



Εαρινό Εξάμηνο Ακαδημαϊκού Έτους 2022-2023

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

ΜΕΤΑΦΡΑΣΤΕΣ
κ. ΓΕΩΡΓΙΟΣ ΜΑΝΗΣ

ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΗ ΑΣΚΗΣΗ:
Η ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ CUTEPU

Όνοματεπώνυμο	ΑΜ	Email	Έτος
Αλκιβιάδης Πάτρας	4778	cs04778@uoi.gr	4ο
Θεοχάρης Καζακίδης	4679	cs04679@uoi.gr	4ο

ΕΝΟΤΗΤΕΣ ΑΝΑΦΟΡΑΣ

1. ΕΝΟΤΗΤΑ: “ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ”
 2. ΕΝΟΤΗΤΑ: “ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ”
 3. ΕΝΟΤΗΤΑ: “ΕΝΔΙΑΜΜΕΣΟΣ ΚΩΔΙΚΑΣ”
 4. ΕΝΟΤΗΤΑ: “ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ”
 5. ΕΝΟΤΗΤΑ: “ΤΕΛΙΚΟΣ ΚΩΔΙΚΑΣ”
 6. ΕΝΟΤΗΤΑ: “ΣΧΟΛΙΑΣΜΟΣ ΠΡΟΒΛΗΜΑΤΩΝ”
-

1. Ενότητα: “Λεκτικός Αναλυτής”

1.1 Περίληψη Ενότητας

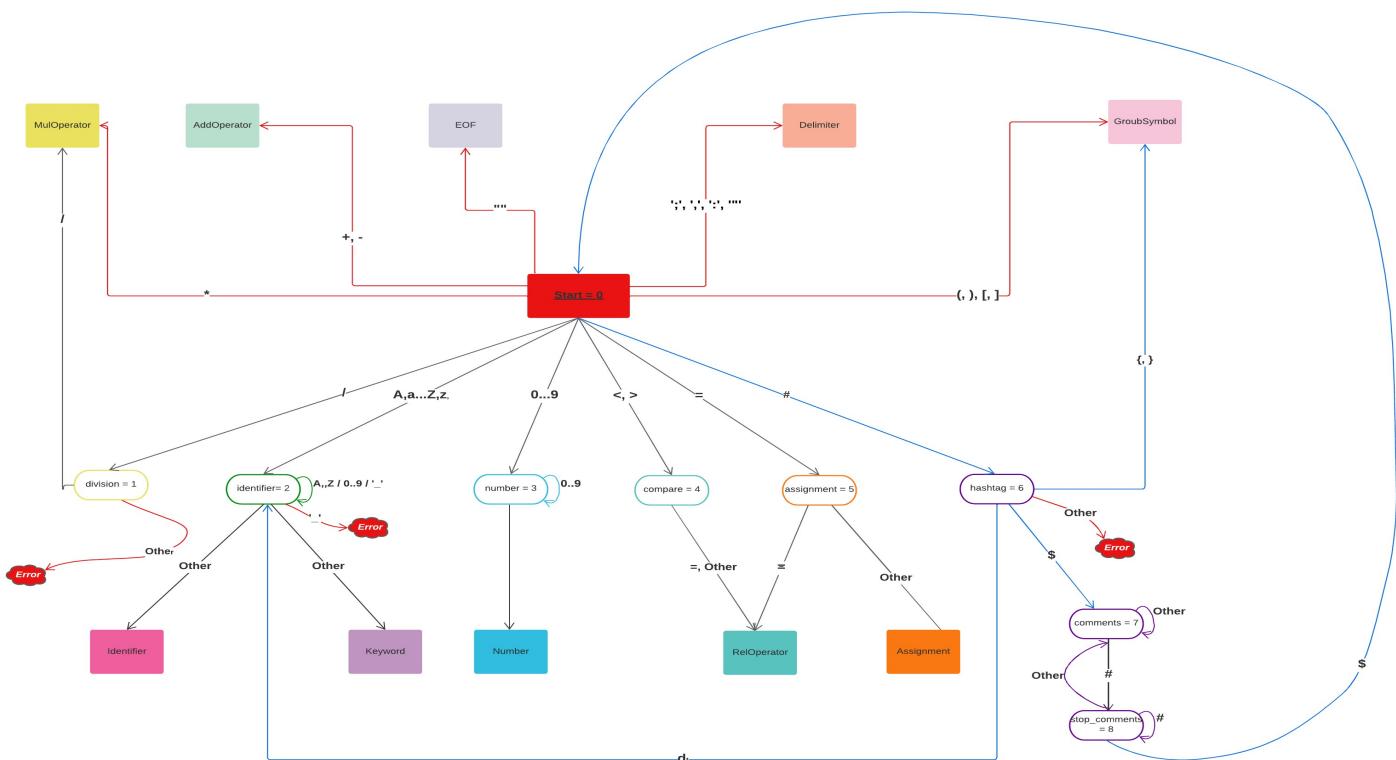
Στην πρώτη ενότητα θα σχολιάσουμε την υλοποίηση μας για τον λεκτικό αναλυτή. Αρχικά θα παρουσιάσουμε ένα διάγραμμα «Αυτόματο Στοίβας» με σκοπό να διευκολύνουμε την κατανόηση του κώδικα μας.

Υστέρα θα αναφέρουμε τον τρόπο με τον οποίο οργανώσαμε τα απαραίτητα δεδομένα για την δημιουργία του λεκτικού αναλυτή όπως το αλφάριθμο της γλώσσας CutePy, τους τύπους/families των tokens και τα αντικείμενα της κλάσης “Token” που επιστρέφει η συνάρτηση “lex()” δηλαδή ο λεκτικός αναλυτής.

Τέλος θα σχολιάσουμε εκτενώς κάθε «μπλοκ» κώδικα της συνάρτησης lex() (λεκτικός αναλυτής).

1.2 Διάγραμμα «Αυτόματο Στοίβας»

Για την υλοποίηση του διαγράμματος χρησιμοποιήσαμε το διαδικτυακό εργαλείο «Lucid.app», όπου προσπαθήσαμε να υλοποιήσουμε ένα αντίστοιχο διάγραμμα με αυτό που υπάρχει στις διαφάνειες του μαθήματος.



1.3 Οργάνωση/Αρχικοποίηση Λεκτικού Αναλυτή

Όλο το αλφάβητο της γλώσσας CutePy έχει αρχικοποιηθεί σε λίστες. Στην αρχή του προγράμματος όπως μας παροτρύνουν και οι διαφάνειες του μαθήματος δημιουργήσαμε μια κλάση «Token» στην οποία, εάν ένα token είναι έγκυρο θα ενημερώνουμε καταλλήλως τα πεδία της με τιμή του token, τον τύπο του και τον αριθμό γραμμής στην οποία εμφανίστηκε στον πηγαίο κώδικα εισόδου CutePy.

```
# The Token class is defined in accordance with the specifications provided in 'lex.pdf'.
class Token:
    def __init__(self, recognized_string, family, line_number):
        self.recognized_string = recognized_string
        self.family = family
        self.line_number = line_number
```

Ως γενική παρατήρηση, η λεκτική ανάλυση περιλαμβάνει τρεις διαδικασίες:

1. Τη δημιουργία των tokens διαβάζοντας γράμμα προς γράμμα τον πηγαίο κώδικα εισόδου της γλώσσας Cutedpy.
2. Την κατηγοριοποίηση των tokens με βάση τον τύπο τους.
3. Και τη διασφάλιση ότι τα tokens είναι έγκυρα σύμφωνα με τους κανόνες της γλώσσας.

Όπως αναφέραμε και προηγούμενος όταν ένα token είναι έγκυρο, επιστρέφεται ως αντικείμενο της κλάσης Token, σε αντίθετη περίπτωση φροντίζουμε να ανακύψει σφάλμα.

Στον πίνακα που ακολουθεί φαίνονται τα ονόματα των τύπων που επιλέξαμε για να κατηγοριοποιήσουμε τα tokens (1^η στήλη) σε αντιπαράθεση με το αλφάβητο της γλώσσας CutedPy που αποτελούνται τα tokens αυτά (2^η στήλη).

Ονοματολογία Τύπων/Οικογενειών κατηγοριοποίησης των Tokens	Λίστες με το Αλφάβητο της Γλώσσας CutedPy
"Identifier"	identifier = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '_']

"Number"	number = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
"Keyword"	keyword = ['def', 'print', 'input', '#declare', 'if', 'while', 'else', 'return', 'and', 'or', 'not', 'int', '__name__', '__main__']
"MulOperator"	mulOperator = ['*', '//']
"RelOperator"	compare = ['<', '>']
"RelOperator"	relOperator = ['!=', '<=', '>=', '==']
"AddOperator"	addOperator = ['+', '-']
"Assignment"	assignment = ['=']
"Delimiter"	delimiter = [';', ',', ':', '"']
"GroupSymbol"	groupSymbol_with_hashtag = ['#{', '#}', '#\$']
"GroupSymbol"	groupSymbol_without_hashtag = ['(', ')', '[', ']']

- Όπως παρατηρείτε για κάποιες περιπτώσεις του αλφάβητου της γλωσσάς CutePy δημιουργήσουμε δυο λίστες για τον ίδιο τύπο token, εξαιτίας του τρόπου με τον οποίο επιλέξαμε να υλοποιήσουμε την συνάρτηση του λεκτικού αναλυτή «lex()» στον κώδικα μας.

1.4 Ανάλυση του κώδικα για τον Λεκτικού Αναλυτή

1.4.1

Αρχικά όπως αναφέραμε και παραπάνω ορίσαμε λίστες με έγκυρα σύμβολα για την δημιουργία των tokens, της γλώσσας CutePy.

Οι λίστες αυτές με το έγκυρο αλφάβητο της γλώσσας κατηγοριοποιούνται όπως φαίνεται παρακάτω στον κώδικα:

```

# The alphabet of the CutePy language is defined as lists.
identifier = [
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd',
    'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
    't', 'u', 'v', 'w', 'x', 'y', 'z', '_'
]

number = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

keyword = [
    'def', 'print', 'input', '#declare', 'if', 'while', 'else', 'return', 'and',
    'or', 'not', 'int', '__name__', '__main__'
]

mulOperator = ['*', '//']

compare = ['<', '>']

relOperator = ['!=', '<=', '>=', '==']

addOperator = ['+', '-']

assignment = ['=']

delimiter = [';', ',', ':', '"']

groupSymbol_without_hashtag = ['(', ')', '[', ']']

groupSymbol_with_hashtag = ['#(', '#)', '#$']

family = [
    "Identifier", "Number", "Keyword", "MulOperator", "RelOperator",
    "AddOperator", "Assignment", "Delimiter", "GroupSymbol"
]

```

1.4.2

Συνάρτηση lex()

Αυτή η συνάρτηση είναι υπεύθυνη για την εκτέλεση της λεκτικής ανάλυσης. Διαβάζει το αρχείο εισόδου CutePy χαρακτήρα προς χαρακτήρα και κατασκευάζει τα tokens. Η διαδικασία κατασκευής των tokens εκτελείται στο πλαίσιο μιας μηχανής καταστάσεων (Αυτόματο Στοίβας) με επτά καταστάσεις αριθμημένες από το 0 έως το 6, καθεμία από τις οποίες αντιπροσωπεύει μια διαφορετική συνθήκη/κανόνα για το token που κατασκευάζεται.

```

# Lexical Analysis
line_number = 1

def lex():
    global line_number # Accessing the variable 'line_number'.
    state = 0
    token = ""

    # The beginning of the 'while' loop in the 'lex' function marks the start of the tokenization process of the input file.
    while state >= 0 and state <= 6:
        char = file.read(1) # Reading one character at a time from the input file
        token += char # Appending the character to the token string

```

1.4.3

Κατάσταση 0: Έναρξη

Αυτή η κατάσταση χειρίζεται τον αρχικό χαρακτήρα ενός token που πρόκειται να σχηματιστεί.

Με άλλα λόγια, ελέγχει αν ο χαρακτήρας ανήκει σε κάποια από τις λίστες συμβόλων.

Εάν είναι αληθής ο έλεγχος αυτός, η συνάρτηση δημιουργεί αμέσως ένα token.

Για χαρακτήρες που απαιτούν επιπρόσθετο έλεγχο (όπως η κάθετος '/' που μπορεί να είναι μέρος ενός τελεστή διαίρεσης ακεραίων), η μηχανή καταστάσεων μεταβαίνει σε ενδιάμεση κατάσταση ενώ για χαρακτήρες που δεν επιθυμούν περεταίρω ελέγχους η μηχανή καταστάσεων μεταβαίνει σε τελική κατάσταση και επιστρέφει ένα αντικείμενο της κλάσης "Token", όπου η τιμή του token είναι ο ίδιος ο χαρακτήρας που διάβασε από το αρχείο εισόδου.

Για τους χαρακτήρας που δεν ανήκουν σε κάποια λίστα δηλαδή δεν ανήκουν στο αλφάβητο της γλώσσας η κατάσταση μηδέν τυπώνει μήνυμα λάθους στον χρήστη και τερματίζει το πρόγραμμα.

```
# Lexical Analysis
line_number = 1

def lex():
    global line_number # Accessing the variable 'line_number'.
    state = 0
    token = ""

    # The beginning of the 'while' loop in the 'lex' function marks the start of the tokenization process of the input file.
    while state >= 0 and state <= 6:
        char = file.read(1) # Reading one character at a time from the input file
        token += char # Appending the character to the token string

        # state 'Start' (state = 0)
        # This state acts like a manager, where some symbols immediately transition to a final state and return a Token object,
        # while other symbols like letters and numbers require additional checks and need to be sent to other states before reaching a final state.
        # However, all symbols are checked in the 'Start' state. If a symbol is invalid, a message is printed to indicate invalid input in the cutePy file.
        if state == 0:
            # If the character is a newline or a carriage return, increment line_number and reset token.
            if char in ('\n', '\r'):
                line_number += 1
                token = ""

            # If the character is empty, return a token indicating end of file.
            elif char == '':
                return Token("!!!", "EOF", line_number)

            # If the character is a whitespace, remove the whitespace from the token string.
            elif token.isspace():
                token = token[:-1]

            # If the token is an addition operator, return a token of type 'AddOperator'.
            elif token in addOperator: # 'addOperator' => name of list
                return Token(token, 'AddOperator', line_number)

            # If the token is a group symbol without a hashtag, return a token of type 'GroupSymbol'.
            elif token in groupSymbol_without_hashtag: # 'groupSymbol_without_hashtag' => name of list
                return Token(token, 'GroupSymbol', line_number)

            # If the token is a delimiter, return a token of type 'Delimiter'.
            elif token in delimiter:
                return Token(token, 'Delimiter', line_number)

            # If the token is a multiplication operator, return a token of type 'MulOperator'.
            # If the token is a forward slash ('/'), the lexer moves to the 'Division' state (state = 1)
            elif token in mulOperator or token == '/':
                if token == '*':
                    return Token(token, 'MulOperator', line_number)
                else:
                    state = 1
```

```

# If the token is a valid identifier, move to the 'Identifier' state (state = 2)
elif token in identifier:
    state = 2

# If the token is a valid number, move to the 'Number' state (state = 3)
elif token in number:
    state = 3

# If the token is a comparison operator or an exclamation point, move to the 'Compare' state (state = 4)
elif token in compare or token == '!':
    state = 4

# If the token is an assignment operator, move to the 'Assignment' state (state = 5)
elif token in assignment:
    state = 5

# If the token is a hashtag, move to the 'Hashtag' state (state = 6)
elif token == '#':
    state = 6

else:
    # The code generates an error message indicating that the input contains an invalid character and prints the line number where the error occurred.
    error_message = f"Invalid input. The character you are trying to type is not recognized in CutePy. Please check your input. Line: {line_number}"
    raise ValueError(error_message)

```

1.4.4

Κατάσταση 1: Διαίρεση

Μεταβαίνουμε στην παρούσα κατάσταση όταν ο αρχικός χαρακτήρας είναι η κάθετος '/'. Η συνάρτηση ελέγχει αν ο επόμενος χαρακτήρας είναι επίσης μια κάθετος, η οποία θα ολοκλήρωνε τον τελεστή διαίρεσης ακέραιου αριθμού. Αν δεν είναι, η συνάρτηση δημιουργεί σφάλμα, καθώς μια απλή κάθετος προς τα εμπρός δεν είναι έγκυρο σύμβολο του αλφαριθμητικού της CutePy.

```

# state 'division' (state = 1)
# This block of code checks if the current token is '/'.
# If it is, the lexer reads the next character to check if it is also a forward slash '/'.
# If it is, the lexer returns a MulOperator token
# Otherwise, it raises a ValueError.
if state == 1:
    if token == '/':
        next_char = file.read(1)
        if next_char == '/':
            token += next_char
            return Token(token, 'MulOperator', line_number)
        else:
            file.seek(file.tell() - 1)
            raise ValueError(
                f"Invalid input. Please ensure that you have used two forward slashes ('//') to start division. Line: {line_number}"
            )

```

1.4.5

Κατάσταση 2: Αναγνωριστικό

Αυτή η κατάσταση χειρίζεται την κατασκευή των αναγνωριστικών tokens. Συνεχίζει να διαβάζει χαρακτήρες εφόσον ανήκουν στις λίστες αναγνωριστικών ή αριθμών. Εάν το αναγνωριστικό είναι επίσης λέξη-κλειδί, επιστρέφει ένα αντικείμενο λέξη-κλειδί- διαφορετικά, επιστρέφει ένα αντικείμενο αναγνωριστικού. Η συνάρτηση δημιουργεί σφάλμα αν το αναγνωριστικό αρχίζει με '#' ή '_' ή αν το μήκος του υπερβαίνει τους 30 χαρακτήρες.

```

# state 'identifier' (state = 2)
# This block of code checks if the current token is in identifier list or starts with '#'.
# If it is, the lexer reads the next character and continue adding to token until a non-identifier or a non-number character is encountered.
# If the next character doesn't meet conditions, lexer moves file pointer back one character.
# Next lexer checks if the current token is less than 30 characters and if it is a keyword or an identifier.
# It returns a Keyword token or an Identifier token, accordingly.
# Otherwise, it raises an error.
if state == 2:
    if not token in identifier and token[0] != '#':
        raise ValueError("Invalid input.")
    while True:
        next_char = file.read(1)
        if next_char in identifier or next_char in number:
            token += next_char
        else:
            file.seek(file.tell() - 1)
            if token in keyword:
                return Token(token, 'Keyword', line_number)
            else:
                if token[0] == ' ' or token[0] == '#':
                    raise ValueError(
                        f"Invalid input. Please ensure that you have not used '#' or '_' to start an Identifier token. Line: {line_number}")
                elif len(token) > 30:
                    raise ValueError(f"Invalid input. Please ensure that your input identifier falls within the acceptable range of characters. Line: {line_number}")
                else:
                    return Token(token, 'Identifier', line_number)

```

1.4.6

Κατάσταση 3: Αριθμός

Σε αυτή την κατάσταση, η συνάρτηση κατασκευάζει ένα αριθμητικό token. Συνεχίζει να διαβάζει χαρακτήρες εφόσον ανήκουν στη λίστα αριθμών. Κάθε μη αριθμητικός χαρακτήρας ή αριθμός που υπερβαίνει τη μέγιστη επιτρεπόμενη τιμή (MAX_NUMBER = 2**32) προκαλεί την εμφάνιση σφάλματος.

```

# state 'number' (state = 3)
# This block of code checks if the current token is in number list.
# If it is, lexer reads the next character and continue adding to token until non-number character is encountered.
# If the next character doesn't meet conditions, lexer moves file pointer back one character.
# Next lexer checks if the current token is a number and doesn't contains letters.
# if the token meets the above conditions and belongs within the acceptable range for numbers, lexer returns a Number token.
# Otherwise, it raises an error.
if state == 3:
    if not token.isdigit():
        raise ValueError("Invalid input.")
    while True:
        next_char = file.read(1)
        if next_char in identifier:
            raise ValueError(f"Invalid input. Please ensure that you have not used letters in a number. Line: {line_number}")
        if next_char in number:
            token += next_char
        else:
            file.seek(file.tell() - 1)
            if token.isdigit() and int(token) < MAX_NUMBER:
                return Token(token, 'Number', line_number)
            else:
                raise ValueError(f"Invalid input. Please ensure that your input number falls within the acceptable range of digits. Line: {line_number}")

```

1.4.7

Κατάσταση 4: Σύγκριση

Αυτή η κατάσταση είναι για τη δημιουργία token που αφορούν τελεστές σύγκρισης για την γλώσσα μας. Ελέγχει αν ο πρώτος χαρακτήρας ανήκει στην λίστα με τα σύμβολα σύγκρισης ή είναι ο '!'. Στην συνέχεια κοιτάει τον επόμενο χαρακτήρα και ελέγχει αν είναι '='. Οποιοσδήποτε έλεγχος και αν είναι αληθής (είτε ένας από τους δυο είτε και οι δυο μαζί) η κατάσταση επιστρέφει το token που σχηματίστηκε με τύπο 'RelOperator'. Στην περίπτωση που ο πρώτος χαρακτήρας είναι '!' και δεν ακολουθείτε από '=' η κατάσταση επιστρέφει μήνυμα σφάλματος.

```

# state 'compare' (state = 4)
# This block of code checks if the current token is in compare list or starts with '!'.
# If it is, the lexer reads the next character to check if it is '='.
# if it is, lexer returns a RelOperator token [>=, <=, !=]
# If the next character isn't '=', lexer moves file pointer back one character.
# Next lexer checks if the current token is in compare list, it returns a RelOperator token.
# Otherwise, it raises an error.
if state == 4:
    if token in compare or token == '!':
        next_char = file.read(1)
        if next_char == '=':
            token += next_char
            return Token(token, 'RelOperator', line_number)
        else:
            if token in compare:
                file.seek(file.tell() - 1)
                return Token(token, 'RelOperator', line_number)
            else:
                raise ValueError(
                    f"Invalid input. Please ensure that you have used '=' to start the RelOperator '!='. Line: {line_number}"
                )

```

1.4.8

Κατάσταση 5: Ανάθεση

Αυτή η κατάσταση είναι υπεύθυνη για τη δημιουργία τελεστών ανάθεσης και ισότητας. Εάν ο πρώτος χαρακτήρας είναι '=' και ο επόμενος '=', δημιουργεί τον τελεστή ισότητας- διαφορετικά, δημιουργεί τον τελεστή ανάθεσης.

```

# state 'assignment' (state = 5)
# This block of code checks if the current token is in assignment list.
# If it is, the lexer reads the next character to check if it is '='.
# If it is, lexer returns a RelOperator token [==]
# If the next character isn't '=', lexer moves file pointer back one character.
# Next lexer checks if the current token is in assignment list, it returns an Assignment token.
if state == 5:
    if token in assignment:
        next_char = file.read(1)
        if next_char == '=':
            token += next_char
            return Token(token, 'RelOperator', line_number)
        else:
            file.seek(file.tell() - 1)
            return Token(token, 'Assignment', line_number)

```

1.4.9

Κατάσταση 6: Hashtag

Αυτή η κατάσταση ενεργοποιείται όταν ο αναγνωριστεί ένα σύμβολο ως '#'. Μόλις ο λεκτικός αναλυτής βρεθεί σε αυτή την κατάσταση, θα διαβάσει τον επόμενο χαρακτήρα προκειμένου να τον προσδιορίσει. Αναλόγως του προσδιορισμού αυτού διακρίνουμε τρεις κατηγορίες:

- i. Εάν ο επόμενος χαρακτήρας είναι '{' ή '}', η συνάρτηση κατατάσσει το token '#{' #'}' αυτό στον τύπο 'GroupSymbol'.
- ii. Εάν ο επόμενος χαρακτήρας μετά τo '#' είναι '\$', το token ερμηνεύεται ως την αρχή ενός σχόλιου. Η κατάσταση 6 θα συνεχίσει να διαβάζει τους επόμενους χαρακτήρες σε έναν

βρόχο μέχρι να εντοπίσει το '#\$' που σημαίνει και το τέλος ενός σχόλιου. Κατά τη διαδικασία αυτή λαμβάνουμε υπόψιν, πρώτον τις νέες γραμμές και ενημερώνουμε την καθολική μεταβλητή line_number και δεύτερον τυπώνουμε σφάλμα εάν συναντήσουμε το τέλος του αρχείου πριν από την ολοκλήρωση του σχολίου. Αυτό γίνεται για να εξασφαλιστεί ότι όλα τα σχόλια κλείνουν σωστά με το '#\$'. Προκειμένου να αγνοήσουμε τα σχόλια του κώδικα εισόδου της CutePy μεταβαίνουμε στην "Κατάσταση 0: Έναρξης" με το που αναγνωρίσουμε το token '#\$' για δεύτερη φορά μιας και σηματοδοτεί και την λήξη τους.

- iii. Τέλος εάν ο χαρακτήρας που ακολουθεί το '#' είναι 'd', μεταβαίνουμε στην "Κατάσταση 2: Αναγνωριστικό" μιας και υπάρχει μια λέξη-κλειδί για την γλώσσα CutePy που ξεκινάει με '#d' ή "#declare".

Στην περίπτωση που καμία από τις παραπάνω συνθήκες δεν πληρούνται, δηλαδή εάν ο χαρακτήρας που ακολουθεί το '#' δεν είναι '{', '}', '\$' ή 'd' τυπώνουμε μήνυμα σφάλματος και τερματίζουμε το πρόγραμμα.

```
# state 'Hashtag' (state = 6)
# This block of code checks if the current token is '#'.
# If it is, the lexer reads the next character and checks if it is '(' or ')', it returns GroupSymbol token.
# Otherwise if the next character is '$', lexer adds it to token and reads the next characters and continue adding to token until token ends with '#$'.
# When lexer reads '#$', doesn't return anything and goes to state = 0 in order to ignore the comments of cutePy's file.
# Also if lexer reads a next line character, it informs the variable line_number and if it reads the end of file, it rises an error.
# Next it checks if the current token is 'd' and the next character is 'd', lexer goes to state 'Identifier' to check if this token is about '#declare'.
# Otherwise, If the next character isn't anything of the above, it moves file pointer back one character and rises an error.
if state == 6:
    if token == '#':
        next_char = file.read(1)
        if next_char == '(' or next_char == ')':
            token += next_char
            return Token(token, 'GroupSymbol', line_number)
        elif next_char == '$':
            token += next_char
            while True:
                next_next_char = file.read(1)
                if next_next_char == '#' and file.read(1) == '$':
                    state = 0
                    token = ""
                    break
                if next_next_char in ('\n', '\r'):
                    line_number += 1
                if next_next_char == '':
                    raise ValueError(f"Invalid input. Please ensure that you have used '#$' to close the comments. Line: {line_number}")
            elif next_char == 'd':
                token += next_char
                state = 2
            else:
                file.seek(file.tell() - 1)
                raise ValueError(f"Invalid input. Please ensure that you have not used only '#' without anything else after that. Line: {line_number}")
    else:
```

1.5 Συμπέρασμα

Αφού λοιπόν εξηγήσαμε παραπάνω για κάθε κατάσταση τους ελέγχους που εκτελεί για τα tokens καθώς και τα μηνύματα λάθους που επιστρέφει στο χρήστη, ας δούμε τώρα συνολικά για κάθε κατάσταση σε ποιες τελικές καταστάσεις μπορεί να οδηγεί στο παρακάτω πίνακα.

State = 0 "Start"	Token("","", 'EOF', line_number) Token(token, 'AddOperator', line_number) Token(token, 'GroupSymbol', line_number) Token(token, 'Delimiter', line_number) Token(token, 'MulOperator', line_number)
-------------------	--

State = 1 "division"	Token(token, 'MulOperator', line_number)
State = 2 "identifier"	Token(token, 'Keyword', line_number)
	Token(token, 'Identifier', line_number)
State = 3 "number"	Token(token, 'Number', line_number)
State = 4 "compare"	Token(token, 'RelOperator', line_number)
	Token(token, 'RelOperator', line_number)
State = 5 "assignment"	Token(token, 'Assignment', line_number)
	Token(token, 'GroupSymbol', line_number)
State = 6 "hashtag"	

2. Ενότητα: “Συντακτικός Αναλυτής”

2.1 Περίληψης Ενότητας

Για την επεξήγηση της υλοποίησης του συντακτικού αναλυτή θα αναφέρουμε αρχικά τον τρόπο σκέψης πίσω από τον κώδικα που γράψαμε. Υστέρα μιας και πολλές συναρτήσεις κατασκευάστηκαν με το ίδιο μοτίβο υλοποίησης θα αναλύσουμε διεξοδικά μόνο μια συνάρτηση για κάθε κατηγορία. Τέλος κάτω από κάθε επεξήγηση μιας συνάρτησης (ανά κατηγορία) θα αναφέρουμε και αυτές που υλοποιούνται με το ίδιο τρόπο.

2.2 Κανόνες Γραμματικής

Ο κώδικας συντακτικής ανάλυσης για τη γλώσσα CutePy έχει σχεδιαστεί για να επικυρώνει τη δομή ενός προγράμματος CutePy σύμφωνα με τους γραμματικούς κανόνες της γλώσσας. Το πρώτο βήμα είναι η εξέταση των κανόνων γραμματικής της γλώσσας CutePy.

Η συντακτική ανάλυση πραγματοποιείται μέσω μιας σειράς συναρτήσεων, καθεμία από τις οποίες αντιστοιχεί σε έναν συγκεκριμένο γραμματικό κανόνα. Χρησιμοποιεί έναν λεκτικό αναλυτή (που αναπαρίσταται από τη συνάρτηση `lex()`) για να χαρακτηρίσει τον κώδικα του αρχείου εισόδου CutePy και να τον επικυρώσει σύμφωνα με τη γραμματική.

Για κάθε συνάρτηση ή όνομα μπλοκ της γραμματικής, δημιουργούμε μια αντίστοιχη υπό-συνάρτηση μέσα στην συνάρτηση “`syntax()`” του συντακτικού αναλυτή. Αυτές οι συναρτήσεις θα είναι υπεύθυνες για την ανάλυση και την επικύρωση της δομής που ορίζεται από τους κανόνες της γραμματικής.

`syntax()`: Αυτό είναι το κύριο σημείο εισόδου για την ενότητα συντακτικής ανάλυσης. Αρχικοποιεί τις παγκόσμιες μεταβλητές `line_number`, `token` και `ACCESS` (για να εισάγει το πρώτο αλφαριθμητικό από το αρχείο εισόδου) και στη συνέχεια καλεί τη συνάρτηση `startRule()` για να ξεκινήσει η διαδικασία συντακτικής ανάλυσης.

Καθ' όλη τη διάρκεια της διαδικασίας της συντακτικής ανάλυσης, συμπεριλαμβάνουμε ελέγχους σφαλμάτων για να διασφαλίσουμε ότι ο κώδικας εισόδου τηρεί τους κανόνες γραμματικής. Εάν εντοπιστεί σφάλμα, δημιουργούμε μια εξαίρεση `ValueError` με ένα λεπτομερές μήνυμα σφάλματος για να βοηθήσουμε τον χρήστη να εντοπίσει και να διορθώσει το συντακτικό σφάλμα. Τα μηνύματα σφάλματος περιλαμβάνουν αναλυτικές πληροφορίες σχετικά με τη φύση του σφάλματος και τον αριθμό της γραμμής όπου εμφανίστηκε.

Πιο αναλυτικά, μια επισκόπηση μερικών κανόνων της γραμματικής που τους μετατρέψαμε σε συναρτήσεις είναι:

`startRule()`: Αυτή η συνάρτηση αντιπροσωπεύει το σημείο εκκίνησης της γραμματικής του CutePy. Καλεί διαδοχικά τις συναρτήσεις `def_main_part()` και `call_main_part()`.

`def_main_part()`: Αυτή η συνάρτηση είναι υπεύθυνη για την επεξεργασία του ορισμού (των ορισμών) της κύριας συνάρτησης. Καλεί επανειλημμένα τη συνάρτηση `def_main_function()` για κάθε ορισμό κύριας συνάρτησης που συναντάται στον κώδικα.

`def_main_function()`: Αυτή η συνάρτηση επεξεργάζεται έναν ορισμό κύριας συνάρτησης αναλύοντας και επικυρώνοντας τη δομή του. Χειρίζεται επίσης τις δηλώσεις και τις κλήσεις της συνάρτησης `def_function()` για ένθετους ορισμούς συναρτήσεων.

`def_function()`: Αυτή η συνάρτηση επεξεργάζεται έναν ορισμό συνάρτησης, αναλύοντας τη δομή του και διαχειρίζομενη τις δηλώσεις, τους ορισμούς φωλιασμένων συναρτήσεων και τις δηλώσεις του.

`declarations()`, `declaration_line_number()` και `id_list()`: Αυτές οι συναρτήσεις διαχειρίζονται τις δηλώσεις μεταβλητών τόσο στη `main` όσο και σε άλλες συναρτήσεις.

`statement()` και `statements()`: Αυτές οι συναρτήσεις διαχειρίζονται δηλώσεις, τόσο απλές όσο και δομημένες, μέσα σε μπλοκ συναρτήσεων.

`simple_statement()`, `structured_statement()`, `assignment_stat()`, `print_stat()` και `return_stat()`: Αυτές οι συναρτήσεις διαχειρίζονται συγκεκριμένους τύπους δηλώσεων, συμπεριλαμβανομένων των δηλώσεων ανάθεσης, των δηλώσεων εκτύπωσης και των δηλώσεων επιστροφής.

`if_stat()` και `while_stat()`: Αυτές οι συναρτήσεις χειρίζονται τις δηλώσεις 'if' και 'while' αντίστοιχα, αναλύοντας τις συνθήκες και τις σχετικές δηλώσεις τους.

2.3 Τρόπος Σκέψης Υλοποίησης Κανόνων Γραμματικής υπό μορφή Συναρτήσεων

Για να μετατρέψουμε του κανόνες της γραμματικής σε υπο-συναρτήσεις μέσα στην συνάρτηση `syntax()` ακολουθήσαμε την εξής διαδικασία:

- i. Αρχικά δημιουργήσαμε μια συνάρτηση για κάθε κανόνα γραμματικής της εκφώνησης, όπως αναφέραμε και παραπάνω.
- ii. Υστέρα για κάθε `token` που υπαγόρευε ένας κανόνας γραμματικής δημιουργήσαμε έναν έλεγχο `if()` προκειμένου να ελέγχουμε την ύπαρξη του και αμέσως μετά καλούσαμε τον λεκτικό αναλυτή `lex()` καθώς και την γραμμή που βρισκόμαστε ώστε να την χρησιμοποιήσουμε μετέπειτα στο μήνυμα λάθους.
- iii. Για κάθε κανόνα που υπήρχε στην γραμματική από την στιγμή που τον είχαμε μετατρέψει σε υπο-συνάρτηση, τον καλούσαμε μέσα στο αντίστοιχο μπλοκ κώδικα.
- iv. Για κάθε κανόνα ή `token` που βρισκόταν στην γραμματική υπό τον συμβολισμό `()*` ή `()+` δημιουργούσαμε ένα βρόχο `while` με την διαφορά ότι στην δεύτερη περίπτωση γράφαμε τον ίδιο κανόνα (συνάρτηση) ακόμα μια φορά εκτός του βρόχου προκειμένου να διασφαλίσουμε ότι εμφανίζεται τουλάχιστον μια φορά.
- v. Τέλος για κάθε "|" της γραμματικής εκτελούσαμε έναν ακόμα έλεγχο 'else' διασφαλίζοντας όλες τις δυνατές περιπτώσεις που έλεγε ο κανόνας.

Κάποια ενδεικτικά παραδείγματα που επιβεβαιώνουν την παραπάνω ανάλυση είναι τα εξής:

2.3.1

```
def_main_part
:   ( def_main_function )+
;

def_main_function
:   'def' ID '(' ')' ':'
'#{'
    declarations
    ( def_function )*
    statements
'#}'
;

def def_main_part():
    global ACCESS
    def_main_function()
    while (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        def_main_function()

def def_main_function():    # Testing: def main_factorial(): #{ .... #
    global line_number
    global token
    global ACCESS

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Identifier"):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ';'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '#{'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        declarations()
        while (ACCESS.family == "Keyword"
            and ACCESS.recognized_string == 'def'):
            def_function()
            statements()

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '#('):
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing the two block's closing symbols after the statements of a main function. In line: {line_number}")
    else:
        raise ValueError(f"Missing the two block's openning symbols before the declarations of a main function. In line: {line_number}")
    else:
        raise ValueError(f":' after right and left parenthesis of a main function. In line: {line_number}")
    else:
        raise ValueError(f"Missing right parenthesis ')', not closed after the name of a main function. In line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened after the name of a main function. In line: {line_number}")
    else:
        raise ValueError(f"Missing the 'name' of main function. In line: {line_number} after token: {token}")
    else:
        raise ValueError(f"Missing the keyword 'def' at the beginning of the program's block code. In line: {line_number} before token: {token}")


```

2.3.2

```
assignment_stat
    :   ID '='
        (   expression ;'
        |   'int' '(' 'input' '(' ')' ')' ;'
        )
    ;
;

else:
    raise ValueError(f"Missing semicolon ';' after expression in assignment statement. Line: {line_number}")
else:
    raise ValueError(f"Missing assignment symbol '=' after Identifier token, in assignment statement. Line: {line_number}")
else:
    raise ValueError(f"Missing Identifier token in assignment statement. Line: {line_number}")

def assignment_stat():
    global ACCESS
    global line_number

    if (ACCESS.family == "Identifier"):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Assignment" and ACCESS.recognized_string == '='):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'int'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Keyword"
        and ACCESS.recognized_string == 'input'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ';'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing semicolon ';' in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing right parenthesis ')', not closed in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing right parenthesis ')', not closed in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing the keyword 'input' in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened in assignment statement. Line: {line_number}")

else:
    expression()

if (ACCESS.family == "Delimiter"
    and ACCESS.recognized_string == ';'):
    ACCESS = lex()
    line_number = ACCESS.line_number
```

3. Ενότητα: “Ενδιάμεσος Κώδικας”

3.1 Περίληψη Ενότητας

Στην ενότητα αύτη θα παρουσιάσουμε αρχικά όλες τις «Βοηθητικές Υπορουτίνες» σχολιάζοντας εκτενώς τον κώδικα που δημιουργήσαμε για την υλοποίηση τους.

Επίσης θα αναφέρουμε όλες τις τετράδες ‘quads’ που χρησιμοποιήσαμε για την δημιουργία του ενδιαμέσου κώδικα, αναλυτικά.

Τέλος θα προβάλουμε τα σημεία εκείνα που καλούμε τις βοηθητικές υπορουτίνες μέσα στο συντακτικό αναλυτή.

3.2 «Βοηθητικές Υπορουτίνες» ενδιάμεσου κώδικα

Προτού σχολιάσουμε των κώδικα για τις βοηθητικές υπορουτίνες πρέπει να αναφέρουμε ότι για την δημιουργία τους ορίσαμε αρχικά τα εξής:

- i. Τον πίνακα “quads” που θα αποθηκεύσουμε τις τετράδες του ενδιάμεσου κώδικα
- ii. Τον πίνακα “temp_var” που θα αποθηκεύσουμε τις προσωρινές μεταβλητές που θα χρησιμοποιήσουμε.
- iii. Την μεταβλητή “quad_num” που κρατάει τον αύξοντα αριθμό (κατά μία μονάδα) της τρέχουσας τετράδας.
- iv. Την μεταβλητή “temp_num” που κρατάει τον αριθμό της τρέχουσας προσωρινής μεταβλητής.

```
quads = [] ##Array that holds the quads of the intermediate code
temp_vars = [] ##Array that holds to name of the temp variables used
quad_num = -1
temp_num = -1
```

3.2.1

Κατασκευάσαμε λοιπόν 7 βιοηθητικές υπορουτίνες σύμφωνα με την καθοδήγηση των διαφανειών του μαθήματος «Ενδιάμεσος Κώδικας.pdf».

i. nextquad()

- επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί

```
def next_quad(): #returns the number of the next quad
    global quad_num
    return quad_num+1
```

2. genquad(op, x, y, z)

- δημιουργεί την επόμενη τετράδα (op, x, y, z)

```
def gen_quad(op,x,y,z): ##generates a quad
    global quad_num
    global quads
    quad_num +=1
    quads.append([quad_num,op,x,y,z])
```

3. newtemp()

- δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή
- οι προσωρινές μεταβλητές είναι της μορφής {T_1, T_2, T_3 ...}

```
def new_temp_var(): #returns a new temp variable
    global temp_num
    global temp_vars
    temp_num +=1
    temp = "T_" + str(temp_num)
    temp_vars.append(temp)
    return temp
```

4. emptylist()

- δημιουργεί μία κενή λίστα ετικετών τετράδων

```
def empty_list(): #returns an empty quads list
    new_list = []
    return new_list
```

5. makelist(x)

- δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x

```
def make_list(x): #returns a quads list containing x
    new_list = []
    new_list.append(x)
    return new_list
```

6. merge(list1, list2)

- δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών list1, list2

```
def merge_lists(l1,l2): #merges two quads lists into one
    merged_l = l1+l2
    return merged_l
```

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

```
def backpatch(param_list, z): #fills the label z of a quad
    global quads
    for quad in quads:
        if quad[0] in param_list:
            quad[4]=z
```

3.3 Κατηγοριοποίηση Τετράδων

Μετά την δημιουργία των βοηθητικών συναρτήσεων σειρά έχει ο σχολιασμός των ιδίων των ‘quads’ του ενδιάμεσου κώδικα. Ο ενδιάμεσος κώδικας αποτελείται από μια σειρά αριθμημένων (μεταβλητή “quad_num”) τετράδων, όπου αποτελούνται από έναν τελεστή και τρία τελούμενα. Οι τετράδες αυτές κατηγοριοποιούνται ως εξής:

3.3.1 « ΑΡΙΘΜΗΤΙΚΕΣ ΠΡΑΞΕΙΣ »

op , x , y , z

- Το op αντιπροσωπεύει κάποια αριθμητική πράξη όπως: +,-,*,/

- Τα τελούμενα x, y μπορεί να είναι είτε μεταβλητές είτε αριθμητικές σταθερές.
- Το τελούμενο z μπορεί να είναι όνομα μεταβλητής.
- Παράδειγμα: τα τελούμενα x και y εκτελούν την πράξη της πρόσθεσης (δηλαδή ο τελεστής op είναι ίσος με '+') και το αποτέλεσμα της πρόσθεσης ανατίθεται στο τελούμενο z .

3.3.2 « ΕΚΧΩΡΗΣΕΙΣ »

= , x , _ , z

- Ο συμβολισμός '=' έχει το νόημα της εικώρησης.
- Το τελούμενο x μπορεί να είναι μεταβλητή ή κάποια σταθερά αριθμητική.
- το τελούμενο z μπορεί να είναι κάποια μεταβλητή.
- Παράδειγμα: η τιμή του x εκχωρείται στη μεταβλητή z

3.3.3 « ΑΛΜΑΤΑ »

jump , _ , _ , z

- Το τελούμενο z μπορεί να είναι ο μετρητής των τετράδων "quad_num"
- Παράδειγμα: `jump _ _ 27` άλμα στο quad με αρίθμηση 27

relop , x , y , z

- Το `relop` μπορεί να είναι: ($=, >, <, <>, >=, <=$)
- Το τελούμενο x, y μπορεί να είναι μεταβλητή ή κάποια αριθμητική σταθερά.
- Το τελούμενο z επίσης.
- Παράδειγμα: με `relop` να είναι ' $>$ ' αν ισχύει $x > y$ τότε πήγαινε στο αντίστοιχο quad που υποδεικνύει η τιμή που έχει αποθηκευμένη το τελούμενο z .

3.3.4 « ΑΡΧΗ ΚΑΙ ΤΕΛΟΣ ΕΝΟΤΗΤΑΣ »

begin_block, name, _ , _

- Έναρξη προγράμματος, υπό-προγράμματος με ονομασία 'name'

end_block, name, _, _

- Λήξη προγράμματος, υπό-προγράμματος με ονομασία 'name'

halt, _, _, _

- Τερματισμός όλου του προγράμματος

3.3.5 « ΣΥΝΑΡΤΗΣΕΙΣ- ΔΙΑΔΙΚΑΣΙΕΣ »

par, x, m, _

- όπου x παράμετρος συνάρτησης και m ο τρόπος μετάδοσης
- CV : μετάδοση με τιμή
- EF: μετάδοση με αναφορά
- RET: επιστροφή τιμής συνάρτησης

call, name, _, _

- κλήση συνάρτησης name

ret, x, _, _

- επιστροφή τιμής συνάρτησης

3.4 Δομές Ενδιάμεσου Κώδικα

Αφού κατηγοριοποιήσαμε τις τετράδες ας δούμε τώρα και τους κανόνες με τους οποίους τις εφαρμόσαμε στον κώδικα του συντακτικού αναλυτή “syntax()”:

- i. Αρχή και Τέλος Block
- ii. Αριθμητικές Παραστάσεις
- iii. Λογικές Παραστάσεις
 - a. OR
 - b. AND
 - c. Not & RELOP
- iv. Κλήση Υποπρογραμμάτων
- v. Εντολή return

- vi. Εκχώρηση
- vii. Δομή while
- viii. Δομή if
- ix. Είσοδος -Έξοδος

Προκειμένου να αναπαραστήσουμε τις παραπάνω δομές ενδιάμεσου κώδικα στον δικό μας κώδικα, ακολουθησε τα εξής βήματα ανά δομή:

3.4.1 « ΑΡΧΗ ΚΑΙ ΤΕΛΟΣ BLOCK»

Εισήλθαμε στις συναρτήσεις “def_main_function”, “def_function” και “call_main_part”

Δημιουργήσαμε τις παρακάτω τετράδες:

- i. gen_quad("begin_block", όνομα_συνάρτησης, "_", "_")
- ii. gen_quad("end_block", όνομα_συνάρτησης, "_", "_")
- iii. gen_quad("halt", "_", "_", "_")

```

def def_main_function(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global line_number
    global token
    global ACCESS

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        if (ACCESS.family == "Identifier"):
            f_id = ACCESS.recognized_string
            gen_quad("begin_block", f_id, "_", "_")
            temp_offset=offset
            main_function=Entity(f_id, "main_function", temp_offset)
            offset+=1
            if exists(nesting,main_function)==True:
                raise ValueError(f"Function {f_id} already declared at this scope! Line(line_number)")
            main_function.setStart_quad(quad_num)
            functions_list.append(main_function)
            scope.add_entity(main_function)
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '('):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == ')'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "Delimiter"
            and ACCESS.recognized_string == ':'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '#{'):
            nesting+=1
            main_function_scope=Scope(nesting)
            temp_scope.append(main_function_scope)

            ACCESS = lex()
            line_number = ACCESS.line_number
            declarations(main_function_scope)
            while (ACCESS.family == "Keyword"
                  and ACCESS.recognized_string == 'def'):
                def_function(main_function_scope)
                statements(main_function_scope)

            if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '#'):
                gen_quad("end_block", f_id, "_", "_")
                frame_length=(offset-4)-12

def def_function(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting, quad_num
    global line_number
    global token
    global ACCESS

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        ACCESS = lex()
        line_number = ACCESS.line_number

        if (ACCESS.family == "Identifier"):
            f_id = ACCESS.recognized_string
            gen_quad("begin_block", f_id, "_", "_")
            temp_offset=offset
            function=Entity(f_id, "function", temp_offset)
            offset+=12
            if exists(nesting,function)==True:
                raise ValueError(f"Function {f_id} already declared at this scope! Line(line_number)")
            function.setStart_quad(quad_num)
            functions_list.append(function)
            scope.add_entity(function)
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '('):
            ACCESS = lex()
            line_number = ACCESS.line_number
            nesting+=1
            function_scope=Scope(nesting)
            temp_scope.append(function_scope)
            par_list_id.list("par",function_scope)
            function.setPar_list(par_list)
            if (ACCESS.family == "GroupSymbol"
                and ACCESS.recognized_string == ')'):
                ACCESS = lex()
                line_number = ACCESS.line_number

        if (ACCESS.family == "Delimiter"
            and ACCESS.recognized_string == ':'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '#{'):
            ACCESS = lex()
            line_number = ACCESS.line_number
            declarations(function_scope)
            while (ACCESS.family == "Keyword"
                  and ACCESS.recognized_string == 'def'):
                def_function(function_scope)
                statements(function_scope)

            if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '#'):
                gen_quad("end_block", f_id, "_", "_")

```

Στην περίπτωση που το block στο οποίο βρισκόμαστε είναι αυτό της “`__main__`” προσθέτουμε και την τετράδα “`halt`” προκειμένου να δηλώσουμε τερματισμό του προγράμματος, όπως φαίνεται και στην παρακάτω εικόνα:

```

def call_main_part(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting, quad_num
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'if'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Keyword"
        and ACCESS.recognized_string == '__name__'):
        gen_quad("begin_block", "__main__", "___", "___")
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "RelOperator"
        and ACCESS.recognized_string == '=='):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ''):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Keyword"
        and ACCESS.recognized_string == '__main__'):

        main_function=Entity("__main__", "main", 0)
        offset+=4
        main_function.setStart_quad(quad_num)
        scope.add_entity(main_function)
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ''):

        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ':'):

        ACCESS = lex()
        line_number = ACCESS.line_number

    main_function.call(scope)
    while (ACCESS.family == "Identifier"):
        main_function_call(scope)
        gen_quad("halt", "___", "___", "___")
        gen_quad("end_block", "__main__", "___", "___")

```

3.4.2 « ΑΡΙΘΜΗΤΙΚΕΣ ΠΑΡΑΣΤΑΣΕΙΣ»

Εισήλθαμε στις συναρτήσεις “expression” και “term”

Δημιουργήσαμε τις παρακάτω τετράδες:

- i. `gen_quad(add_opper,temp_term1,temp_term2,temp_var)`
 - ii. `gen_quad(mul_opper,temp_factor1,temp_factor2,temp_var)`
-
-
1. Πιο αναλυτικά για την συνάρτηση “expression” που είναι υπεύθυνη για την τέλεση της πρόσθεσης ακολουθήσαμε την εξής διαδικασία:
 2. Δημιουργήσαμε νέα προσωρινή μεταβλητή “`temp_var`” που διατηρεί το τρέχων αποτέλεσμα χρησιμοποιώντας την βοηθητική υπορουτίνα “`new_temp_var()`”:
 - `temp_var = new_temp_var()`
 3. Επίσης δημιουργήσαμε 2 τοπικές μεταβλητές ως τελούμενα της αριθμητικής παράστασης:

- temp_term1=term(scope)
 - temp_term2=term(scope)
4. Αποθηκεύσαμε στην μεταβλητή “add_opper” τον τελεστή της αριθμητικής πράξης που θα εκτελεστεί, όπου για την παρούσα συνάρτηση “expression” μπορεί να είναι ‘+’ ή ‘-’:
- add_opper=ACCESS.recognized_string
5. Δημιουργήσαμε την πρώτη τετράδα που προσθέτει το μέχρι στιγμής αποτέλεσμα στο νέο “temp_term2”
- gen_quad(add_opper,temp_term1,temp_term2,temp_var)
6. Το μέχρι στιγμής αποτέλεσμα τοποθετείται στην “temp_term1” ώστε να χρησιμοποιηθεί αν υπάρχει επόμενο “temp_term2”
- temp_term1=temp_var
7. Όταν δεν υπάρχει άλλο “temp_term2” το αποτέλεσμα είναι στο temp_term1
- return temp_term1

```

def expression(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    optional_sign()
    temp_term1=term(scope)

    while (ACCESS.family == "AddOperator" and ACCESS.recognized_string == '+'
           or ACCESS.family == "AddOperator"
           and ACCESS.recognized_string == '-'):
        add_opper=ACCESS.recognized_string
        ADD_OP()
        temp_term2=term(scope)
        temp_var=new_temp_var()
        var=Entity(temp_var,"temp_var",offset)
        offset+=4
        scope.add_entity(var)
        gen_quad(add_opper,temp_term1,temp_term2,temp_var)
        temp_term1=temp_var
    return temp_term1

```

Με την ανάλογη λογική που αναδείξαμε παραπάνω συμπληρώσαμε και την συνάρτηση “term” για τους τελεστές '*' ή '/*'

```

-- 
def term(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    temp_factor1=factor(scope)

    while (ACCESS.family == "MulOperator" and ACCESS.recognized_string == '**'
           or ACCESS.family == "MulOperator"
           and ACCESS.recognized_string == '//'):
        mul_opper=ACCESS.recognized_string
        MUL_OP()
        temp_factor2=factor(scope)
        temp_var=new_temp_var()
        var=Entity(temp_var,"temp_var",offset)
        offset+=4
        scope.add_entity(var)
        gen_quad(mul_opper,temp_factor1,temp_factor2,temp_var)
        temp_factor1=temp_var
    return temp_factor1

```

3.4.3 «ΛΟΓΙΚΕΣ ΠΑΡΑΣΤΑΣΕΙΣ»

Εισήλθαμε στις συναρτήσεις “condition”, “bool_term” και “bool_factor” του συντακτικού αναλυτή. Κάθε μια από τις προαναφερθέντες συναρτήσεις είναι υπεύθυνη για την υλοποίηση μίας λογικής πράξης (OR, AND, NOT & RELOP) αντιστοίχως. Ας τις δούμε παρακάτω καλύτερα κάθε μία ξεχωριστά:

3.4.3.1 «OR»

Μεταφέρουμε τις τιμές από τη συνάντηση bool_term στις μεταβλητές t1_true και t1_false

`t1_true,t1_false=bool_term(scope)`

Συμπλήρωση όσον τετράδων μπορούν να συμπληρωθούν μέσα στον κανόνα, με την χρήση της βοηθητικής υπορουτίνας “backpatch()”

`backpatch(t1_false,next_quad())`

Μεταφέρουμε τις τιμές από τη συνάντηση bool_term στις μεταβλητές t2_true και t2_false

`t2_true,t2_false=bool_term(scope)`

Συσσώρευση στη λίστα t1_true των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε αληθή αποτίμηση λογικής παράστασης

```
t1_true=merge_lists(t1_true,t2_true)
```

Η λίστα t1_false περιέχει την τετράδα ή οποία αντιστοιχεί σε μια μη αληθή αποτίμηση της λογικής παράστασης

```
t1_false=t2_false
```

```
def condition(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    t1_true,t1_false=bool_term(scope)

    while (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'or'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        backpatch(t1_false,next_quad())
        t2_true,t2_false=bool_term(scope)
        t1_true=merge_lists(t1_true,t2_true)
        t1_false=t2_false
    return t1_true,t1_false
```

3.4.3.2 «AND»

Μεταφέρουμε τις τιμές από τη συνάντηση bool_factor στις μεταβλητές f1_true και f1_false

```
f1_true,f1_false=bool_factor(scope)
```

Συμπλήρωση όσον τετράδων μπορούν να συμπληρωθούν μέσα στον κανόνα, με την χρήση της βοηθητικής υπορουτίνας “backpatch()”

```
backpatch(f1_true,next_quad())
```

Μεταφέρουμε τις τιμές από τη συνάντηση bool_factor στις μεταβλητές f2_true και f2_false

```
f2_true,f2_false=bool_factor(scope)
```

Συσσώρευση στη λίστα f1_false των τετράδων που δεν μπορούν να συμπληρωθούν και αντιστοιχούν σε μη αληθή αποτίμηση λογικής παράστασης

```
f1_false=merge_lists(f1_false,f2_false)
```

Η λίστα f1_true περιέχει την τετράδα ή οποία αντιστοιχεί στην αληθή αποτίμηση της λογικής παράστασης

```
f1_true=f2_true
```

```

def bool_term(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    f1_true,f1_false=bool_factor(scope)

    while (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'and'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        backpatch(f1_true,next_quad())
        f2_true,f2_false=bool_factor(scope)
        f1_false=merge_lists(f1_false,f2_false)
        f1_true=f2_true
    return f1_true,f1_false

```

3.4.3.3 «NOT & RELOP»

Αρχικά ας σχολιάσουμε για την λογικό τελεστή 'not':

1. Μεταφέρουμε τις τιμές από τη συνάντηση condition() στις μεταβλητές bTrue και bFalse
 - bTrue,bFalse=condition(scope)
2. Αντιστρέφουμε τις τιμές των bTrue και bFalse ξαναγράφοντάς τελικά την παραπάνω έκφραση ως:
 - bFalse,bTrue=condition(scope)

```

def bool_factor(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'not'):
        ACCESS = lex(scope)
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '['):
        ACCESS = lex(scope)
        line_number = ACCESS.line_number

        bFalse,bTrue=condition(scope)

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ']'):
        ACCESS = lex()
        line_number = ACCESS.line_number

```

Ας αναφερθούμε τώρα και στην 'relop':

1. Μεταφέρουμε τις τιμές από τη συνάντηση expression() στις μεταβλητές exp1 και exp2, ενώ παράλληλα μεταφέρουμε τον τελεστή της εκάστοτε σύγκρισης καλώντας την REL_OP()

- `exp1=expression(scope)`
 - `rel_opper=ACCESS.recognized_string`
 - `REL_OP()`
 - `exp2=expression(scope)`
2. Δημιουργία μη συμπληρωμένης τετράδας και εισαγωγή στη λίστα μη συμπληρωμένων τετράδων για την αληθή αποτίμηση της relop.
- `bTrue=make_list(next_quad())`
 - `gen_quad(rel_opper,exp1,exp2,"_")`
3. Δημιουργία μη συμπληρωμένης τετράδας και εισαγωγή στη λίστα μη συμπληρωμένων τετράδων για την μη αληθή αποτίμηση της relop.
- `bFalse=make_list(next_quad())`
 - `gen_quad("jump","_","_","_")`

```

def bool_factor(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'not'):
        ACCESS = lex(scope)
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '['):
        ACCESS = lex(scope)
        line_number = ACCESS.line_number

        bTrue,bFalse=condition(scope)

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == ']'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        else:
            raise ValueError(f"Missing right ]', after bool_factor's condition. Line: {line_number}")
        else:
            raise ValueError(f"Missing right '[', after keyword 'not' in bool factor. Line: {line_number}"]

    elif (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '['):
        ACCESS = lex()
        line_number = ACCESS.line_number

        bTrue,bFalse=condition(scope)

        if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == ']'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        else:
            raise ValueError(f"Missing right ]', after bool_factor's condition. Line: {line_number}")
        else:

            exp1=expression(scope)
            rel_opper=ACCESS.recognized_string
            REL_OP()
            exp2=expression(scope)
            bTrue=make_list(next_quad())
            gen_quad(rel_opper,exp1,exp2,"_")
            bFalse=make_list(next_quad())
            gen_quad("jump","_","_","+")
    return bTrue,bFalse

```

3.4.4 « ΚΛΗΣΗ ΥΠΟΠΡΟΓΡΑΜΜΑΤΩΝ »

Εισήλθαμε στις συναρτήσεις “idtail” και “actual_par_list”

Δημιουργήσαμε τις παρακάτω τετράδες:

- i. gen_quad("par",temp_exp2,"CV","_")
- ii. gen_quad("call",caller,"_","_")
- iii. gen_quad("par",temp,"RET","_")

Πιο αναλυτικά για την συνάρτηση “idtail” ακολουθήσαμε τα παρακάτω βήματα :

1. Δημιουργήσαμε μια προσωρινή μεταβλητή temp χρησιμοποιώντας την βοηθητική υπορουτίνα ‘new_temp_var’ η οποιά αναπαρηστά την τιμή επιστροφής της συνάρτησης που θα κληθεί.
 - temp=new_temp_var()
2. Υλοποιήσαμε την παρακάτω τετράδα για να αναδείξουμε την συνάρτηση αυτή που θα κληθεί
 - gen_quad("call",caller,"_","_")
3. Υλοποιήσαμε την παρακάτω τετράδα για να αναδείξουμε την μεταβλητή στην οποιά θα αποθηκευτεί η τιμή που επιστρέφει η συνάρτηση που κλήθηκε
 - gen_quad("par",temp,"RET","_")

```
def idtail(caller,scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

        actual_par_list(scope)

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == ')'):
        temp=new_temp_var()
        tempVar=Entity(temp,"par",offset)
        tempVar.setPar_mode("RET")
        scope.add_entity(tempVar)
        offset+=4
        gen_quad("call",caller,"_","_")
        gen_quad("par",temp,"RET","_")
        ACCESS = lex()
        line_number = ACCESS.line_number
        return temp
    return caller
```

Μέσα συνάρτηση “actual_par_list” δημιουργήσαμε quads για κάθε παράμετρο της εκάστοτε κληθέντας συνάρτησης ως εξής

- gen_quad("par","όνομα παραμέτρου,"CV","_")

```
def actual_par_list(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Number"
        or ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('
        or ACCESS.family == "Identifier"):

        temp_expl=expression(scope)
        gen_quad("par",temp_expl,"CV","_")
        while (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ','):
            ACCESS = lex()
            line_number = ACCESS.line_number

        temp_exp2=expression(scope)
        gen_quad("par",temp_exp2,"CV","_")
```

3.4.5 « ΕΝΤΟΛΗ RETURN»

Εισήλθαμε στην συνάρτηση “return_stat”

Μεταφέρουμε την τιμή της έκφρασης που θα επιστραφεί στην μεταβλητή exp κάνοντας χρήση της συνάρτησης expression

exp=expression(scope)

Υλοποιούμε την τετράδα επιστροφής

gen_quad("retv",exp,"_","_")

```

def return_stat(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'return'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

    exp=expression(scope)

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ';'):
        gen_quad("retv",exp,"_","_")
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing semicolon ';' after expression in return statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing right parenthesis ')', not closed in return statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened in return statement. Line: {line_number}")
else:
    raise ValueError(f"Missing the keyword 'return' in return statement. Line: {line_number}")

```

3.4.6 « ΕΚΧΩΡΗΣΗ»

Εισήλθαμε στην συνάρτηση **“assignment_stat”** και ακολουθήσαμε την ιδιά μεθοδολογία που περιγράψαμε παραπάνω στην υποενότητα της αναφοράς μας **«3.4.5 ΕΝΤΟΛΗ RETURN»**.

```

else:
    exp=expression(scope)
    if (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ';'):
        gen_quad("=",exp,"_","var")
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing semicolon ';' after expression in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing assignment symbol '=' after Identifier token, in assignment statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing Identifier token in assignment statement. Line: {line_number}")

```

3.4.7 « ΔΟΜΗ WHILE»

Εισήλθαμε στην συνάρτηση **while_stat** και ακολουθήσαμε τα εξής βήματα:

1. Ξεκινήσαμε παίρνοντας τις τιμές των “bTrue”, “bFalse” από την κλήση της συνάρτησης **condition()**
 - **bTrue,bFalse=condition(scope)**
2. Συμπλήρωση των τετράδων που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, δηλαδή για την “bTrue”
 - **backpatch(bTrue,next_quad())**

3. Μετάβαση στην αρχή της συνθήκης ώστε να ξαναγίνει έλεγχος.

- `gen_quad("jump","_","_",start_of_while)`
4. Συμπλήρωση των τετραδίων που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, (με το `false` έξω από τη δομή) δηλαδή την “`bFalse`”
- `backpatch(bFalse,next_quad())`

```
def while_stat(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'while'):
        ACCESS = lex()
        line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
            start_of_while = next_quad()
            ACCESS = lex()
            line_number = ACCESS.line_number

            bTrue,bFalse=condition(scope)

            if (ACCESS.family == "GroupSymbol"
                and ACCESS.recognized_string == ')'):
                ACCESS = lex()
                line_number = ACCESS.line_number

            if (ACCESS.family == "Delimiter"
                and ACCESS.recognized_string == ':'):
                ACCESS = lex()
                line_number = ACCESS.line_number

            if (ACCESS.family == "GroupSymbol"
                and ACCESS.recognized_string == '#('):
                ACCESS = lex()
                line_number = ACCESS.line_number
                backpatch(bTrue,next_quad())
                statements(scope)
                gen_quad("jump","_","_",start_of_while)
                backpatch(bFalse,next_quad())
                if (ACCESS.family == "GroupSymbol"
                    and ACCESS.recognized_string == '#)'):
                    ACCESS = lex()
                    line_number = ACCESS.line_number
                else:
                    raise ValueError(f"Missing the two block's closing symbols after 'while' statement. Line: {line_number}")
            else:
                backpatch(bTrue,next_quad())
                statement(scope)
                gen_quad("jump","_","_",start_of_while)
                backpatch(bFalse,next_quad())
            else:
                raise ValueError(f"Missing ')' after 'while' statement. Line: {line_number}")
        else:
            raise ValueError(f"Missing right parenthesis ')', not closed after 'while' statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened before 'while' staement. Line: {line_number}")
else:
    raise ValueError(f"Missing the keyword 'while' in while_statement. Line: {line_number}")
```

3.4.8 « ΔΟΜΗ IF »

Εισήλθαμε στην συνάρτηση “`if_stat()`”

Ξεκινήσαμε παίρνοντας τις τιμές των “`bTrue`”, “`bFalse`” από την κλήση της συνάρτησης `condition()`

Συμπλήρωση των τετράδων που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, για το `if`

Εξασφαλίζουμε ότι εάν εκτελεστούν οι εντολές του if δεν θα εκτελεστούν στη συνέχεια οι εντολές του else.

Συμπλήρωση των μέτρων που έχουν μείνει ασυμπλήρωτες και γνωρίζουμε τώρα ότι πρέπει να συμπληρωθούν με την επόμενη τετράδα, για το else

Εξασφαλίζουμε ότι θα εκτελεστούν οι εντολές του else.

```
def if_stat(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'if'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

    bTrue,bFlase=condition(scope)

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "Delimiter"
        and ACCESS.recognized_string == ':'):
        ACCESS = lex()
        line_number = ACCESS.line_number

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '#{'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        backpatch(bTrue,next_quad())
        statements(scope)
        temp=make_list(next_quad())
        gen_quad("jump","_","_","_")
        backpatch(bFlase,next_quad())

    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '#}'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing the two block's closing symbols after 'if' statement. Line: {line_number}")
    else:
        backpatch(bTrue,next_quad())
        statement(scope)
        temp=make_list(next_quad())
        gen_quad("jump","_","_","_")
        backpatch(bFlase,next_quad())
        backpatch(temp,next_quad())
```

3.4.9 « ΕΙΣΟΔΟΣ – ΕΞΟΔΟΣ »

Εισήλθαμε στην συνάρτηση print_stat(), assignment_stat)() και ακολουθήσαμε την ιδιά μεθοδολογία που περιγράψαμε παραπάνω στην υποενότητα της αναφοράς μας «**3.4.5 ΕΝΤΟΛΗ RETURN**».

Για την είσοδο:

```

def assignment_stat(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global line_number
    if (ACCESS.family == "Identifier"):
        var=ACCESS.recognized_string
        if not_exists(var)==False:
            raise ValueError(f"Trying to initialize variable {var} that was never declared. Line {line_number}")
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "Assignment" and ACCESS.recognized_string == '='):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'int'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "Keyword"
        and ACCESS.recognized_string == 'input'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ';'):
        gen_quad("=", "input", "_", var)
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:

```

Ενώ για την έξοδο:

```

def print_stat(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number
    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'print'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number
        exp=expression(scope)
    if (ACCESS.family == "GroupSymbol"
        and ACCESS.recognized_string == ')'):
        ACCESS = lex()
        line_number = ACCESS.line_number
    if (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ';'):
        gen_quad("out",exp,"_","_")
        ACCESS = lex()
        line_number = ACCESS.line_number
    else:
        raise ValueError(f"Missing semicolon ';' after expression in print statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing right parenthesis ')', not closed in print statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing left parenthesis '(', not opened in print statement. Line: {line_number}")
    else:
        raise ValueError(f"Missing the keyword 'print' in print statement. Line: {line_number}")

```

3.5 Αρχείο « .init »

Τέλος προκειμένου να εκτυπώσουμε τα αποτελέσματα (quads) του ενδιάμεσου κώδικα πίσω στον χρήστη μέσα σε αρχείο με κατάληξη .int και όνομα ίδιο με αυτό του αρχείου εισόδου CutePy, δημιουργήσαμε ακόμα την παρακάτω συνάρτηση:

```

def int_file(): #creates file that contains the quads
    file=cutePy_file_name.split(".cpy")
    file=file[0]+".int"
    myfile=open(file,'w')
    for quad in quads:
        myfile.write(str(quad[0])+": "+str(quad[1])+" "+str(quad[2])+" "+str(quad[3])+" "+str(quad[4]))
        myfile.write("\n")
    myfile.close()

```

4. Ενότητα: “Πίνακας Συμβόλων”

4.1 Περίληψη Ενότητας

Στην ενότητα αυτή θα αναφέρουμε όλες τις κλάσεις, τις συναρτήσεις, και τις λίστες που υλοποιούσαμε προκειμένου να δημιουργήσουμε τον πίνακα συμβολών. Υστέρα θα κατηγοριοποιήσουμε τις μεθοδολογίες που κατασκευάσαμε προκειμένου να εντάξουμε στον υπάρχων κώδικα του συντακτικού αναλυτή και ενδιάμεσου κώδικα, τον «Πίνακα Συμβολών». Τέλος, θα αναφέρουμε και στο αρχείο που παράγουμε μετά την εκτέλεση του προγράμματος μας, που αφορά αποκλειστικά τον πίνακα συμβόλων.

4.2 Κλάσεις, Συναρτήσεις και Λίστες

Προκειμένου να υλοποιήσουμε τον Πίνακα Συμβολών στο πρόγραμμα μας δημιουργήσαμε τα παρακάτω βοηθητικά εργαλεία:

4.2.1 ΚΛΑΣΕΙΣ

Δημιουργήσαμε δύο κλάσεις την Entity που αναπαριστά μία οντότητα του προγράμματος και την scope που αναπαριστά ένα scope.

```

class Entity: ##This class represents program entities such as functions, variables, parameters and temp variables

    def __init__(self, name, type, offset):
        self.name=str(name)
        self.type=str(type)
        self.offset=int(offset)
        self.start_quad=0
        self.par_list=[]
        self.frame_length=0
        self.par_mode=""

    def setStart_quad(self, x):
        self.start_quad=x

    def getStart_quad(self):
        return self.start_quad

    def setPar_list(self, x):
        self.par_list=x

    def getPar_list(self):
        return self.par_list

    def setFrame_length(self, x):
        self.frame_length=x

    def getFrame_length(self):
        return self.frame_length

    def setPar_mode(self, x):
        self.par_mode=x

    def getPar_mode(self):
        return self.par_mode


class Scope:
    def __init__(self, nesting_level):
        self.entity_list=[]
        self.nesting_level=nesting_level

    def add_entity(self, e):
        self.entity_list.append(e)

```

Επίσης, οι δυο κλάσεις που παρουσιάσαμε παραπάνω εμπειρέχουν κάποιες συναρτήσεις. Οι συναρτήσεις αυτές κατά κύριο λόγο είναι setters και getters για την εκχώρηση και διάβασμα των τιμών των πεδίων των συναρτήσεων. Οι συναρτήσεις οι οποίες παίζουν ρόλο στην υλοποίηση μας και αφορούν την σημασιολογική ανάλυση θα παρουσιαστούν παρακάτω.

4.2.2 ΛΙΣΤΕΣ

- functions_list = λίστα με όλες τις συναρτήσεις του προγράμματος
- variables_list = λίστα με όλες τις μεταβλητές του προγράμματος
- temp_scope = Αυτό είναι ουσιαστικά μία στοίβα με scopes
- final_scope = Εδώ αποθηκεύεται ο τελικός πίνακας συμβόλων

- offset = Αυτό είναι το offset θα πεις ότι είδαμε από τις σημειώσεις ότι πρέπει να ξεκινάει από το 12
- nesting = και αυτό είναι το επίπεδο φωλιάσματος

```
functions_list=[] ##Array that holds function entities declared in the programm
variables_list=[] ##Array that holds user declared variable entities in the programm
temp_scope=[] ##Stack that holds scopes
final_scope=[] ##Array for all the scopes used in the programm
offset = 12
nesting = 0
```

4.2.3 ΣΥΝΑΡΤΗΣΕΙΣ

- not_exists = ελέγχει αν μία μεταβλητή ή συνάρτηση έχει οριστεί πριν χρησιμοποιηθεί (κοιτάει στο τρέχον και σε παραπάνω scopes)
- exists = ελέγχει αν μια μεταβλητή ή συνάρτηση έχει οριστεί παραπάνω από 1 φορά στο ίδιο scope

```
def not_exists(x): #this function is used to check if an entity is declared before used
    global temp_scope
    for i in temp_scope:
        for j in i.entity_list:
            if j.name==x:
                return True
    return False

def exists(nesting, x): #this function is used to check if an entity is already declared in a scope
    global temp_scope
    for i in temp_scope:
        if i.nesting_level==nesting:
            for j in i.entity_list:
                if j.name==x.name:
                    return True
    return False
```

4.3 Κατηγοριοποίηση Μεθοδολογίας Ανά Συναρτήσεις

4.3.1 DEF_MAIN_FUNCTION

Για την υλοποίηση των λειτουργιών του πίνακα συμβόλων στην συνάρτηση def_main_function ακολουθούμε τα εξής βήματα:

1. Κρατάμε σε μία μεταβλητή (`temp_offset`) το offset της συνάρτησης scope στο οποίο ορίζεται.
2. Δημιουργούμε ένα καινούργιο Entity για την συνάρτηση το οποίο ελέγχουμε αν έχει οριστεί πάλι στο τρέχον scope
3. Ενημερώνουμε τα πεδίο `start_quad` του Entity της συνάρτησης
4. Προσθέτουμε το Entity στο τρέχον scope καθώς και στον πίνακα που κρατάει τις συναρτήσεις που υπάρχουν στο πρόγραμμα
5. Μόλις μπούμε στο μπλοκ της συνάρτησης αυξάνουμε κατά ένα το nesting level και δημιουργούμε ένα καινούργιο Scope για την συνάρτηση το οποίο και προσθέτουμε στο πίνακα `temp_scope`
6. Όταν βγούμε από το μπλοκ της συνάρτησης μειώνουμε κατά ένα το nesting level ενημερώνουμε το πεδίο `frame_length` του Entity της συνάρτησης προσθέτουμε το Scope της συνάρτηση στο πίνακα `final_scope` και το αφαιρούμε από τον πίνακα `temp_scope`

Παρουσιάζεται το κομμάτι κώδικα της συνάρτησης `main_function` όπου υλοποιούνται οι παραπάνω λειτουργίες:

```

def def_main_function(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global line_number
    global token
    global ACCESS

    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        if (ACCESS.family == "Identifier"):
            f_id = ACCESS.recognized_string
            gen_quad("begin_block", f_id, "_", "_")
            temp_offset=offset
            main_function=Entity(f_id, "main_function", temp_offset)
            offset+=12
            if exists(nesting,main_function)==True:
                raise ValueError(f"Function {f_id} already declared at this scope! Line({line_number})")
            main_function.setStart_quad(quad_num)
            functions_list.append(main_function)
            scope.add_entity(main_function)
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '('):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == ')'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "Delimiter"
            and ACCESS.recognized_string == ':'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '#{'):
            nesting+=1
            main_function_scope=Scope(nesting)
            temp_scope.append(main_function_scope)

        ACCESS = lex()
        line_number = ACCESS.line_number
        declarations(main_function_scope)
        while (ACCESS.family == "Keyword"
               and ACCESS.recognized_string == 'def'):
            def_function(main_function_scope)
            statements(main_function_scope)

        if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '#}'):
            gen_quad("end_block", f_id, "_", "_")
            frame_length=(offset-4)-12
            offset=temp_offset
            main_function.setFrame_length(frame_length)
            final_scope.append(main_function_scope)
            temp_scope.pop()
            nesting-=1
            ACCESS = lex()
            line_number = ACCESS.line_number

```

4.3.2 DEF_FUNCTION

Για την υλοποίηση των λειτουργιών του πίνακα συμβόλων στην συνάρτηση function ακολουθούμε τα εξής βήματα:

1. Κρατάμε σε μία μεταβλητή (temp_offset) το offset της συνάρτησης scope στο οποίο ορίζεται.
2. Δημιουργούμε ένα καινούργιο Entity για την συνάρτηση το οποίο ελέγχουμε αν έχει οριστεί πάλι στο τρέχον scope
3. Προσθέτουμε το Entity στο τρέχον scope καθώς και στον πίνακα που κρατάει τις συναρτήσεις που υπάρχουν στα πρόγραμμα
4. Ενημερώνουμε τα πεδία start_quad και par_list του Entity της συνάρτησης
5. Μόλις μπούμε στο μπλοκ της συνάρτησης αυξάνουμε κατά ένα το nesting level και δημιουργούμε ένα καινούργιο Scope για την συνάρτηση το οποίο και προσθέτουμε στο πίνακα temp_scope
6. Όταν βγούμε από το μπλοκ της συνάρτησης μειώνουμε κατά ένα το nesting level
ενημερώνουμε το πεδίο frame_length του Entity της συνάρτησης προσθέτουμε το Scope της συνάρτηση στο πίνακα final_scope και το αφαιρούμε από τον πίνακα temp_scope

Παρουσιάζεται το κομμάτι κώδικα της συνάρτησης function όπου υλοποιούνται οι παραπάνω λειτουργίες:

```

def def_function(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting, quad_num
    global line_number
    global token
    global ACCESS
    if (ACCESS.family == "Keyword" and ACCESS.recognized_string == 'def'):
        ACCESS = lex()
        line_number = ACCESS.line_number
        if (ACCESS.family == "Identifier"):
            f_id = ACCESS.recognized_string
            gen_quad("begin_block",f_id,"_","_")
            temp_offset=offset
            function=Entity(f_id,"function",temp_offset)
            offset+=12
            if exists(nesting,function)==True:
                raise ValueError(f"Function {f_id} already declared at this scope! Line{line_number}")
            function.setStart_quad(quad_num)
            functions_list.append(function)
            scope.add_entity(function)
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '('):
            ACCESS = lex()
            line_number = ACCESS.line_number
            nesting+=1
            function_scope=Scope(nesting)
            temp_scope.append(function_scope)
            par_list=id_list("par",function_scope)
            function.setPar_list(par_list)
            if (ACCESS.family == "GroupSymbol"
                and ACCESS.recognized_string == ')'):
                ACCESS = lex()
                line_number = ACCESS.line_number

        if (ACCESS.family == "Delimiter"
            and ACCESS.recognized_string == ':'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        if (ACCESS.family == "GroupSymbol"
            and ACCESS.recognized_string == '#{'):
            ACCESS = lex()
            line_number = ACCESS.line_number

        declarations(function_scope)
        while (ACCESS.family == "Keyword"
            and ACCESS.recognized_string == 'def'):
            def_function(function_scope)
            statements(function_scope)

        if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '#'):
            gen_quad("end_block",f_id,"_","_")
            frame_length=(offset-4)-12
            offset=temp_offset+4
            function setFrame_length(frame_length)
            final_scope.append(function_scope)
            temp_scope.pop()
            nesting-=1
            ACCESS = lex()
            line_number = ACCESS.line_number

```

4.3.3 ID_LIST

Για την υλοποίηση των λειτουργιών του πίνακα συμβόλων στην συνάρτηση function ακολουθούμε τα εξής βήματα:

1. Αρχικά στην συνάρτηση id_list προσθέσαμε ένα ακόμα όρισμα το caller. Ανάλογα με την τιμή αυτού του ορίσματος διαπιστώνουμε αν τα στοιχεία της id_list είναι μεταβλητές (αν η τιμή του caller είναι “decl”) ή παράμετροι συνάρτησης (αν η τιμή του caller είναι “par”).
2. Στην περίπτωση που τα στοιχεία της id_list είναι μεταβλητές:
 - a. Δημιουργούμε ένα καινούργιο Entity για την μεταβλητή
 - b. Αυξάνουμε κατά 4 το offset του τρέχοντος Scope
 - c. Προσθέτουμε την μεταβλητή στον πίνακα που κρατάει τις μεταβλητές του προγράμματος
 - d. Ελέγχουμε αν η μεταβλητή αυτή έχει ορισθεί πάλι στο τρέχον scope και αν αυτό ισχύει τυπώνουμε αντίστοιχο μήνυμα λάθους
 - e. Προσθέτουμε το Entity της μεταβλητής στο τρέχον scope
3. Στην περίπτωση που τα στοιχεία της id_list είναι παράμετροι:
 - a. Δημιουργούμε ένα καινούργιο Entity για την παράμετρο
 - b. Αυξάνουμε κατά 4 το offset του τρέχοντος Scope
 - c. Ενημερώνουμε την τιμή του πεδίου par_mode του Entity που κρατάει τον τρόπο περάσματος της παραμέτρου
 - d. Προσθέτουμε το Entity της παραμέτρου στο τρέχον scope
4. Η διαδικασία αυτή επαναλαμβάνεται για όλα τα στοιχεία της id_list

Παρουσιάζεται το κομμάτι κώδικα της συνάρτησης id_list όπου υλοποιούνται οι παραπάνω λειτουργίες:

```

def id_list(caller,scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global line_number
    global ACCESS
    pars=[]

    if caller=="decl":
        id_type="var"
    elif caller == "par":
        id_type="par"

    if (ACCESS.family == "Identifier"):
        id=ACCESS.recognized_string
        var=Entity(id,id_type,offset)
        offset+=4
    if caller == "decl":
        variables_list.append(var)
        if exists(nesting,var)==True:
            raise ValueError(f"Variable {id} already exists in this scope {line_number}")
    if caller == "par":
        pars.append(var)
        var.setPar_mode("CV")
        scope.add_entity(var)
        ACCESS = lex()
        line_number = ACCESS.line_number

    while (ACCESS.family == "Delimiter" and ACCESS.recognized_string == ','):
        ACCESS = lex()
        line_number = ACCESS.line_number

        if (ACCESS.family == "Identifier"):
            id=ACCESS.recognized_string
            var=Entity(id,id_type,offset)
            offset+=4
        if caller == "decl":
            variables_list.append(var)
            if exists(nesting,var)==True:
                raise ValueError(f"Variable {id} already exists in this scope {line_number}")
        if caller == "par":
            pars.append(var)
            var.setPar_mode("CV")
            scope.add_entity(var)
            ACCESS = lex()
            line_number = ACCESS.line_number
        else:
            raise ValueError(f"Missing Identifier token after comma(','). Line: {line_number}")
    return pars

```

4.3.4 EXPRESSION KAI TERM

Στις συναρτήσεις αυτές δημιουργούνται προσωρινές μεταβλητές. Για να προσθέσουμε μία προσωρινή μεταβλητή στον πίνακα συμβόλων ακολουθόσαμε να εξής βήματα:

1. Δημιουργούμε ένα καινούργιο Entity για την προσωρινή μεταβλητή
2. Αυξάνουμε κατά 4 το offset του τρέχοντος Scope
3. Προσθέτουμε το Entity της προσωρινής μεταβλητής στο τρέχον scope

Παρουσιάζεται το κομμάτι κώδικα της συνάρτησης "όποια από τις δύο θες" όπου υλοποιούνται οι παραπάνω λειτουργίες:

```

def expression(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    optional_sign()
    temp_term1=term(scope)

    while (ACCESS.family == "AddOperator" and ACCESS.recognized_string == '+'
        or ACCESS.family == "AddOperator"
        and ACCESS.recognized_string == '-'):
        add_pper=ACCESS.recognized_string
        ADD_OP()
        temp_term2=term(scope)
        temp_var=new_temp_var()
        var=Entity(temp_var,"temp_var",offset)
        offset+=4
        scope.add_entity(var)
        gen_quad(add_pper,temp_term1,temp_term2,temp_var)
        temp_term1=temp_var
    return temp_term1

def term(scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    temp_factor1=factor(scope)

    while (ACCESS.family == "MulOperator" and ACCESS.recognized_string == '**'
        or ACCESS.family == "MulOperator"
        and ACCESS.recognized_string == '//'):
        mul_pper=ACCESS.recognized_string
        MUL_OP()
        temp_factor2=factor(scope)
        temp_var=new_temp_var()
        var=Entity(temp_var,"temp_var",offset)
        offset+=4
        scope.add_entity(var)
        gen_quad(mul_pper,temp_factor1,temp_factor2,temp_var)
        temp_factor1=temp_var
    return temp_factor1

```

4.3.5 ID_TAIL

Για την υλοποίηση των λειτουργιών του πίνακα συμβόλων στην συνάρτηση id_tail ή οποία δημιουργεί μία προσωρινή μεταβλητή που κρατάει την τιμή που επιστρέφει μια συνάρτηση ακολουθούμε τα εξής βήματα:

1. Δημιουργούμε ένα καινούργιο Entity για την προσωρινή μεταβλητή
2. Ενημερώνουμε την τιμή του πεδίου par_mode του Entity που κρατάει τον τρόπο περάσματος της παραμέτρου επιστροφής
3. Προσθέτουμε το Entity της προσωρινής μεταβλητής στο τρέχον scope
4. Αυξάνουμε κατά 4 το offset του τρέχοντος Scope

Παρουσιάζεται το κομμάτι κώδικα της συνάρτησης id_tail όπου υλοποιούνται οι παραπάνω λειτουργίες:

```
def idtail(caller,scope):
    global functions_list, variables_list, temp_scope, final_scope, offset, nesting
    global ACCESS
    global line_number

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == '('):
        ACCESS = lex()
        line_number = ACCESS.line_number

        actual_par_list(scope)

    if (ACCESS.family == "GroupSymbol" and ACCESS.recognized_string == ')'):
        temp=new_temp_var()
        tempVar=Entity(temp,"par",offset)
        tempVar.setPar_mode("RET")
        scope.add_entity(tempVar)
        offset+=4
        gen_quad("call",caller,"_","_")
        gen_quad("par",temp,"RET","_")
        ACCESS = lex()
        line_number = ACCESS.line_number
        return temp
    return caller
```

4.4 Αρχειο « .symb »

Κατασκευάσαμε επιπλέον μια συνάρτηση για την παραγωγή του αρχείου του πίνακα συμβόλων προκειμένου να είναι πιο οργανωμένη η υλοποίηση μας , όπως αυτή φαίνεται παρακάτω:

```
def symbol_file(): #creates file that contains the symbol table
    file=cutePy_file_name.split(".cpy")
    file=file[0] + ".symb"
    myfile=open(file,'w')
    for i in final_scope:
        nesting=i.nesting_level
        if(nesting==0):
            name="main"
        elif(nesting==1):
            func=functions_list.pop(len(functions_list)-2)
            name=func.name
        else:
            func=functions_list.pop()
            name=func.name
        myfile.write("Nesting Level: "+str(nesting)+" -- Scope of function "+name+"\n")
    for j in i.entity_list:
        line=" Entity : "+j.name+", offset: "+str(j.offset)+", type: "+j.type
        if(j.type=="main_function"):
            line=line+", next quad: "+str(j.start_quad)+", frame length: "+str(j.frame_length)
        elif(j.type=="function"):
            line=line+", next quad: "+str(j.start_quad)+", frame length: "+str(j.frame_length)+" parameters: "
            for par in j.par_list:
                line=line+" "+par.name
        elif(j.type=="par"):
            line=line+", parMode: "+j.par_mode
        myfile.write(line+"\n")
    myfile.close()
```

5. Ενότητα: “Τελικός Κώδικας”

5.1 Περίληψη Ενότητας

Για την παραγωγή του τελικού κώδικα δημιουργήσαμε δύο λίστες και πολλές συναρτήσεις. Αφού αναφέρουμε τις δυο λίστες που υλοποιήσαμε και την χρησιμότητα τους θα αναλύσουμε μιας προς μια τις συναρτήσεις που δημιουργήσαμε για τον τελικό κώδικα της γλώσσας μας. Τέλος θα δείξουμε την συνάρτηση που παράγει το αρχείο με το τελικό κώδικα με κατάληξη « .asm».

5.2 Λίστες

Ξεκινάμε με την πρώτη λίστα που συναντάμε στον κώδικα μας την:

- `asm_commands=[]`

Η λίστα/πίνακας αυτή, είναι η λίστα στην οποία θα αποθηκεύσουμε όλες τις εντολές που θα δημιουργήσουμε σε RISC-V, κατά την διαδικασία παραγωγής του τελικού κώδικα.

Η δεύτερη λίστα :

- `labels=[]`

Στην λίστα αυτή αποθηκεύουμε όλες τις ετικέτες στην οποίες μεταβαίνει μία εντολή άλματος. Οι ετικέτες αυτές παίρνονται από τις τετράδες του ενδιάμεσου κώδικα.

Επίσης πρέπει να υπενθυμίσουμε ακόμα δύο λίστες από τον ενδιάμεσο κώδικα και των πίνακα συμβόλων που θα μας φανούν απαραίτητες για την παραγωγή του τελικού κώδικα σε αρχιτεκτονική RISC-V.

- `quads = []`
- `final_scope=[]`

Με την πρώτη (`quads`) να διατηρεί τις τετράδες που δημιουργήθηκαν με την ολοκλήρωση της φάσης του ενδιάμεσου κώδικα και η δεύτερη λίστα (`final_scope`) αντιστοίχως να διατηρεί τα scopes του πίνακα συμβόλων.

5.3 Συναρτήσεις Τελικού Κώδικα

Οι συναρτήσεις του τελικού κώδικα μπορούν να διαχωριστούν σε δυο κατηγορίες της βοηθητικές και τις κύριες. Ξεκινώντας με τις βοηθητικές παρατηρούμε:

5.3.1 « ΒΟΗΘΗΤΙΚΕΣ ΣΥΝΑΡΤΗΣΕΙΣ »

5.3.1.1 GNVLCODE()

Μεταφέρει στον `t0` καταχωρητή την διεύθυνση μιας μη τοπικής μεταβλητής από τον πίνακα συμβόλων και βρίσκει πόσα επίπεδα επάνω βρίσκεται η μη τοπική μεταβλητή και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει.

Ξεκινήσαμε την συνάρτηση με έναν βρόχο που διατρέχει τον πίνακα με όλα τα scopes που δημιουργήθηκαν από τον πίνακα συμβόλων.

Εντός του βρόχου δημιουργήσαμε έναν εμφωλευμένο για να διατρέξουμε τις οντότητες κάθε επιπέδου της παραπάνω λίστας των `final_scopes`.

Στην συνέχεια η συνάρτηση ελέγχει αν το όνομα της τρέχουσας οντότητας (`j.name`) ταιριάζει με την παράμετρο εισόδου `x`. Αν ο έλεγχος είναι αληθής και βρεθεί μια αντιστοιχία, ανακτά το επίπεδο του `nesting_level` της μεταβλητής (`i.nesting_level`) και τη μετατόπισή της (`j.offset`) του τρέχοντος scope που βρίσκεται ο βρόχος.

Μετά την εύρεση των παραπάνω πληροφοριών της μεταβλητής, η συνάρτηση αρχικοποιεί μια συμβολοσειρά με όνομα `line`.

Ενημερώνει την μεταβλητή αυτή με την συμβολοσειρά: "`lw $t0, -4($sp)`" ώστε μετέπειτα η RISC-V να φορτώσει την τιμή από τον καταχωρητή `$sp` (μείον 4 byte) και να την αποθηκεύει στον `$t0 ->` (στοίβα του γονέα)

Επιπλέον, υλοποιούμε έναν ακόμα βρόχο που διατρέχει από το 0 (με την χρήση της `varscope`) έως το `cur_level` και επαναλαμβάνει την πρόσθεση της συμβολοσειρά "`lw $t0, -4($t0)`" στην μεταβλητή `line` για όσες φορές χρειαστεί `->` (στοίβα του προγόνου που έχει τη μεταβλητή)

Μετά τον βρόχο, προσθέτουμε άλλη μια γραμμή κώδικα στη `line` ώστε να προσθέτει την αρνητική τιμή του `varOffset` (το `offset` της μεταβλητής) στην τρέχουσα τιμή στο `$t0 ->` (διεύθυνση της μη τοπικής μεταβλητής)

Τέλος, ο λόγος ύπαρξης της συνάρτησης αυτής είναι ώστε να φορτώνει τις τιμές των διευθύνσεων μιας μη τοπικής μεταβλητής που βρίσκεται σε υψηλότερο `nesting level` από το τρέχον.

```

def gnvicode(x, cur_level): #writes value of non-local variable to register $t0
    for i in final_scope:
        for j in i.entity_list:
            if j.name == x:
                varscope=i.nesting_level
                varOffset=j.offset

    line = " lw $t0,-4($sp)\n"
    for i in range(varscope,cur_level):
        line+=" lw $t0, -4($t0)\n"
    line+= " add $t0,$t0,-" + str(varOffset) + "\n"
    return line

```

5.3.1.2 LOADVR()

Η συνάρτηση loadvr είναι υπεύθυνη για τη δημιουργία κώδικα που μετακινεί την τιμή μιας μεταβλητής ν από τη μνήμη σε έναν καθορισμένο καταχωρητή r. Η συνάρτηση λαμβάνει τρεις παραμέτρους: v (το όνομα της μεταβλητής), r (ο αριθμός του καταχωρητή) και cur_level (το τρέχον επίπεδο φωλιασμού). Ας αναλύσουμε τη συνάρτηση βήμα προς βήμα:

Αρχικοποίηση του varscope σε -1 και του temp_ent στην πρώτη οντότητα της λίστας final_scope.

Ξεκινήσαμε την συνάρτηση με έναν βρόχο που διατρέχει τον πίνακα με όλα τα scopes που δημιουργήθηκαν από τον πίνακα συμβόλων.

Εντός του βρόχου δημιουργήσαμε έναν εμφωλευμένο για να διατρέξουμε τις οντότητες κάθε επιπέδου της παραπάνω λίστας των final_scopes.

Στην συνέχεια η συνάρτηση ελέγχει αν το όνομα της τρέχουσας οντότητας (j.name) ταιριάζει με την παράμετρο εισόδου v. Αν ο έλεγχος είναι αληθής και βρεθεί μια αντιστοιχία, ανακτά το επίπεδο του nesting_level της μεταβλητής (i.nesting_level) αναθέτοντας το στο varscope και ενημερώνει το temp_ent να δείχνει στην αντιστοιχη οντότητα.

Η συνάρτηση ελέγχει διάφορες περιπτώσεις με βάση την τιμή του v και τη μεταβλητή varscope για να παράγει τον κατάλληλο κώδικα:

1. Εάν η v είναι σταθερά (προσδιορίζεται με τη χρήση της isdigit()), η συνάρτηση παράγει κώδικα χρησιμοποιώντας την εντολή "li" για να φορτώσει την τιμή της σταθεράς στον καταχωρητή \$t[r].
2. Εάν η v είναι μια καθολική μεταβλητή (με varscope ίσο με 1) δηλαδή αν ανήκει σε main συνάρτηση , η συνάρτηση παράγει κώδικα χρησιμοποιώντας την εντολή "lw" για να φορτώσει την τιμή από τη μετατόπιση της καθολικής μεταβλητής (temp_ent.offset) σε σχέση με τον καταχωρητή (\$gp) στον καταχωρητή \$t[r].
3. Εάν η v έχει δηλωθεί στο τρέχον πεδίο ορατότητας και είναι προσωρινή μεταβλητή, μεταβλητή ή παράμετρος επιστροφής που περνάει με τιμή, η συνάρτηση παράγει κώδικα χρησιμοποιώντας την εντολή "lw" για να φορτώσει την τιμή από τη μετατόπιση της μεταβλητής (temp_ent.offset) σε σχέση με τον καταχωρητή (\$sp) στον καταχωρητή \$t[r].

4. Εάν η ν δηλώνεται σε μικρότερο πεδίο ορατότητας από το τρέχον και είναι προσωρινή μεταβλητή, μεταβλητή ή παράμετρος επιστροφής που περνάει με τιμή, η συνάρτηση παράγει κώδικα χρησιμοποιώντας τη συνάρτηση gnvicode για να φορτώσει την τιμή της ν στον καταχωρητή \$t0 και στη συνέχεια χρησιμοποιεί την εντολή "lw" για να φορτώσει την τιμή από τη μνήμη στη διεύθυνση που είναι αποθηκευμένη στην \$t0, στον καταχωρητή \$t[r].
5. Εάν η ν δηλώνεται στο τρέχον πεδίο ορατότητας και είναι μια παράμετρος που περνάει μέσω αναφοράς, η συνάρτηση παράγει κώδικα για να φορτώσει τη διεύθυνση της παραμέτρου (temp_ent.offset) σε σχέση με τον καταχωρητή (\$sp) στον καταχωρητή \$t0 και στη συνέχεια χρησιμοποιεί την εντολή "lw" για να φορτώσει την τιμή από τη μνήμη στη διεύθυνση που είναι αποθηκευμένη στο \$t0 στον καταχωρητή \$t[r].
6. Εάν η ν δηλώνεται σε μικρότερο πεδίο ορατότητας από το τρέχον και είναι μια παράμετρος που περνάει μέσω αναφοράς, η συνάρτηση χρησιμοποιεί τη συνάρτηση gnvicode για να φορτώσει τη διεύθυνση της ν στον καταχωρητή \$t0 και στη συνέχεια χρησιμοποιεί δύο εντολές "lw" για να φορτώσει πρώτα την τιμή από τη μνήμη στη διεύθυνση που είναι αποθηκευμένη στην \$t0 στον ίδιο τον καταχωρητή \$t0 και στη συνέχεια να φορτώσει την τιμή από τη μνήμη στην ενημερωμένη διεύθυνση στην \$t0 στον καταχωρητή \$t[r].
7. Αν καμία από τις παραπάνω συνθήκες δεν ταιριάζει, η συνάρτηση επιστρέφει ένα μήνυμα σφάλματος που υποδεικνύει ότι η φόρτωση του temp_ent.name αντιμετώπισε σφάλμα.

```

def loadvr(v, r, cur_level): #moves value of data from memory to a register
    varscope=-1
    temp_ent=final_scope[0].entity_list[0]
    for i in final_scope:
        for j in i.entity_list:
            if(j.name==v):
                varscope=i.nesting_level
                temp_ent=j
                break
    if(v.isdigit()):
        code=" li $t"+str(r)+","+str(v)+" /* loads "+str(v)+" */\n" #loads constants
    elif(varscope==1):
        code=" lw $t"+str(r)+","+str(temp_ent.offset)+"(gp) /* loads "+str(v)+" */\n" #loads global variables
    elif(varscope==cur_level and (temp_ent.type=="temp_var" or temp_ent.type=="var" or (temp_ent.type=="par" and temp_ent.par_mode=="RET"))):
        #loads temp variables, variable, or return parameters that are in current scope
        code=" lw $t"+str(r)+",-"+str(temp_ent.offset)+"(sp) /* loads "+str(v)+" */\n"
    elif(varscope<cur_level and (temp_ent.type=="temp_var" or temp_ent.type=="var" or (temp_ent.type=="par" and temp_ent.par_mode=="RET"))):
        #loads temp variables, variable, or return parameters that are in lower scope
        code=gnvicode(temp_ent.name, cur_level)
        code+=" lw $t"+str(r)+",($t0) /* loads "+str(v)+" */\n"
    elif(varscope==cur_level and (temp_ent.type=="par" and temp_ent.par_mode=="CV")): #loads pass by refernce parameters in current scope
        code=" lw $t0,-"+str(temp_ent.offset)+(sp)\n"
        code+=" lw $t"+str(r)+",($t0) /* loads "+str(v)+" */\n"
    elif(varscope<cur_level and (temp_ent.type=="par" and temp_ent.par_mode=="CV")): #loads pass by refernce parameters in lower scope
        code=gnvicode(temp_ent.name, cur_level)
        code+=" lw $t0,($t0)\n"
        code+=" lw $t"+str(r)+",($t0) /* loads "+str(v)+" */\n"
    else:
        code="error loading "+temp_ent.name+"\n"
    return code

```

5.3.1.3 STORERV()

Η βοηθητική συνάρτηση storerv() μεταφέρει δεδομένα από τον καταχωρητή r στην μνήμη (μεταβλητή v) και όπως και προηγουμένως διακρίνουμε τις παρακάτω περιπτώσεις:

1. Αν v είναι καθολική μεταβλητή – δηλαδή αν ανήκει σε main συνάρτηση τότε “sw tr,-offset(gp)”
2. Αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή τότε “sw tr,-offset(sp)”
3. Αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον τότε:
 - a. lw t0,-offset(sp)
 - b. sw tr,(t0)
4. Αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον τότε:
 - a. gnlvcode(v)
 - b. sw tr,(t0)
5. Αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:
 - a. gnlvcode(v)
 - b. lw t0,(t0)
 - c. sw tr,(t0)
6. Αν καμία από τις παραπάνω συνθήκες δεν ταιριάζει, η συνάρτηση επιστρέφει ένα μήνυμα σφάλματος που υποδεικνύει ότι η μεταφορά του temp_ent.name αντιμετώπισε σφάλμα.

```

def storerv(v, r, cur_level): #moves value of data from a register to memory
    varscope=-1
    temp_ent=final_scope[0].entity_list[0]
    for i in final_scope:
        for j in i.entity_list:
            if(j.name==v):
                varscope=i.nesting_level
                temp_ent=j
                break

    if(varscope==1):
        code="    sw $t"+str(r)+"," +str(temp_ent.offset)+"($gp)      /* stores "+str(v)+" */\n" #stores global variables
    elif(varscope==cur_level and (temp_ent.type=="temp_var" or temp_ent.type=="var" or (temp_ent.type=="par" and temp_ent.par_mode=="RET"))):
        #stores temp variables, variable, or return parameters that are in current scope
        code="    sw $t"+str(r)+"," +str(temp_ent.offset)+"($sp)      /* stores "+str(v)+" */\n"
    elif(varscope<cur_level and (temp_ent.type=="temp_var" or temp_ent.type=="var" or (temp_ent.type=="par" and temp_ent.par_mode=="RET"))):
        #stores temp variables, variable, or return parameters that are in lower scope
        code=gnvlcode(temp_ent.name, cur_level)
        code+="    sw $t"+str(r)+",($t0)      /* stores "+str(v)+" */\n"
    elif(varscope==cur_level and (temp_ent.type=="par" and temp_ent.par_mode=="CV")): #stores pass by reference parameters in current scope
        code="    lw $t0," +str(temp_ent.offset)+"($sp)\n"
        code+="    sw $t"+str(r)+",($t0)      /* stores "+str(v)+" */\n"
    elif(varscope<cur_level and (temp_ent.type=="par" and temp_ent.par_mode=="CV")): #stores pass by reference parameters in lower scope
        code=gnvlcode(temp_ent.name, cur_level)
        code+="    lw $t0,($t0)\n"
        code+="    sw $t"+str(r)+",($t0)      /* stores "+str(v)+" */\n"
    else:
        code="error storing "+temp_ent.name+"\n"

    return code

```

5.3.1.4 FIND_LABELS()

Την συγκεκριμένη βοηθητική συνάρτηση την δημιουργήσαμε προκειμένου να αποθηκεύσουμε τις τιμές όλων των γραμμών στις οποίες θα θελήσουμε να μεταπηδήσουμε (jumps) στη λίστα ‘labels’.

Προκειμένου να το επιτύχουμε, ακολουθήσαμε τα εξής βήματα υλοποίησης για την συνάρτηση `find_labels()`:

1. Αρχικά δημιουργήσαμε μια καθολική κενή λίστα με όνομα `labels`.
2. Δημιουργήσαμε έναν βρόχο προκειμένου να διατρέξουμε όλες τις τετράδες που δημιουργήθηκαν μετά την ολοκλήρωση του ενδιάμεσου κώδικα.
3. Ύστερα αναζητήσαμε από την λίστα των τετράδων εκείνα τα quads που υλοποιούν άλματα, δηλαδή όλα τα quads που ξεκινούν με οποιανδήποτε τελεστή σύγκρισης ή αναγράφουν το αλφαριθμητικό ‘jump’ και καταλήγουν σε αριθμητική σταθερά.
4. Η τιμή αυτή, της αριθμητικής σταθερά, υποδεικνύει και την γραμμή στην οποία θα πρέπει να μεταπηδήσουμε, έτσι προκειμένου να αποθηκεύσουμε όλες τις τιμές αυτές, τις προσθέσαμε στην λίστα `labels` με την εντολή “`labels.append(i[4])`”

```

def find_labels():
    global labels
    for i in quads:
        if(i[1] in ["<", ">", "==", "!=" , "<=", ">=", "jump"] and i[4]!="_"):
            labels.append(i[4])

```

5.3.2 « ΚΥΡΙΑ ΣΥΝΑΡΤΗΣΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ»

Η συνάρτηση `make_assembly()` είναι η συνάρτηση που έχει την ευθύνη να παράξει τον τελικό κώδικα χρησιμοποιώντας όλες τις παραπάνω βιοηθητικές συναρτήσεις που αναλύσαμε εκτενώς στην προηγουμένη ενότητα της αναφοράς μας. Με άλλα λόγια, διατρέξαμε την λίστα των quads και δημιουργήσαμε blocks κώδικα όπου ανάλογα με την εκάστοτε δομή που θέλουμε να δημιουργήσουμε και το δεύτερο όρισμα της τετράδες (του ενδιάμεσου κώδικα) στην οποία βρισκόμαστε, δημιουργήσαμε τον αντίστοιχο κώδικα για την υλοποίηση της δομής σε γλώσσα μηχανής RISC-V. Για να το κάνουμε αυτό χρησιμοποιήσαμε τις βιοηθητικές συναρτήσεις όπως αλώστε αναφέραμε προηγουμένως, καθώς και τις υποδείξεις των τελευταίων διαφανειών του μαθήματος από το αρχείο «Παραγωγή Τελικού Κώδικα.pdf».

Η συνάρτηση αποθηκεύει στην λίστα `asm_commands` όλες τις γραμμές κώδικα που δημιουργεί σε γλώσσα μηχανής RISK-V κατά την διάρκεια εκτέλεσή της, καθώς και ένα μικρό σχόλιο διπλά από κάθε εντολή προκειμένου να είναι ποιο ευανάγνωστος ο τελικός κώδικας στο αρχείο «`.asm`».

Συγκεκριμένα η μεθοδολογία που χρησιμοποιήσαμε είναι η εξής:

1. Αν αριθμός της τετράδας που προσπελάζουμε ανήκει στον πίνακα `labels` δημιουργούμε ένα `assembly label` με τον αριθμό αυτόν.
2. Φορτώνουμε σε καταχωρητές τις τιμές των μεταβλητών που εμπλέκονται στην εντολή που θα εκτελεσθεί με την `loadr`
3. Καλούμε την εντολή `assembly` που αντιστοιχεί στο περιεχόμενο της τετράδας
4. Αν η εντολή `assembly` αποθηκεύει κάποια τιμή αποτελέσματος σε καταχωρητή αποθηκεύουμε στην μνήμη την τιμή του καταχωρητή, στην μεταβλητή στην οποία αντιστοιχεί

Οι εντολές `assembly` που καταφέραμε να υλοποιήσουμε είναι:

1. Αριθμητικές πράξεις (`+, -, *, //`)
2. Λογικές πράξεις (`<, >, <=, >=, ==, !=`)
3. Εντολή άλματος (`jump`)
4. Εντολή εκχώρησης τιμής σε μεταβλητή (`assignment`)
5. Εντολή εισόδου, εξόδου και επιστροφής δεδομένων (`input, print, return`)
6. Αρχή και τέλος μπλοκ συνάρτησης
7. Κλήση `main` συνάρτησης

```

def make_assembly():
    find_labels()
    global labels
    curr_level=0
    for i in quads:
        #print("quad num is "+str(i[0])+" labels is "+''.join(map(str, labels)))
        code=""
        if(i[1]=="+"): #makes assembly code for addition
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    add $" + str(i[4]) + ", $t1, $t2      /* " + str(i[4]) + "=" + str(i[2]) + str(i[1]) + str(i[3]) + " */\n"
            code+=storerv(i[4],3,curr_level)
            asm_commands.append(code)
        elif(i[1]=="-"): #makes assembly code for subtraction
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    sub $" + str(i[4]) + ", $t3, $t2      /* " + str(i[4]) + "=" + str(i[2]) + str(i[1]) + str(i[3]) + " */\n"
            code+=storerv(i[4],3,curr_level)
            asm_commands.append(code)
        elif(i[1]=="*"): #makes assembly code for multiplication
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    mul $" + str(i[4]) + ", $t3, $t2      /* " + str(i[4]) + "=" + str(i[2]) + str(i[1]) + str(i[3]) + " */\n"
            code+=storerv(i[4],3,curr_level)
            asm_commands.append(code)
        elif(i[1]=="/"): #makes assembly code for division
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    div $" + str(i[4]) + ", $t3, $t2      /* " + str(i[4]) + "=" + str(i[2]) + str(i[1]) + str(i[3]) + " */\n"
            code+=storerv(i[4],3,curr_level)
            asm_commands.append(code)
        elif(i[1]=="//"): #makes assembly code for less than jump
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    blt $" + str(i[4]) + ", $t1, " + str(i[4]) + "      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
            asm_commands.append(code)
        elif(i[1]=="<="): #makes assembly code for less equal jump
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    ble $" + str(i[4]) + ", $t1, " + str(i[4]) + "      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
            asm_commands.append(code)
        elif(i[1]==">"): #makes assembly code for greater than jump
            if(i[0] in labels):
                code+="label : "+str(i[0])+"\n"
            code+=loadrv(i[2],1,curr_level)
            code+=loadrv(i[3],2,curr_level)
            code+="    bgt $" + str(i[4]) + ", $t1, " + str(i[4]) + "      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
            asm_commands.append(code)

```

```

elif(i[1]==">>="): #makes assembly code for greater equal jump
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        code+=loadvr(i[2],1,curr_level)
        code+=loadvr(i[3],2,curr_level)
        code+="    bge $" + str(i[1])+",$" + str(i[4])+"      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
        asm_commands.append(code)
elif(i[1]==">="): #makes assembly code for equals jump
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        code+=loadvr(i[2],1,curr_level)
        code+=loadvr(i[3],2,curr_level)
        code+="    beq $" + str(i[1])+",$" + str(i[4])+"      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
        asm_commands.append(code)
elif(i[1]==!="): #makes assembly code for not equals jump
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        code+=loadvr(i[2],1,curr_level)
        code+=loadvr(i[3],2,curr_level)
        code+="    bne $" + str(i[1])+",$" + str(i[4])+"      /* if "+str(i[2])+str(i[1])+str(i[3])+" then jump to "+str(i[4])+" */\n"
        asm_commands.append(code)
elif (i[1]=="jump" and i[4]!="_"): #makes assembly code for jump to label
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        code+="    b "+str(i[4])+"      /* jump to "+str(i[4])+" */\n"
        asm_commands.append(code)
elif (i[1]==":="): #makes assembly code for value assignment to variable...
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        if(i[2]=="input"): #...from input
            code+="    li $v0,5\n"
            code+="    ecall\n"
            code+="    move $t0,$v0      /* input("+str(i[4])+" */\n"
            code+=storerv(0,i[4],curr_level)
            asm_commands.append(code)
        else: #...or other variable
            code+=loadvr(i[2],1,curr_level)
            code+="    /* "+str(i[4])+" = "+str(i[2])+" */\n"
            code+=storerv(i[4],1,curr_level)
            asm_commands.append(code)
    elif(i[1]=="retv"): #makes assembly code for return statement
        if(i[0] in labels):
            code+="label : "+str(i[0])+"\n"
            code+=loadvr(i[2],1,curr_level)
            code+="    lw t0,-8($p)\n"
            code+="    sw t1,(t0)      /* return("+str(i[2])+" */\n"
            asm_commands.append(code)

```

```

asm_commands.append(i)
elif(i[1]=="out"): #makes assembly code for print statement
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        code+=loadvr(i[2],0,curr_level)
        code+="    mv $a0, $t0\n"
        code+="    li $a7, 1\n"
        code+="    ecall\n"
        code+="    la $a0,str_nl\n"
        code+="    li $a7,4\n"
        code+="    ecall /* print("+str(i[2])+") /*\n"
    asm_commands.append(code)
elif(i[1]=="begin_block"): #makes assembly code for function blocks
    if(i[2]=="__main__"): #__main function
        main_framelength=0
        for j in final_scope[0].entity_list:
            if j.type=="main":
                main_framelength=j.framelength
        code="\nj Lmain /* main block begins */\n"
        code+="    addi sp,sp,"+str(main_framelength)+"\n"
        code+="    move gp,sp\n"
        asm_commands.append(code)
    else: ... all other functions
        curr_level+=1
        code="    label : "+i[2]+/* "+str(i[2])+" block begins */\n"
        asm_commands.append(code)
elif(i[1] == "end_block"): #marks endong of function blocks
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        labels.pop(0)
    if(i[2]=="__main__"):
        code+="/* main block ends */\n\n"
        asm_commands.append(code)
    elif("main" in i[2] and i[2]!="__main__"):
        curr_level-=1
        code+="/* "+str(i[2])+" block ends */\n"
        asm_commands.append(code)
    else:
        curr_level-=1
        code+="/* "+str(i[2])+" block ends */\n\n"
        asm_commands.append(code)
elif(i[1]=="call" and ("main" in i[2] and i[2]!="__main__")):
    code+="    j "+i[2]+\n"
    asm_commands.append(code)
elif(i[1]=="call" and "main" not in i[2]):
    code+="    #\n cant call function "+i[2]+\n\n"
    asm_commands.append(code)
else:
    if(i[0] in labels):
        code+="label : "+str(i[0])+"\n"
        labels.pop(0)

```

5.4 Αρχειο « .asm »

Η συνάρτηση που εμφανίζεται παρακάτω καλεί και εκτελεί την συνάρτηση `make_assembly()` καθώς παράλληλα δημιουργεί ένα αρχείο με όνομα την ονομασία του αρχείου της CutePy και κατάληξη «.asm». Τέλος, στο αρχείο αυτό που δημιούργησε η συνάρτηση, γράφουμε όλων των κώδικα της RISK-V διατρέχοντας την λίστα `asm_commands()`.

```

def asm_file():
    make_assembly()
    filename = cutePy_file_name.split(".cpy")
    filename=filename[0]+".asm"
    asm_file = open(filename, 'w')
    for i in asm_commands:
        asm_file.write(i)

```

6. Ενότητα: “Σχολιασμός Προβλημάτων”

Κατά την ανάπτυξη της λύσης μας εντοπίσαμε κάποια λάθη σε σημεία του κώδικα τα οποία και διορθώσαμε για την τελική παράδοση. Συγκεκριμένα λάθη βρέθηκαν:

1. Στην αποτίμηση των συνθηκών `bTrue`, `bFalse` σε συνθήκη που περιέχει `not`
2. Στην δημιουργία τετράδων που αφορούν την παράμετρο που περιέχει την τιμή, που επιστρέφει μια κλήση συνάρτησης
3. Στον υπολογισμό των σωστών `offset` των οντοτήτων του προγράμματος, στη φάση του πίνακα συμβολών

Επίσης μετά την παράδοση του πίνακα συμβόλων αντιληφθήκαμε ότι θα είναι καλύτερα να τυπώνουμε τα περιεχόμενα του σε ένα αρχείο παρά στο τερματικό. Έτσι προσθέσαμε την λειτουργία αυτή στην τελική παράδοση

Όσον αφορά την πληρότητα της λύσης μας έχουν υλοποιηθεί όλα τα ζητούμενα εκτός της κλήσης συναρτήσεων στον τελικό κώδικα.