuadPointer:
    def __init__(self, label):
        self.label=label

    def __str__(self):
        return f'{self.label}'
```

Η κλάση **QuadPointer** είναι απλά ένα αντικείμενο που κρατάει την ετικέτα μιας τετράδας.

Υλοποιείται μια μέθοδος **__str__()** που τυπώνει την ετικέτα.

**QuadPointerList:**

```
class QuadPointerList:
    def __init__(self, labelList):
        self.labelList=labelList

    def __str__(self):
        return f'QuadPointerList(List of the labels: {self.labelList})'

    def mergeList(self, l1, l2):
        l=list(set(l1+l2))
        return l

    def append(self, label):
        self.labelList.append(label)
```

Η κλάση QuadPointerList είναι ένα αντικείμενο που κρατάει μια λίστα από ετικέτες τετράδων. Επίσης, υλοποιεί την βοηθητική συνάρτηση **mergeList()**, διότι η υλοποίησή της χειρίζεται λίστες από ετικέτες. Αυτό που κάνει είναι να ενώνει δύο λίστες και να επιστρέφει την λίστα-ένωσή τους.

Προσθέσαμε μία επιπλέον συνάρτηση, την **append()**, η οποία προσθέτει μια ετικέτα στην labelList της QuadPointerList. Αυτό έγινε γιατί μας χρειάστηκε στην **genQuad() της QuadList** που παρουσιάστηκε παραπάνω.

Υλοποιείται η μέθοδος **__str__()** που τυπώνει τη λίστα με τις ετικέτες.

Ένα τελευταίο βήμα πριν περάσουμε στην ενσωμάτωση των συναρτήσεων του ενδιάμεσου κώδικα στον συντακτικό αναλυτή είναι να προσθέσουμε ένα ακόμη πεδίο στον constructor του Syntax μας:

```
class Syntax:
    def __init__(self, lexer, generated_program, symbolTable):
        self.lexer=Lex(token,file_name)
        self.generated_program=generated_program
        self.symbolTable=symbolTable
        self.current_token=None
```

Το **generated_program** είναι η λίστα που θα κρατάει τις τετράδες που θα δημιουργούνται κατά τη διάρκεια της συντακτικής ανάλυσης.

Είδαμε και αναλύσαμε, λοιπόν, όλες τις κλάσεις και τις βοηθητικές μεθόδους που έχουν υλοποιηθεί, επομένως τώρα λείπει να τις καλέσουμε στις συναρτήσεις του συντακτικού αναλυτή μας εκεί που χρειάζεται. Ας αρχίσουμε με τις πιο απαιτητικές συναρτήσεις που χρειάζονται αλλαγές και αυτές είναι όσες εμπλέκονται με την ικανοποίηση συνθηκών. Από την γραμματική παρατηρούμε ότι τα **if/while statements** χρησιμοποιούν την **condition()**.

```
if_stat         :   'if' condition ':'
                        statement_or_block
                    ('elif' condition':'
                        statement_or_block )*
                    (   'else' ':'
                        statement_or_block )?
                ;

while_stat      :   'while' condition ':'
                        statement_or_block
                        ;
```

Η **condition()** χρησιμοποιεί την **bool_term()**, όπου η **bool_term()** χρησιμοποιεί την **bool_factor()**.

```
condition       :   bool_term ( 'or' bool_term )*
                ;

bool_term       :   bool_factor ( 'and' bool_factor )*
                ;

bool_factor     :   expression rel_op expression
                |   'not' expression rel_op expression
                ;
```

Εν τέλει, καταλήγουμε στην **expression()** που χρησιμοποιεί η **bool_factor()**, οπότε αρχικά ας δούμε αυτή:

```
def expression(self):
    expr_place=''
    term1_place=''
    term2_place=''
    temp=''
    addOp_string=''
    self.optional_sign()
    term1_place=self.term()
    while token.family=="addOperator":
        addOp_string=self.add_oper()
        term2_place=self.term()
        temp=self.newTemp()
        self.generated_program.genQuad(addOp_string,term1_place,term2_place,temp)
        term1_place=temp
    expr_place=term1_place
    return expr_place
```

Μέσα στην **expression()** έχουμε φτιάξει 5 επιπλέον μεταβλητές:

- **expr_place**: η μεταβλητή που θα κρατήσει την τελική τιμή επιστροφής
- **term1_place, term2_place**: οι μεταβλητές που θα αποθηκεύσουν το αποτέλεσμα που επιστρέφει τη **term()** (θα το δούμε στη συνέχεια)
- **temp**: η μεταβλητή που κρατάει την προσωρινή μεταβλητή που ενδεχομένως θα δημιουργηθεί
- **addOpString()**: μεταβλητή που κρατάει το operator που προκύπτει από την κλήση της **add_oper()**

Ας δούμε αναλυτικά τη διαδικασία που ακολουθείται μέσα στη συνάρτηση:

1. Αρχικά, ελέγχουμε για πιθανό επιπλέον πρόσημο (δεν είναι απαραίτητο να υπάρχει)
2. Στη συνέχεια, καλούμε την **term()** η οποία ελέγχει αν υπάρχει πολλαπλασιασμός μεταξύ κάποιων όρων και στην συνέχεια επιστρέφει τον όρο (**t_place**) που προέκυψε.

   Ας δούμε την **term()** πιο αναλυτικά.

```
def term(self):
    t_place=''
    fact1_place=''
    fact2_place=''
    temp=''
    mulOp_string=''
    fact1_place=self.factor()
    while token.family=="mulOperator":
        mulOp_string=self.mul_oper()
        fact2_place=self.factor()
        temp=self.newTemp()
        self.generated_program.genQuad(mulOp_string,fact1_place,fact2_place,temp)
        fact1_place=temp
    t_place=fact1_place
    return t_place

def factor(self):
    global token
    f_place=''
    expr_place=''
    if token.family=='number':
        f_place=token.recognized_string
        token=self.get_token()
    elif token.recognized_string=='(':
        token=self.get_token()
        expr_place=self.expression()
        f_place=expr_place
        if token.recognized_string==')':
            token=self.get_token()
        else:
            self.error("Error: FACTOR EXPRESSION never closes, missing ')'")
    elif token.family=='identifier':
        ID=token.recognized_string
        f_place=token.recognized_string
        token=self.get_token()
        self.idtail(ID)
    return f_place
```

Η **term()** καλεί εσωτερικά την **factor()** η οποία εν τέλει επιστρέφει ως **f_place** το **recognized_string** το οποίο διάβασε, είτε είναι αριθμός είτε κάποιο expression είτε αναγνωριστικό.

Η **term()** αποθηκεύει το **f_place** που επέστρεψε η **factor()** σε μία μεταβλητή **fact1_place** και στη συνέχεια ελέγχει αν υπάρχει επιπλέον όροι που πολλαπλασιάζονται μεταξύ τους. Αν ισχύει κάτι τέτοιο, αποθηκεύει το **mulOp_string** το οποίο είναι ή '*' ή '//' ή '%' και στη συνέχεια αποθηκεύει τον επόμενο όρο της πράξης στην **fact2_place**. Επειδή σε αυτό το σημείο ξέρουμε σίγουρα ότι υπάρχει πράξη ανάμεσα σε δύο αριθμούς καταλαβαίνουμε ότι πρέπει να δημιουργήσουμε μια προσωρινή μεταβλητή που φαινομενικά θα κρατήσει την πράξη αυτών των δύο όρων. Αυτό, βέβαια, γίνεται μόνο για λόγους δημιουργίας της τετράδας. Έτσι, φτιάχνουμε μια προσωρινή μεταβλητή και φτιάχνουμε μια νέα τετράδα με τους όρους που βρήκαμε προηγουμένως, την προσωρινή μεταβλητή που τους "αποθηκεύει" και τη **mulOp_string** που αποθηκεύει το σύμβολο της πράξης. Ύστερα, η **t_place** παίρνει την τιμή της προσωρινής μεταβλητής. Αν η συνθήκη του while loop δεν ακολουθήθηκε ποτέ, τότε η **t_place** παίρνει την τιμή του πρώτου και **μοναδικού όρου fact1_place**.

3. Μετά, αν βρεθεί η επόμενη λεκτική μονάδα να είναι ή '+' ή '-', δηλαδή να υπάρχει προσθαφαίρεση κάποιων όρων, ακολουθείται η διαδικασία που περιγράφηκε για την **term()**.
4. Στο τέλος επιστρέφεται η τελική τιμή του **expr_place**.

Έχουμε εξηγήσει μέχρι στιγμής τι γίνεται με τις πράξεις μεταξύ όρων και πως οι όροι αυτοί αποθηκεύονται και επιστρέφονται και πως δημιουργούνται οι τετράδες κατά τη διάρκεια. Όμως, η διαδικασία των conditions δεν τελειώνει εκεί, αλλά συνεχίζεται στις συναρτήσεις **bool_term()** και **bool_factor()**. Αυτές οι δύο συναρτήσεις είναι υπεύθυνες για τον συντακτικό έλεγχο των συνθηκών μέσα στα if/while statements και γι' αυτόν τον λόγο θα εξηγήσουμε τη λογική που ακολουθείται για την υλοποίηση των τετράδων μέσα σε αυτές.

Επειδή ενδεχομένως να εμπλακούν πολλές συνθήκες στις οποίες θα θελήσουμε να κάνουμε λογικές πράξεις μέσα σε μία γλώσσα (πχ. *"while isPrime(2) and (leap(2023) or leap(2012))"*) και το αν ισχύουν τελικά ή όχι μπορεί να μας οδηγήσει σε διαφορετικά μονοπάτια, καλούμαστε να τις διαχειριστούμε κρατώντας τις ετικέτες των τετράδων που δείχνουν που πρέπει να πάμε μετά τον έλεγχο των συνθηκών. Γι' αυτό φτιάχνουμε μια κλάση Cond όπως φαίνεται παρακάτω:

```
class cond: ##na krataei tis listes twn labels gia true/false otan elegxoume ta conditions sta while kai if
    def __init__(self, true, false):
        self.true=true
        self.false=false

    def __str__(self):
        return f'{self.true},{self.false}'
```

Τα **true** και **false** πεδία υποδεικνύουν τις λίστες μέσα στις οποίες θα αποθηκεύσουμε τις ετικέτες των τετράδων που θα οδηγήσουν οι συνθήκες που θα ελέγχουμε.

Συνεχίζουμε παραθέτοντας τον κώδικα της **bool_factor()**:

```
def bool_factor(self):
    global token
    c=cond([],[])
    expr1_place=""
    expr2_place=""
    relOp_string=""
    if token.recognized_string=='not':
        token=self.get_token()
        expr1_place=self.expression()
        relOp_string=self.rel_oper()
        expr2_place=self.expression()
        c.false=self.generated_program.makeList(self.generated_program.nextQuad())
        self.generated_program.genQuad(relOp_string, expr1_place, expr2_place, '_')
        c.true=self.generated_program.makeList(self.generated_program.nextQuad())
        self.generated_program.genQuad('jump','_','_','_')
    else:
        expr1_place=self.expression()
        relOp_string=self.rel_oper()
        expr2_place=self.expression()
        c.true=self.generated_program.makeList(self.generated_program.nextQuad())
        self.generated_program.genQuad(relOp_string, expr1_place, expr2_place, '_')
        c.false=self.generated_program.makeList(self.generated_program.nextQuad())
        self.generated_program.genQuad('jump','_','_','_')
    return c
```

Ξεκινάμε από αυτή τη συνάρτηση γιατί είναι η τελευταία που καλείται μεταξύ των **condition()**, **bool_term()** και της ίδιας.

Η συνάρτηση ξεκινάει με την αρχικοποίηση ενός αντικειμένου Cond, δύο μεταβλητών **expr1_place**, **expr2_place** για την τιμή επιστροφής της **expression()** και μία μεταβλητή relOp_string για την πράξη σύγκρισης που γίνεται ανάμεσα στα expressions. Αν η τρέχουσα λεκτική μονάδα είναι το “not”, τότε το **c.false** παίρνει την τιμή της ετικέτας που δείχνει στην τετράδα ***relOp_string,expr1_place,expr2_place,_***, διότι αν η σύγκριση των expression δεν ισχύει (πχ. not 2<1) τότε η άρνηση της σύγκρισης ισχύει, επομένως ισχύει και η συνθήκη. Αν όμως η σύγκριση των expression ισχύει (πχ not 2>1), τότε η άρνησή της σύγκρισης δεν ισχύει, επομένως δεν ισχύει ούτε η συνθήκη, άρα πρέπει να βγούμε από το statement. Επομένως, η ετικέτα που θα κρατήσει το **c.true** είναι αυτή της τετράδας ***jump,_,_,_***. Τελικά επιστρέφεται το αντικείμενο Cond.

Το επιστρεφόμενο αντικείμενο της **bool_factor()** θα φανεί χρήσιμο στην **bool_term()**, η οποία ακολουθεί παρόμοια διαδικασία με την **term()**, απλά για λογικές συνθήκες. Ας τη δούμε αναλυτικά:

```
def bool_term(self):
    global token
    c=cond([],[])
    c1=cond([],[])
    c2=cond([],[])
    c1=self.bool_factor()
    c.true=c1.true
    c.false=c1.false
    while token.recognized_string=='and':
        token=self.get_token()
        self.generated_program.backpatch(c.true,self.generated_program.nextQuad())
        c2=self.bool_factor()
        c.true=c1.true
        c.false=self.generated_program.quadPointers.mergeList(c.false,c2.false)
    return c
```

Εδώ αρχικοποιούμε 3 αντικείμενα κλάσης Cond, το πρώτο το οποίο και θα επιστραφεί από τη συνάρτηση και τα άλλα δύο που είναι βοηθητικά. Κρατάμε στο c1 το αποτέλεσμα της **bool_factor()** και περνάμε στα **c.true** και **c.false** τα αντίστοιχα **c1.true** και **c1.false**. Αν η επόμενη λεκτική μονάδα που βρεθεί μετά το **bool_factor()** είναι “and” τότε περνάμε σε ένα while loop στο οποίο χρησιμοποιούμε τις **backpatch()** και **mergeList()** για να επιτύχουμε αυτό που θέλουμε. Αρχικά καλούμε την **backpatch()** με παραμέτρους το **c.true** που βρήκαμε πριν και την επόμενη ετικέτα, ώστε να βρει την τετράδα με την ετικέτα που έχει κρατήσει το

**c.true** και βάλει ως 3ο operator την ετικέτα της τρέχουσας τετράδας. Μετά, αποθηκεύει στο c2 τα true και fault που προκύπτουν από το **bool_factor()** που καλείται και κρατάει στο **c.true** το c1.true ενώ ύστερα προσθέτει την ετικέτα που έχει το **c2.false** στην **c.false**. Τελικά επιστρέφει το c.

```
def condition(self):
    global token
    c=cond([],[])
    c1=cond([],[])
    c2=cond([],[])
    c1=self.bool_term()
    c.true=c1.true
    c.false=c2.false
    while token.recognized_string=='or':
        token=self.get_token()
        self.generated_program.backpatch(c.false,self.generated_program.nextQuad())
        c2=self.bool_term()
        c.false=c2.false
        c.true=self.generated_program.quadPointers.mergeList(c.true,c2.true)
    return c
```

Η ίδια διαδικασία γίνεται και στην **condition()** με τη διαφορά ότι η **backpatch()** αποθηκεύει την επόμενη ετικέτα στην τετράδα που δείχνει η **c.false**. Η **c.false** παίρνει τις τιμές της **c2.false**, γιατί είναι η μόνη που περιέχει τις τετράδες που ο κώδικας θα μεταβεί αν δεν ισχύει η συνθήκη. Στη **c.true** προστίθεται η **c2.true** και αυτό γίνεται διότι η λίστα **c2.true** περιέχει τετράδες οι οποίες θα κάνουν λογικό άλμα στο ίδιο σημείο που θα κάνουν άλμα και οι τετράδες της **c.true**.

Οι παραπάνω μέθοδοι βοήθησαν στην εύκολη και αποτελεσματική υλοποίηση των if και while statements όπως φαίνεται παρακάτω:

```
def if_stat(self):
    global token
    c=cond([],[])
    l=[]
    if token.recognized_string=='if':
        token=self.get_token()
        c=self.condition()
        if token.recognized_string==':':
            token=self.get_token()
            self.generated_program.backpatch(c.true,self.generated_program.nextQuad())
            self.statement_or_block()
            l=self.generated_program.makeList(self.generated_program.nextQuad())
            self.generated_program.genQuad('jump','_','_','_')
            self.generated_program.backpatch(c.false,self.generated_program.nextQuad())
            while token.recognized_string=='elif':
                token=self.get_token()
                c=self.condition()
                if token.recognized_string==':':
                    token=self.get_token()
                    self.generated_program.backpatch(c.true,self.generated_program.nextQuad())
                    self.statement_or_block()
                    l=self.generated_program.makeList(self.generated_program.nextQuad())
                    self.generated_program.genQuad('jump','_','_','_')
                    self.generated_program.backpatch(c.false,self.generated_program.nextQuad())
                else:
                    self.error("Error: ELIF missing ':'")
            if token.recognized_string=='else':
                token=self.get_token()
                if token.recognized_string==':':
                    token=self.get_token()
                    self.statement_or_block()
                else:
                    self.error("Error: ELSE missing ':'")
        else:
            self.error("Error: IF missing ':'")
```

```
def while_stat(self):
    global token
    c=cond([],[])
    quad=0
    temp=''
    if token.recognized_string=='while':
        token=self.get_token()
        quad=self.generated_program.nextQuad()
        c=self.condition()
        if token.recognized_string==':':
            token=self.get_token()
            self.generated_program.backpatch(c.true,self.generated_program.nextQuad())
            self.statement_or_block()
            temp=str(quad)
            self.generated_program.genQuad('jump','_','_',temp)
            self.generated_program.backpatch(c.false,self.generated_program.nextQuad())
        else:
            self.error("Error: WHILE missing ':'")
```

Κάποιες επιπλέον μικρές προσθέσεις τετράδων για τις απλές λειτουργίες του κώδικα όπως **begin_block**, **end_block**, **halt**, **in**, **ret**, **out**, **= (assignment)**:

```
def program(self):
    self.generated_program.genQuad('begin_block','program','_','_')
    scope=self.symbolTable.create_scope("program")
    actRec=ActivationRecord('main')
    func=Function('main',self.generated_program.nextQuad(),actRec)
    self.declarations("localVar", func)
    self.functions()
    self.main(func)
    self.symbolTable.print_table(scope)
    self.symbolTable.delete_scope()
    self.generated_program.genQuad('begin_block','program','_','_')
```

```
def main(self, func):
    global token
    if token.recognized_string=='#def':
        token=self.get_token()
        if token.recognized_string=='main':
            self.generated_program.genQuad("begin_block","main","_","_")
            token=self.get_token()
            self.declarations("localVar",func)
            self.code_block()
            self.generated_program.genQuad("halt","_","_","_")
            self.generated_program.genQuad("end_block","main","_","_")
```

```
def print_stat(self):
    global token
    if token.recognized_string=='print':
        self.generated_program.genQuad("out","_","_","_")
        token=self.get_token()
        if token.recognized_string=='(':
            token=self.get_token()
            self.expression()
            if token.recognized_string==')':
                token=self.get_token()
            else:
                self.error("Error: PRINT function never closes, missing ')'")
        else:
            self.error("Error: PRINT function missing parentheses")

def return_stat(self):
    global token
    expr_place=''
    if token.recognized_string=='return':
        token=self.get_token()
        self.generated_program.genQuad("ret",token.recognized_string,"_","_")
        expr_place=self.expression()
```

```
def assignment_stat(self):
    global token
    expr_place=''
    ID=''
    if token.family=='identifier':
        ID=token.recognized_string
        token=self.get_token()
        if token.recognized_string=='=':
            token=self.get_token()
            if token.recognized_string=='int':
                token=self.get_token()
                if token.recognized_string=='(':
                    token=self.get_token()
                    if token.recognized_string=='input':
                        token=self.get_token()
                        self.generated_program.genQuad("in","_","_","_")
                        if token.recognized_string=='(':
                            token=self.get_token()
                            if token.recognized_string==')':
                                token=self.get_token()
                                if token.recognized_string==')':
                                    token=self.get_token()
                                else:
                                    self.error("Error: never closed int(input()), missing ')'")

                            else:
                                self.error("Error: never closed input, missing ')'")
                        else:
                            self.error("Error: INPUT function missing parentheses")
                else:
                    self.error("Error: INT missing INPUT function")
            else:
                expr_place=self.expression()
                self.generated_program.genQuad('=',expr_place,'_',ID)
        else:
            self.error("Error: ASSIGNMENT missing")
```

Παρακάτω θα προσθέσουμε άλλο ένα στιγμιότυπο από τον κώδικα που προσθέσαμε στην function() στη φάση της υλοποίησης του ενδιάμεσου κώδικα:

```
def function(self):
    global token
    global tmpVarList
    nestedFunctions=[]
    if token.recognized_string=='def':
        token=self.get_token()
        if token.family=='identifier':
            ID=token.recognized_string
            actRec=ActivationRecord(ID)
            func=Function(ID,self.generated_program.nextQuad(),actRec)
            self.symbolTable.scopes[-1].insert_entity(func)
            self.generated_program.genQuad("begin_block", ID,"_","_")
            scope=self.symbolTable.create_scope(ID)
            token=self.get_token()
            if token.recognized_string=='(':
                token=self.get_token()
                self.id_list("parameter", "formalPar", func)
                if token.recognized_string==')':
                    token=self.get_token()
                    if token.recognized_string==':':
                        token=self.get_token()
                        if token.recognized_string=='#{':
                            token=self.get_token()
                            self.declarations("localVar",func)
                            self.nested_functions()
                            self.glob_decl("globalVar",func)
                            while token.family=='identifier' or token.family=='keyword':
                                self.code_block()
                            if token.recognized_string=='#}':
                                self.generated_program.genQuad("end_block",ID,"_","_")
                                func.activationRecord.tmpVars=tmpVarList
                                for v in func.activationRecord.tmpVars:
                                    func.activationRecord.framelength+=4
                                tmpVarList=[]
                                self.symbolTable.print_table(scope)
                                self.symbolTable.delete_scope()
                                token=self.get_token()
                            else:
                                self.error("Error: opened the function but never closed it, missing '#}'")
```

Εδώ φαίνεται ο κύριος κώδικας της function() (λίγο πριν τις εκτυπώσεις των error δηλαδή). Επειδή σημειώθηκε (*Manis Handbook Chapter 6 'Ενδιάμεσος Κώδικας'' , σελ. 90-91*) ότι τα begin_block και end_block δεν πρέπει να είναι εμφωλευμένα, δηλαδή να εμφανίζονται με τη

σειρά που εμφανίστηκε το τμήμα των εκτελέσιμων εντολών τους, χρειάστηκε να δημιουργήσουμε μια επιπρόσθετη συνάρτηση που θα διαχειρίζεται τις εμφωλευμένες δηλώσεις συναρτήσεων.

**nested_functions():**

```
def nested_functions(self):
    global token
    global quadLabel
    while token.recognized_string=='def':
        q=Quad(0,"","","","")
        q=self.generated_program.deleteLastQuad()
        quadLabel=quadLabel-1
        self.functions()
        self.generated_program.genQuad(q.op,q.op1,q.op2,q.op3)
```

Αυτό που κάνει ουσιαστικά αυτή η μέθοδος είναι να διαγράφει και να αποθηκεύσει την προηγούμενη τετράδα που δημιουργήθηκε από την λίστα τετράδων του προγράμματος **generated_program** με τη μέθοδο **deleteLastQuad()** της QuadList που αναφέρθηκε παραπάνω σε ένα αντικείμενο κλάσης Quad, να μειώνει το συνολικό αριθμό ετικετών **quadLabel** κατά 1 και μετά να εκτελεί όλα τα nested functions μέσω της **functions()**. Αφού τελειώσει αυτή τη διαδικασία, φτιάχνει ξανά την τετράδα που διαγράψαμε και συνεχίζει με την εκτέλεση της εξωτερικής συνάρτησης.

## Φάση 3η: Πίνακας Συμβόλων

Στον πίνακα συμβόλων αποθηκεύουμε , κατά τη φάση της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου κώδικα, πληροφορίες σχετικά με τα συμβολικά ονόματα που χρησιμοποιούμε σε ένα πρόγραμμα: μεταβλητές, συναρτήσεις,παράμετροι, κλπ.Η πληροφορία αυτή χρησιμοποιείται στη φάση της σημασιολογικής ανάλυσης αλλά και της παραγωγής του τελικού κώδικα.

*(Manis Handbook Chapter 8 'Πίνακας Συμβόλων' , σελ.121)*

```
class Entity:
    def __init__(self, name):
        self.name = name

class Variable(Entity):
    def __init__(self, name):
        super().__init__(name)
        self.datatype = "INTEGER"
        self.offset = 0
        self.nestinglevel = 0
```

Μία αφηρημένη κλάση, **η Entity** τοποθετείτε υψηλότερα στην ιεραρχία και ενοποιεί εννοιολογικά όλες τις εγγραφές. Υπάρχει μόνο ένα πεδίο κοινό σε όλες τις κλάσεις και αυτό είναι το **name,** οπότε η αφηρημένη κλάση Entity έχει μόνο ένα πεδίο. Κάθε εγγραφή που αντιστοιχεί σε κάποια βασική έννοια **κληρονομεί** από την Entity.

Η κλάση **Variable:** ορίζουμε την κλάση Varible με την οποία μπορούμε να δημιουργούμε εισαγωγές στον πίνακα συμβόλων που αναπαριστούν μεταβλητές.Τα πεδία της κλάσης είναι τα εξής:

- datatype : ο τύπος δεδομένων της μεταβλητής ,τον οποίο τον έχουμε θέσει σε INTEGER καθώς στην γλώσσα προγραμματισμού cpy έχουμε μόνο ακέραιες μεταβλητές.
- offset : η απόσταση της μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης
- nestinglevel : το επίπεδο εμφωλιάσματος της μεταβλητής

Στον πίνακα συμβόλων αποθηκεύεται η θέση της μεταβλητής μέσα στο εγγράφημα δραστηριοποίησης της συνάρτησης ,η απόσταση δηλαδή από την αρχή του.Το εγγράφημα δραστηριοποίησης είναι ο χώρος που δίνεται σε μία συνάρτηση για να τοποθετήσει τα δεδομένα της στη μνήμη.

```
class Parameter(Entity):
    def __init__(self, name):
        super().__init__(name)
        self.datatype = "INTEGER"
        self.mode = "cv"
        self.offset = 0
        self.nestinglevel = 0
```

Η κλάση **Parameter:**η κλάση αυτή περιγράφει μια παράμετρο και έχει τα εξής πεδία:

- datatype: τύπος δεδομένων
- mode: τρόπος περάσματος της παραμέτρου που έχει οριστεί σε cv καθώς στην cpy έχουμε πέρασμα παραμέτρων μόνο με τιμή.
- offset: η απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης
- nestinglevel : το επίπεδο εμφωλιάσματος.

```
class Function(Entity):
    def __init__(self, name, startingQuad, activationRecord):
        super().__init__(name)
        self.datatype = "INTEGER"
        self.startingQuad = startingQuad
        self.nestinglevel = 0
        self.offset=0
        self.activationRecord=activationRecord
```

Η κλάση **Function:**περιγράφει μία συνάρτηση , χρησιμοποιείται για να σημειώσει την ύπαρξη ενός υποπρογράμματος στον κώδικα.*(Manis Handbook chapter 8 'Πίνακας Συμβόλων' , σελ.125)*

Τα πεδία της κλάσης είναι τα εξής:

- datatype : τύπος δεδομένων
- startingQuad: . Στο πεδίο αυτό αποθηκεύουμε την ετικέτα της πρώτης εκτελέσιμης τετράδας του ενδιάμεσου κώδικα που αντιστοιχεί στη συνάρτηση αυτή. Εκεί σημειώνουμε την τετράδα στην οποία πρέπει η καλούσα συνάρτηση να κάνει άλμα προκειμένου να ξεκινήσει η εκτέλεση της κληθείσας.
- nestinglevel : επίπεδο εμφωλιάσματος
- offset : απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης
- activationRecord: εγγράφημα δραστηριοποίησης

Οι εγγραφές εισάγονται στον πίνακα συμβόλων κατά τη δήλωσή τους στο αρχικό πρόγραμμα ,ενώ τα πεδία τους συμπληρώνονται όταν η πληροφορία που απαιτείται για τη συμπλήρωσή τους γίνει γνωστή. Για παράδειγμα, η εγγραφή για μία συνάρτηση δημιουργείται και εισάγεται στον πίνακα συμβόλων όταν συναντάμε τη δήλωση συνάρτησης στον κώδικα του αρχικού προγράμματος ,αλλά το πεδίο startingQuad της εγγραφής συμπληρώνεται όταν μεταφραστεί η πρώτη εκτελέσιμη τετράδα ενδιάμεσου κώδικα της υπόμετάφρασης συνάρτησης.

*(Manis Handbook chapter 8 'Πίνακας Συμβόλων' , σελ.123)*

Επιλέξαμε το πεδίο FormalParameters και framelength να διαχειρίζονται από την κλάση ActivationRecord ,οπώς θα φανεί και παρακάτω.

```
class TemporaryVariable(Entity):
    def __init__(self,name,datatype,offset):
        super().__init__(name)
        self.datatype="INTEGER"
        self.offset=0
```

Η κλάση **TemporaryVariable** έχει τα εξής πεδία:

- datatype:ο τύπος δεδομένων της προσωρινής μεταβλητής
- offset: η απόσταση της προσωρινής μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης.

```
class FormalParameters(Entity):
    def __init__(self,datatype,mode):
        super().__init__(name)
        self.datatype="INTEGER"
        self.mode="cv"
        self.nestinglevel=0
        self.offset=0
```

Η κλάση **FormalParameters:** πρόκειται για την κλάση, στιγμιότυπα της οποίας αποτελούν αντικείμενα που περιγράφουν τη δήλωση παραμέτρων της συνάρτησης και έχει τα εξής πεδία:

- datatype:ο τύπος δεδομένων
- mode: τρόπος περάσματος
- nestinglevel : επίπεδο εμφωλιάσματος
- offset: απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης

## Η κλάση **Scope :**

```
class Scope:
    def __init__(self, name):
        self.name = name
        self.entities = []
        self.nestinglevel = 0

    def insert_entity(self, entity):
        if isinstance(entity, Variable) or isinstance(entity, Parameter) or isinstance(entity, TemporaryVariable) or isinstance(entity, Function):
            if self.entities:
                last_offset = self.entities[-1].offset
                entity.offset = last_offset + 4
            else:
                entity.offset = 12
            entity.nestinglevel = self.nestinglevel

        self.entities.append(entity)

    def search_entity(self,name):
        for entity in self.entities:
            if entity.name==name:
                return entity
        return None
```

Η κλάση scope διαχειρίζεται ένα **scope**(επίπεδο).

Αποτελείται απο τον constructor __init__ , που δέχεται ένα όνομα για το scope, δημιουργεί μια κενή λίστα για τα entities που ανήκουν σε αυτο το ορίζει σε 0.

Η μέθοδος **insert_entity()** ελέγχει αρχικά  αν το entity που πρόκειται να εισαχθεί στην λίστα, είναι αντικείμενο κάποιας από τις κλάσεις Variable ,Parameter ,TemporaryVariable, Function. Αν ναι , εφόσον υπάρχουν ήδη στοιχεία στην λίστα ορίζει το offset της νέας εισαγωγής 4 byte απο το τελευταίο στοιχείο.Διαφορετικά,εάν δεν έχουν υπάρχει προηγούμενες εισαγωγές , το offset του στοιχείου είναι 12.

Η μέθοδος **search_entity()** παίρνει ως όρισμα το όνομα ενός entity και το ψάχνει μέσα στα entities του scope. Αν δεν το βρει επιστρέφει **None**.

Το nestinglevel του entity ορίζεται ίσο με το επίπεδο του  scope και στην συνέχεια γίνεται η εισαγωγή του στοιχείου στην entities λιστα (*self.entities.append(entity)* )

Η κλάση **SymbolTable:** Χρησιμοποιείται για την διαχείριση του Πίνακα Συμβόλων.

Η κλάση Scope και η κλάση SymbolTable έχουν σχέση συναρμολόγησης/σύνθεσης, αφού ένα Table αποτελείται από πολλά scopes.

```
class SymbolTable:
    def __init__(self):
        self.scopes = []

    def create_scope(self, name):
        scope = Scope(name)
        if self.scopes:
            scope.nestinglevel=self.scopes[-1].nestinglevel +1
        else:
            scope.nestinglevel=0
        self.scopes.append(scope)
        return scope

    def delete_scope(self):
        if self.scopes:
            self.scopes.pop()

    def insert_entity(self, entity):
        if self.scopes:
            current_scope = self.scopes[-1]
            current_scope.insert_entity(entity)
        else:
            raise Exception("No scope created yet")

    def print_table(self, scope):
        with open("scopes.int", 'a') as f:
            f.write(scope.name + " ::: ")
            for entity in scope.entities[0:-1]:
                f.write("{}({}, Nesting Level: {}), ".format(entity.name, entity.offset, entity.nestinglevel))
            f.write("{}({}, Nesting Level: {}) ".format(scope.entities[-1].name, scope.entities[-1].offset, scope.entities[-1].nestinglevel))
            f.write("\n")
```

Αποτελείται απο τον construction __init__ ,ο οποίος δημιουργεί μια κενή λίστα για τα scopes.

- Η μέθοδος **create_scope() :** δημιουργεί ένα αντικείμενο Scope με το δοθέν όνομα.Αν υπάρχουν ήδη scopes , το nestinglevel ισούται με το nestinglevel του τελευταίου scope αυξημένο κατά 1.Αλλιώς το nestinglevel του scope ισούται με 0.Το νέο scope στην συνέχεια εισάγεται στην λίστα των scopes.
- Η μέθοδος **delete_scope() ,**ελέγχει αρχικά αν υπάρχουν scopes ,και αν ναι,τοτέ διαγράφει το τελευταίο εισαχθέν scope από την λίστα.( *self.scopes.pop()* ),διαφορετικά εμφανίζει μήνυμα λάθους,
- Η μέθοδος **insert_entity():** Ελέγχει αν έχουν δημιουργηθεί scopes και αν ναι, εισάγει το entity στο τρέχον ενεργό scope ,καλώντας την insert_entity() της κλάσης Scope που στην συνέχεια υλοποιεί την εισαγωγή.
- Η μέθοδος print_table() : Είναι υπεύθυνη για την εκτύπωση των λεπτομερειών (ονομα,offset,nestinglevel) του scope σε ένα αρχείο scopes.int .

## Η κλάση **ActivationRecord:**

```
class ActivationRecord:
    def __init__(self,function):
        self.function=function
        self.returnAddress=0
        self.sp=0
        self.returnValueAddress=0
        self.formalPar=[]
        self.localVar=[]
        self.tmpVars=[]
        self.framelength=0

    def insert(self,t):
        global token
        if t=='formalPar':
            self.formalPar.append(token.recognized_string)
            self.framelength=self.framelength+4
        elif t=='localVar':
            self.localVar.append(token.recognized_string)
            self.framelength=self.framelength+4
        elif t=='globalVar':
            self.framelength=self.framelength
```

```
  def __str__(self):
      return f'Activation Record for func {self.function}:\nFormal parameters list: {self.formalPar}
```

*συνέχεια screenshot οριζόντια*

```
\nLocal variables list: {self.localVar}\nTemporary variables list: {self.tmpVars}\nFramelength: {self.framelength} bytes'
```

Η κλάση ActivationRecord: Περιγράφει τα στοιχεία που αποθηκεύονται στο activation record (εγγράφημα δραστηριοποίησης) κατά την εκτέλεση μιας συνάρτησης.Αποτελείται απο τον constructor __init__ ο οποίος αρχικοποιεί ένα εγγράφημα δραστηριοποίησης θέτοντας τις απαραίτητες παραμέτρους και λίστες ,οι οποίες είναι οι εξης :

- Function : το όνομα της συνάρτησης για την οποία δημιουργείται
- Return Address: η διεύθυνση επιστροφής στην συνάρτηση κλήσης.
- Stack Pointer(sp): δείκτης στο σημείο της στοίβας (stack) όπου βρίσκεται το activation record,αρχικοποιείται στο 0.
- Return Value Adress: η διεύθυνση επιστροφής τιμής της συνάρτησης,αρχικοποιείται στο 0.
- Formal Parameters(formalPar) : λίστα τυπικών παραμέτρων της συνάρτησης
- Local Variables(localVar) : λίστα τοπικών παραμέτρων της συνάρτησης
- Temporary Variables(tmpVar) : λίστα προσωρινών παραμέτρων της συνάρτησης
- Framelength : μέγεθος του activation record.

Η μέθοδος **insert():** προσθέτει νέα στοιχεία στο activation record, ελέγχοντας κάθε φορά αν πρόκειται για global Variable,formal Parameter ή Local Parameter και εισάγοντας τα στις κατάλληλες λίστες και ενημερώνοντας το framelength αναλόγως κάθε φορά.Το frame length των global μεταβλητών παραμένει σταθερό για λόγους που θα εξηγηθούν αναλυτικά παρακάτω , στην εξήγηση του Πίνακα Συμβόλων όταν εφαρμόζεται στον Συντακτικό αναλυτή.

Τα αποτελέσματα του symbol Table αποθηκεύονται σε ένα αρχείο scopes.int

Στο Συνταντικό μέρος:

```
class Syntax:
    def __init__(self, lexer, generated_program, symbolTable):
        self.lexer=Lex(token,file_name)
        self.generated_program=generated_program
        self.symbolTable=symbolTable
        self.current_token=None
```

Αρχικά στον constructor του Syntax γίνεται ορισμός του Symbol Table με τις τιμές που δίνονται ως ορίσματα.

```
    def program(self):
        self.generated_program.genQuad('begin_block','program','_','_')
        scope=self.symbolTable.create_scope("program")
        actRec=ActivationRecord('main')
        func=Function('main',self.generated_program.nextQuad(),actRec)
        self.declarations("localVar", func)
        self.functions()
        self.main(func)
        self.symbolTable.print_table(scope)
        self.symbolTable.delete_scope()
        self.generated_program.genQuad('begin_block','program','_','_')
```

Στην συνάρτηση program δημιουργείται ένα νέο scope με όνομα **program** (η main δηλαδή) και αποθηκεύεται στο Symbol Table.

Επίσης δημιουργείται εγγράφημα δραστηριοποίησης **actRec** με όνομα **main.**

```
def declarations(self, k, func):
    global token
    while token.recognized_string=='#int':
        token=self.get_token()
        self.id_list("variable", k, func)

def id_list(self, t, k, func):
    global token
    if token.family=='identifier':
        if t=="variable":
            var=Variable(token.recognized_string)
            self.symbolTable.insert_entity(var)
            func.activationRecord.insert(k)
        else:
            par=Parameter(token.recognized_string)
            self.symbolTable.insert_entity(par)
            func.activationRecord.insert(k)
        token=self.get_token()
        while token.recognized_string==',':
            token=self.get_token()
            if token.family=='identifier':
                if t=="variable":
                    var=Variable(token.recognized_string)
                    self.symbolTable.insert_entity(var)
                    func.activationRecord.insert(k)
                else:
                    par=Parameter(token.recognized_string)
                    self.symbolTable.insert_entity(par)
                    func.activationRecord.insert(k)
                token=self.get_token()
            else:
                self.error("Error: no ID found after ','")
```

Στην μέθοδο **declarations()** γίνεται η δήλωση μεταβλητών και η εισαγωγή τους στο scope του συγκεκριμένου block και στο αντίστοιχο activation record μέσω της μεθόδου **id_list().**

Στην **declarations()** καλείται η μέθοδος **id_list()** για την διαχείριση της λίστας των μεταβλητών.Γίνεται πρώτα η επεξεργασία της μεταβλητής και ανάλογα με τον τύπο της (**t**) δημιουργείται ένα νέο αντικείμενο της κλάσης Entity (Variable ή Parameter) και εισάγεται στον πίνακα συμβόλων Symbol Table,ενώ ενημερώνεται και το αντίστοιχο activation record.Γίνεται έλεγχος για τον αν υπάρχουν περισσότερες μεταβλητές οι οποίες χωρίζονται με ‘,’ και αν ναι, η διαδικασία επαναλαμβάνεται για κάθε μεταβλητή.

```
def glob_decl(self, k, func):
    global token
    while token.recognized_string=='global':
        token=self.get_token()
        self.id_list("variable", k, func)
        self.symbolTable.scopes[-1].entities.pop(-1)
```

Στην μέθοδο **glob_decl()** η Symbol Table είναι προσβάσιμη μέσω του πεδίου **self.symbolTable**. Η τελευταία μεταβλητή που εισήχθει διαγράφεται από το scope, επειδή οι global μεταβλητές προστίθονται στο scope του **program** (της main δηλαδή). Επίσης, εφόσον δεν τις κρατάμε, στο ActivationRecord αν περαστεί ως παράμετρος το string “**globalVar**” δεν θα προστεθούν σε αυτό και το **framelength** του δεν θα αυξηθεί.

```
def function(self):
    global token
    global tmpVarList
    nestedFunctions=[]
    if token.recognized_string=='def':
        token=self.get_token()
        if token.family=='identifier':
            ID=token.recognized_string
            actRec=ActivationRecord(ID)
            func=Function(ID,self.generated_program.nextQuad(),actRec)
            self.symbolTable.scopes[-1].insert_entity(func)
            self.generated_program.genQuad("begin_block", ID,"_","_")
            scope=self.symbolTable.create_scope(ID)
            token=self.get_token()
            if token.recognized_string=='(':
                token=self.get_token()
                self.id_list("parameter", "formalPar", func)
                if token.recognized_string==')':
                    token=self.get_token()
                    if token.recognized_string==':':
                        token=self.get_token()
                        if token.recognized_string=='#{':
                            token=self.get_token()
                            self.declarations("localVar",func)
                            self.nested_functions()
                            self.glob_decl("globalVar",func)
                            while token.family=='identifier' or token.family=='keyword':
                                self.code_block()
                            if token.recognized_string=='#}':
                                self.generated_program.genQuad("end_block",ID,"_","_")
                                func.activationRecord.tmpVars=tmpVarList
                                for v in func.activationRecord.tmpVars:
                                    func.activationRecord.framelength+=4
                                tmpVarList=[]
                                self.symbolTable.print_table(scope)
                                self.symbolTable.delete_scope()
                                token=self.get_token()
                            else:
                                self.error("Error: opened the function but never closed it, missing '#}'")
```

Στην μέθοδο **function()** γίνεται η δήλωση μιας νέας συνάρτησης και φτιάχνουμε ένα αντικείμενο κλάσης Function το οποίο θέλουμε να προσθέσουμε στο scope. Ωστόσο, επειδή θέλουμε να συμπεριλάβουμε και τις εμφωλευμένες συναρτήσεις ως μεταβλητές του κάθε scope (αν αυτές υπάρχουν, βέβαια), προσθέτουμε την συνάρτηση που διαχειριζόμαστε στο πιο πρόσφατο scope και μετά δημιουργούμε το δικό της. Δημιουργείται ένα νέο scope για την συνάρτηση καθώς και ένα νεό αντικείμενο ActivationRecord. Επιπλέον, όταν καλείται η **id_list()** μέσα στην function() παίρνει ως ορίσματα τα '**parameter**', '**formalPar**' και **func**, διοτί τα χρειάζεται η **id_list()** για να προσθέσει στο ActivationRecord της συνάρτησης τις τυπικές παραμέτρους. Εκτός από αυτό, προς το τέλος του κώδικα της **function()** χειριζόμαστε τις προσωρινές μεταβλητές που θέλουμε να προσθέσουμε στο εγγράφημα. Έχουμε ορίσει μια global μεταβλητή **tmpVarList** στην οποία προστίθονται οι προσωρινές μεταβλητές που φτιάχνει η **newTemp()** οπως φαίνεται παρακάτω:

```
def newTemp(self):
    global tmpVarCount,tmpVarList
    s="T_" + str(tmpVarCount)
    tmpVarCount=tmpVarCount+1
    tmpVarList.append(s)
    return s
```

Θέλουμε να κρατάμε τις μεταβλητές που χρησιμοποιήθηκαν για την συνάρτηση που τρέχει κάθε φορά. Για να το πετύχουμε αυτό, αδειάζουμε στο τέλος του κώδικα της **function()** την λίστα **tmpVarList**, ενώ λίγο πιο πριν την έχουμε αποθηκεύσει μέσα στο ActivationRecord της τρέχουσας συνάρτησης και έχουμε αυξήσει το **framelength** τόσες φορές όσες και οι μεταβλητές που ήταν μέσα στο **tmpVarList**.