

1) Λεκτικός Αναλυτής

Ο λεκτικός αναλυτής υλοποιείται με την συνάρτηση **lex()** (γραμμή 118), ο οποίος κάθε φορά που καλείται, διαβάζει όσους χαρακτήρες χρειαστεί απ' το αρχείο σε Cimple μέχρι να συμπληρωθεί μια λεκτική μονάδα και στο τέλος, αν δεν υπήρξε κάποιο σφάλμα, την επιστρέφει.

Το αρχείο Cimple το ανοίγουμε και το αποθηκεύουμε ως λίστα από χαρακτήρες στο **filedata** με τις παρακάτω εντολές:

```
91 file = open(sys.argv[1], encoding="utf8")
92 filedata = list(file.read())
```

Πριν αρχίσει η επεξήγηση του *lex()*, θα αναφερθούμε πρώτα σε ορισμένες ενέργειες που γίνονται σε global επίπεδο.

Υπάρχουν ορισμένες μεταβλητές που μπορεί να χρειαστούμε τις τιμές τους και αφού ολοκληρωθεί η κλήση του *lex()*. Για αυτό τον λόγο τις ορίζουμε ως global και τις αρχικοποιούμε ως εξής:

82	currentType = ""	96	pos = 0 # position in file
83	currentWord = ""	97	line = 1 # line in file
84	lastWord = ""	98	inputChar = filedata[pos]

Το **pos** θα κρατάει την θέση του χαρακτήρα στο *filedata* που διαβάζει ο *lex()*, ο οποίος χαρακτήρας θα αποθηκεύεται στο **inputChar**. Κάθε φορά που τερματίζει ο *lex()*, θα επιστρέφει μια λεκτική μονάδα, η οποία θα αποθηκεύεται στο **currentWord** με την εντολή *currentWord = lex()*. Το **line** κρατάει την γραμμή του αρχείου που βρίσκεται ο *inputChar*, αλλά εφόσον στην Cimple δεν παίζουν ιδιαίτερο ρόλο οι γραμμές, η μεταβλητή αυτή χρησιμοποιείται μόνο στην εκτύπωση μυνήματος σφάλματος. Οι **currentType** και **lastWord** θα εξηγηθούν παρακάτω.

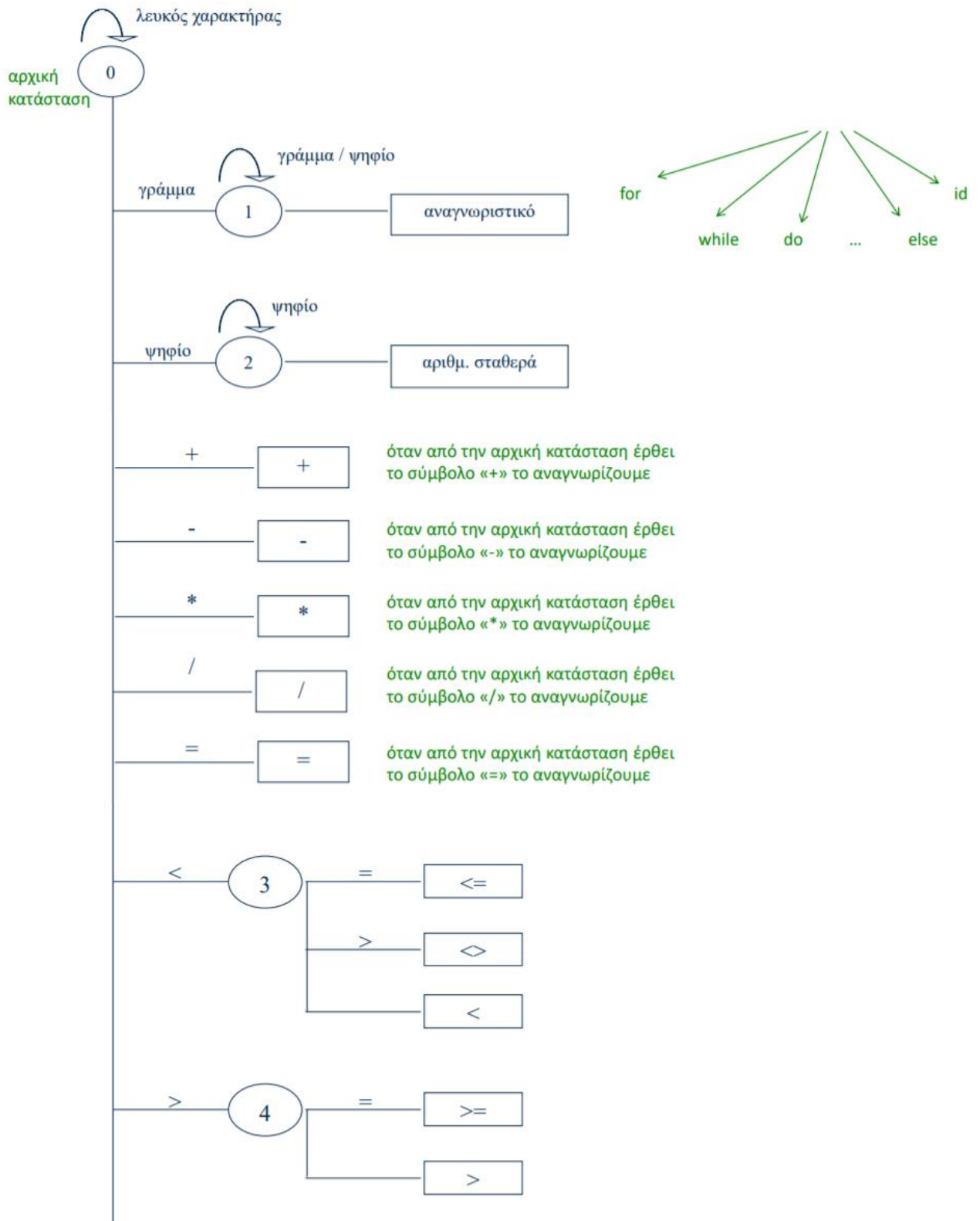
Τέλος, ορίζονται οι παρακάτω global λίστες, που κάθε μια από αυτές κρατάει μια κατηγορία λέξεων ή συμβόλων. Έτσι όταν θα χρειαστεί να βρούμε τι κατηγορίας είναι μια λεκτική μονάδα ή κάποιος χαρακτήρας, θα μπορούμε να ελέγξουμε αν ανήκει σε κάποια από αυτές τις λίστες.

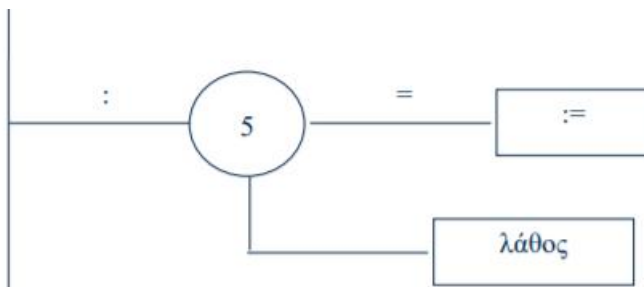
```

70
71     letters = list(string.ascii_letters)
72     numbers = list(string.digits)
73     mathSyms = ["+", "-", "*", "/", "="]
74     specialChars = [",", ";", "(", ")", "{", "}", "[", "]"
75     delimiters = [" ", "\t", "\n"]
76     relopes = ["=", "<", "<=", ">", ">=", "<>"]
77

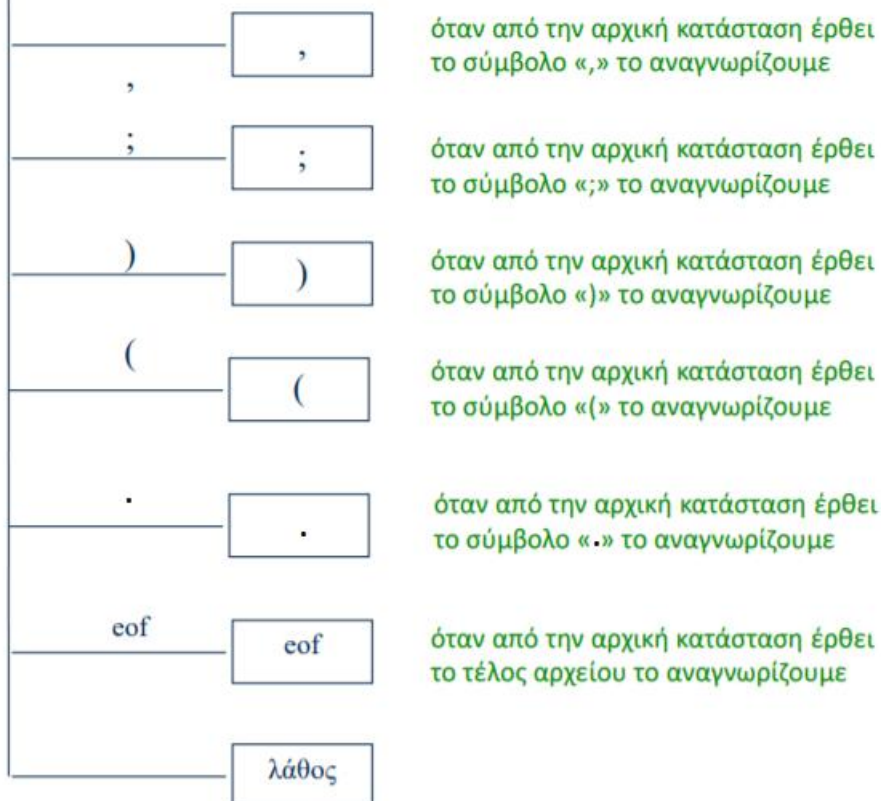
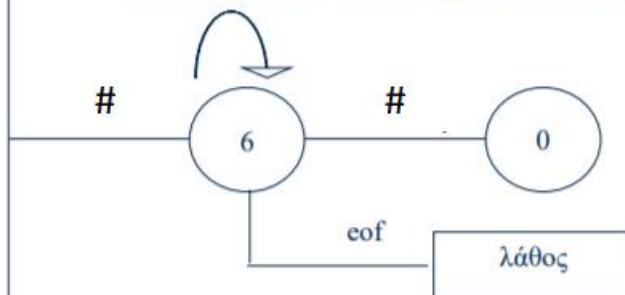
```

Αμέσως μετά ορίζεται η συνάρτηση *lex()* στην οποία υλοποιείται ο λεκτικός αναλυτής, ο οποίος βασίζεται στο παρακάτω αυτόματο:





αν από την αρχική κατάσταση συναντήσουμε άνοιγμα σχολίων τότε ό,τι ακολουθήσει αγνοείται έως ότου βρεθεί κλείσιμο σχολίων οπότε μεταβαίνουμε στην κατάσταση 0, που είναι η αρχική κατάσταση



Η υλοποίηση έγινε με σειρά εντολών απόφασης, με την χρήση `while(state<7)`, όπου το **state** κρατάει την κατάσταση του αυτομάτου, με `state = 7` για κατάσταση σφάλματος και `state = 8` για τελική κατάσταση.

Κάθε φορά που καλείται η συνάρτηση `lex()`, στην αρχή της, πριν ξεκινήσει το `while` γίνονται πρώτα οι εξής αρχικοποιήσεις:

```
125     lastWord = currentWord
126     currentType = ""
127     comment = False
128     state = 0
129     word = "" # Empty list to return word unit
```

Η global μεταβλητή **lastWord** παίρνει την τιμή του `currentWord` έτσι ώστε να κρατάμε κάπου την προηγούμενη λεκτική μονάδα σε περίπτωση που χρειαστεί, και η **currentType** παίρνει τις τιμές "Word" και "Number" (θα φανεί παρακάτω η χρήση του). Από τις τοπικές, η **comment** λειτουργεί σαν flag έτσι ώστε να ξέρουμε αν ο `lex()` διαβάζει χαρακτήρες μέσα σε σχόλιο, η **state** κρατάει την τιμή της κατάστασης του αυτομάτου που υλοποιούμε, και στη **word** θα προστίθεται κάθε φορά ο χαρακτήρας που διαβάζει η `lex()` έτσι ώστε να συμπληρώσει την λεκτική μονάδα και εν τέλει να την επιστρέψει.

Στην συνέχεια αρχίζει ο βρόγχος `while`, όπου θα εκτελεστούν τόσες επαναλήψεις όσες χρειαστούν μέχρι να βρεθεί τελική κατάσταση ή σφάλμα. Συγκεκριμένα όταν το `state = 8` ή `state = 7` αντίστοιχα. Στις πρώτες γραμμές θεωρητικά γίνεται η μετάβαση απ' την μια κατάσταση του αυτόματου στην άλλη, και πρακτικά γίνεται με την αλλαγή της τιμής του `state`, αναλόγως την περίπτωση.

```
131     while state < 7:
132         if state == 0:...
153
154         elif state == 1:...
158
159         elif state == 2:...
167
168         elif state == 3:...
173
174         elif state == 4:...
179
180         elif state == 5:...
187
188         elif state == 6:...
```

Οι διακλαδώσεις `if-elif` υλοποιήθηκαν με βάση το αυτόματο, και ανάλογα με την τιμή του `inputChar` και με την βοήθεια των global λιστών ελέγχει σε ποια περίπτωση θα μεταβεί. Στο τέλος κάθε περίπτωσης είτε θα αλλάξει τιμή το `state` έτσι ώστε να μεταβεί σε άλλη κατάσταση, ή θα τερματίσει ο βρόγχος.

Πιο συγκεκριμένα:

```
132 if state == 0:
133     if inputChar in letters:
134         state = 1
135     elif inputChar in numbers:
136         state = 2
137     elif (inputChar in mathSyms) or (inputChar in specialChars):
138         state = 8 # END
139     elif inputChar == "<":
140         state = 3
141     elif inputChar == ">":
142         state = 4
143     elif inputChar == ":":
144         state = 5
145     elif inputChar == "#":
146         state = 6
147     elif inputChar == ".":
148         state = 8
149     elif (inputChar != "\n") and (inputChar != " ") and (inputChar != "\t"):
150         state = 7
151     print("ERROR! at line " + str(line) + ". Unknown character: '" + inputChar + "'")
152     exit()
153
```

- **state = 0:** Στην περίπτωση αυτή είμαστε στην αρχική κατάσταση του αυτομάτου. Ο *lex()* έχει μόλις κληθεί, οπότε ψάχνει τον επόμενο χαρακτήρα, προκειμένου να αναγνωρίσει την επόμενη λέξη.

Η κατάσταση **state = 0** κατηγοριοποιείται ως εξής:

- **state = 1**, αν ο χαρακτήρας που διαβάζει ανοίκει στην λίστα *letters*, πράγμα που σημαίνει ότι είναι γράμμα του αγγλικού αλφάβητου
- **state = 2**, αν ο χαρακτήρας είναι αριθμός (ανήκει στην λίστα *numbers*).
- **state = 3**, αν ο χαρακτήρας είναι "<".
- **state = 4**, για να καλυφθεί η περίπτωση που ο πρώτος χαρακτήρας είναι ">".
- **state = 5**, αν το *inputchar* είναι ":".
- **state = 6**, για να αρχικοποιήσουμε σχόλιο με "#".
- **state = 7**, για οποιονδήποτε άλλον χαρακτήρα, που δεν μπορεί να αναγνωριστεί ως έγκυρος.
- **state = 8**, όταν φτάσουμε στο σύμβολο ".".

1. ELA RE FILE POG 🤪



Με την επόμενη επανάληψη του **while** (εφόσον το **state** είναι ακόμα μικρότερο του **7**), θα μεταβούμε σε μια από τις παραπάνω καταστάσεις.

- **state = 1:** Ο *lex()* βρήκε λέξη, άρα θα δέχεται μόνο γράμματα κι αριθμούς.

```
154 elif state == 1: # This is a word, only letters and numbers allowed
155     if not ((inputChar in numbers) or (inputChar in letters)):
156         currentType = "Word"
157         break
```

- *state* = 2: Ο *lex()* εντόπισε αριθμό, άρα θα δέχεται μόνο αριθμούς, χωρίς βέβαια να υπερβαίνει τον αριθμό: $2^{23} - 1$ ή να είναι μικρότερος απ' τον αριθμό: $-2^{23} - 1$.

```

159 elif state == 2: # Number, only numbers allowed
160     if not (inputChar in numbers):
161         currentType = "Number"
162         if float(word) > math.pow(2, 23) - 1 or float(word) < -(math.pow(2, 23) - 1):
163
164             print("Number out of boundaries in line " + str(line))
165             exit()
166         break

```

- *state* = 3: Ο *lex()* εντόπισε το σύμβολο "<", άρα θα δεχτεί "=" δηλαδή "<=", που σημαίνει μικρότερο ή ίσο, ή ">" δηλαδή "<=" που συμβολίζει το διάφορο.

```

168 elif state == 3: # means '<'
169     if inputChar == "=" or inputChar == ">":
170         state = 8
171     else:
172         break

```

- *state* = 4: Βρέθηκε ">", άρα θα δέχεται μόνο "=" δηλαδή ">=", συμβολίζοντας σύγκριση μεγαλύτερου ή ίσου.

```

174 elif state == 4: # means '>'
175     if inputChar == "=":
176         state = 8
177     else:
178         break

```

- *state* = 5: Έχει εντοπιστεί ":", επομένως το μόνο που μπορεί να ακολουθήσει είναι "=", ώστε να προκύψει ":", συμβολίζοντας την ανάθεση τιμής. Το ":" πρέπει πάντα να ακολουθείται από "=", οπότε αν δεν ισχύει αυτό τότε υπάρχει σφάλμα και το πρόγραμμα τερματίζει, εμφανίζοντας αντίστοιχο μήνυμα.

```

180 elif state == 5: # For ':'
181
182     if inputChar == "=":
183         state = 8
184     else:
185         print("ERROR! at line " + str(line) + ". '=' was expected after ':')
186         exit()

```

- *state* = 6: Ο *lex()* έχει αναγνωρίσει τον χαρακτήρα "#", συμβολίζοντας το ξεκίνημα ενός σχολίου, επομένως αγνοεί όλα όσα ακολουθούν μέχρι να βρεθεί ξανά "#" ή το *pos* να πάρει την τιμή *len(filedata)*, πράγμα που σημαίνει ότι φτάσαμε στο τέλος του προγράμματος χωρίς να κλείσουμε το σχόλιο. Τέλος, αρχικοποιούμε την boolean μεταβλητή **comment** που μας επιτρέπει να γνωρίζουμε αν είναι σχόλιο

αυτό που εμφανίστηκε, ώστε να μην το επιστρέψει.

```
188 elif state == 6: # For #comment#
189
190     if pos == len(filedata):
191         print("ERROR: a comment is never closed")
192         exit()
193
194     if inputChar == "#":
195         state = 8
196         comment = True
```

Αφού γίνει η μετάβαση απ' την μια κατάσταση στην άλλη, χωρίς να σταματήσει ο βρόγχος, προστίθενται στο word το *inputChar*, εφόσον δεν είναι λευκός χαρακτήρας

```
199     if not inputChar in delimiters:
200         word = word + inputChar
```

Έπειτα, αν το *inputChar* είναι "\n" σημαίνει ότι είχαμε αλλαγή γραμμής, οπότε αμέσως μετά εφόσον ισχύει, αυξάνει το line κατά 1

```
202     if inputChar == "\n": # Keep the current line
203         line = line + 1
```

Στην συνέχεια αυξάνουμε το *pos* κατά 1 και, εφόσον δεν έχουν τελειώσει οι χαρακτήρες του αρχείου, το *inputChar* παίρνει την τιμή του επόμενου. Αν όμως δεν υπάρχει άλλος χαρακτήρας σημαίνει ότι υπήρξε κάποιο συντακτικό λάθος στον κώδικα της Cimple και εμφανίζει μήνυμα σφάλματος, τερματίζοντας το πρόγραμμα με *exit()*

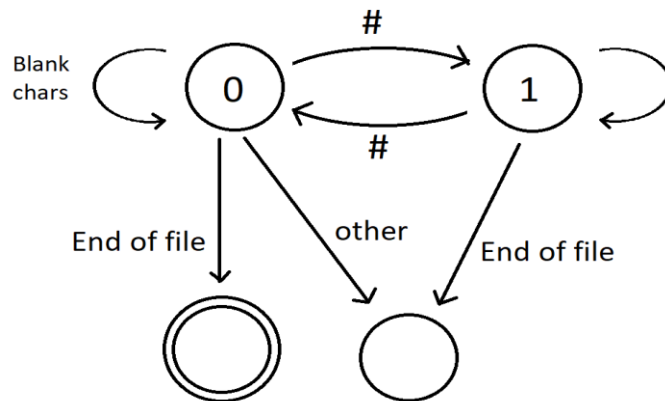
```
205     pos = pos + 1
206     if pos < len(filedata):
207         inputChar = filedata[pos]
208
209     elif pos > len(filedata):
210         print("ERROR! at line " + str(line) + ". Code ended unexpectedly")
211         exit()
```

Τέλος, αφού τερματίσει ο βρόγχος, ήρθε η ώρα να επιστραφεί η λεκτική μονάδα που είναι αποθηκευμένη στο *word*. Αν όμως η μεταβλητή *comment* έχει την τιμή *True*, σημαίνει ότι η λεκτική μονάδα ήταν μέσα σε σχόλιο, επομένως ξανακαλεί τον *lex()*

```
215     if not comment:
216         return word
217     else:
218         return lex()
```

Φαίνεται να έχουν καλυφθεί όλες οι περιπτώσεις απ' την αρχή ενός προγράμματος μέχρι το τέλος του, αλλά υπάρχει και το ενδεχόμενο να υπάρξει κώδικας και μετά το σύμβολο «}» το οποίο είναι το πέρας του προγράμματος. Για να μην επιβαρυνθεί κι άλλο ο *lex()*, για

αυτήν την περίπτωση δημιουργήσαμε την συνάρτηση **mk()**, η οποία υλοποιεί λεκτικό αναλυτή που βασίζεται στο παρακάτω αυτόματο:



Έχει συνολικά 4 καταστάσεις, και σκοπός του είναι να μην υπάρχουν λεκτικές μονάδες μετά την εμφάνιση του “}.” πέρα από λευκούς χαρακτήρες και σχόλια. Σχόλια που ανοίγουν και τελειώνει το αρχείο χωρίς να κλείσουν οδηγούν επίσης σε κατάσταση σφάλματος.

Η δομή του είναι παρόμοια με τον *lex()*, αλλά πιο απλή μιας και πλέον μας ενδιαφέρουν μόνο συγκεκριμένοι χαρακτήρες και δεν χρειάζεται να αποθηκεύουμε κάπου τις λεκτικές μονάδες.

Ξεκινάμε απ’ την αρχική κατάσταση του αυτόματου και για αυτό αρχικοποιούμε το *state* στην τιμή 0. Στην συνέχεια ξεκινάει ο βρόγχος με το *while* όπως και στον *lex()*, με την διαφορά ότι εδώ δεν θα σταματήσει με κάποια συνθήκη και θα κάνει κατευθείαν *return* όταν βρει την τελική κατάσταση *state* = 3.

```
227     state = 0
228     while True: # 2 = error, 3 = final
229         if state == 0: ...
236
237         elif state == 1: ...
242
243         elif state == 2: ...
246
247         elif state == 3:
248             return
```

Πιο συγκεκριμένα οι μεταβάσεις των καταστάσεων υλοποιούνται ως εξής:

-Κατάσταση 0:

Όπως φαίνεται και στο παραπάνω αυτόματο, όταν βρισκόμαστε στην συγκεκριμένη κατάσταση χρειαζόμαστε το τέλος του αρχείου για να πάμε σε τελική κατάσταση, ‘#’ για να

μεταβούμε στην κατάσταση 1, και οποιοδήποτε άλλο σύμβολο για μετάβαση σε κατάσταση σφάλματος. Οι λευκοί χαρακτήρες προφανώς αγνοούνται.

Επομένως, αφού ελεγχθεί ότι το `state` είναι ίσο με 0, εκτελούνται οι εξής εντολές:

```
229         if state == 0:
230             if inputChar in delimiters:
231                 state = 0
232             elif inputChar == "#":
233                 state = 1
234             else:
235                 state = 2
```

Με την βοήθεια της global λίστας `delimiters`, ελέγχουμε αν το `inputChar` είναι λευκός χαρακτήρας, και αν ναι παραμένουμε στην ίδια κατάσταση, εφόσον οι λευκοί χαρακτήρες αγνοούνται. Έπειτα γίνεται ο έλεγχος αν το `inputChar` ισούται με '#' για να μεταβεί στην κατάσταση 1, αλλιώς για οποιονδήποτε άλλο χαρακτήρα μεταβαίνει σε κατάσταση σφάλματος (`state = 2`).

Να σημειωθεί ότι ο έλεγχος για το τέλος του αρχείου γίνεται στο τέλος του βρόγχου `while`. Το ίδιο ισχύει και για τις υπόλοιπες καταστάσεις.

-Κατάσταση 1:

Όταν ο `mk()` βρίσκεται σε αυτή την κατάσταση σημαίνει ότι έχει ανοίξει κάποιο σχόλιο. Επομένως χρειαζόμαστε '#' για να κλείσει το σχόλιο και να μεταβούμε πάλι στην κατάσταση 1, οποιοσδήποτε άλλος χαρακτήρας αγνοείται, και αν εντοπίσει το τέλος του αρχείου σημαίνει ότι το σχόλιο δεν έκλεισε και μεταβαίνει σε κατάσταση σφάλματος

```
237         elif state == 1:
238             if inputChar == "#":
239                 state = 0
240             else:
241                 state = 1
```

Στην ουσία αυτό που γίνεται στον παραπάνω κώδικα είναι ότι αφού ελεγχθεί ότι το `state = 1`, κοιτάει αν το `inputChar` είναι '#' για να μεταβεί πάλι στην κατάσταση 0, αλλιώς παραμένει στην ίδια κατάσταση

-Κατάσταση 2 (σφάλμα):

Αφού είμαστε σε κατάσταση σφάλματος, η συνάρτηση θα τυπώσει το σχετικό μήνυμα και θα επιστρέψει

```
243 elif state == 2:
244     print("WARNING: Unreachable code after the '}' at the end of the program")
245     return
```

Να σημειωθεί εδώ ότι ο *mk()* δεν επιστρέφει κάποιο error, αλλά warning, δηλαδή θα σταματήσει η συνάρτηση με μήνυμα προειδοποίησης, αλλά το πρόγραμμα θα συνεχίσει να τρέχει και μετά το μήνυμα. Θεωρήσαμε ότι αφού το συντακτικό λάθος είναι σε περιοχή μετά από το τέλος του κώδικα της Cimple είναι προτιμότερο να γίνει με αυτόν τον τρόπο.

-Κατάσταση 3 (τελική):

```
247 elif state == 3:
248     return
```

Εφόσον είμαστε σε τελική κατάσταση *state = 3*, η συνάρτηση επιστρέφει

Πλέον έχουν ολοκληρωθεί οι διακλαδώσεις για την μετάβαση της κατάστασης, επομένως παρόμοια με τον *lex()* αυξάνουμε το *pos* κατά 1, και εφόσον υπάρχουν κι άλλοι χαρακτήρες στο *filedata*, το *inputChar* παίρνει την τιμή του επόμενου χαρακτήρα

```
250 pos = pos + 1
251 if pos < len(filedata):
252     inputChar = filedata[pos]
```

Εάν όμως δεν υπάρχουν άλλοι χαρακτήρες σημαίνει ότι φτάσαμε στο τέλος του αρχείου.

```
253 else:
254     if state == 1:
255         print("WARNING: Unreachable code after the '}' at the end of the program")
256         return
257     else:
258         state = 3
```

Αν λοιπόν αφού ο *mk()* έφτασε σε τέλος αρχείου και η κατάσταση του είναι η 1, σημαίνει ότι δεν έχει κλείσει το τελευταίο σχόλιο, επομένως τυπώνει μήνυμα προειδοποίησης και επιστρέφει. Εναλλακτικά, σε οποιαδήποτε άλλη κατάσταση το τέλος του αρχείου οδηγεί σε τελική κατάσταση, οπότε το *state* παίρνει την τιμή 3.

2) Συντακτικός Αναλυτής

Ο ρόλος του συντακτικού αναλυτή είναι το να ελέγξει αν ο κώδικας που δόθηκε από το αρχείο περιγράφεται από την γραμματική της Cimple. Η γραμματική που βασίσαμε τον κώδικα είναι αυτή από την εκφώνηση.

Η διαδικασία που ακολουθούμε είναι η εξής: Κάθε μεταβλητή της γραμματικής αντιστοιχεί σε μια συνάρτηση. Έτσι λοιπόν η συνάρτηση που αντιστοιχεί στην μεταβλητή στο αριστερό μέρος του κανόνα καλεί τις συναρτήσεις/μεταβλητές στο δεξί μέρος, και ταυτόχρονα ελέγχει αν είναι σωστά τα τερματικά σύμβολα.

Πχ στον παρακάτω κανόνα:

```
block    →  {
           declarations
           subprograms
           blockstatements
         }
```

Θα έχουμε δημιουργήσει σε python τις συναρτήσεις *block()*, *declarations()*, *subprograms()* και *blockstatements()*. Όπου η *block()* θα κηλεύει με την σειρά τις άλλες τρεις.

Παρακάτω επεξηγούμε αναλυτικά το πως κάθε κανόνας της γραμματικής μεταφράζεται σε κώδικα για να γίνει πιο ξεκάθαρο.

Πριν όμως αναλύσουμε τις συναρτήσεις, θα αναφερθούμε στην συνάρτηση **error()**, που όπως φαίνεται και από το όνομά της, καλείται όταν παρουσιαστούν κάποια συγκεκριμένα σφάλματα τα οποία δεν τα αναλαμβάνει ο **lex**, δηλαδή συντακτικά σφάλματα.

Η *error()*, χρησιμοποιεί ως κύρια μεταβλητή το **expectation**, τι περιμένει δηλαδή να δει. Φυσικά, δε γίνεται να κάθε φορά να ξέρει τι να περιμένει, οπότε απλά τυπώνει ότι δεν περίμενε το *currentWord* εκείνη τη στιγμή. Αν ωστόσο το *expectation* δεν είναι κενό, τότε τυπώνει τι περίμενε να ακολουθήσει μετά από το *lastWord*, αλλά βρήκε το *currentWord* αντί του *expectation*. Και στις δύο περιπτώσεις, το πρόγραμμα τερματίζει με την εντολή *exit()*. Έτσι, ο χρήστης μπορεί να πάει στο πρόγραμμά του και να επιδιορθώσει τυχόν σφάλματα προκειμένου να λειτουργήσει σωστά.

```
261 def error():
262     if expectation == "":
263         print("Error in line " + str(line) + ": \"" + currentWord + "\" wasn't expected after \"" + lastWord + "\"")
264     else:
265         print("Error in line " + str(line) + ": \"" + expectation + "\" was expected after \"" + lastWord + "\" but found \"" + currentWord + "\"")
266     exit()
```

Μετά από την συνάρτηση *error()*, ορίζονται οι συναρτήσεις που υλοποιούν την γραμματική της Cimple. Σε αυτό το μέρος της αναφοράς θα ασχοληθούμε μόνο με τις εντολές κάθε συνάρτησης που αφορούν την φάση του συντακτικού αναλυτή, και εντολές σχετικά με ενδιαμέσο κώδικα και πίνακα συμβόλων θα αγνοούνται

Οι συναρτήσεις αυτές είναι οι ακόλουθες:

-*program()*:

Αρχικά, έχουμε τον πρώτο κανόνα που αφορά την αρχή και το τέλος του κυρίως προγράμματος, ο οποίος υλοποιείται μέσα στην συνάρτηση *program()* (γραμμή 269)

program → program ID
block
.

Όπως φαίνεται και στην εικόνα, στην Cimple κάθε πρόγραμμα αρχίζει με την λεκτική μονάδα “program”. Οπότε σαν πρώτη εντολή, η συνάρτηση καλεί την *lex()* που θα επιστρέψει την πρώτη λεκτική μονάδα

```
278 if lex() != "program":  
279     print("ERROR! at line " + str(line))  
280     exit() # false
```

Εάν η τιμή που θα επιστρέψει δεν είναι “program”, τότε δεν συμφωνεί με τον κανόνα της γραμματικής και επομένως τυπώνει σχετικό μήνυμα λάθους (χρησιμοποιώντας την μεταβλητή *line* για να δείξει ακριβώς σε ποια γραμμή βρισκόμαστε) και τερματίζει το πρόγραμμα με την χρήση της *exit()*

Έπειτα προχωράμε στην επόμενη λεκτική μονάδα, αποθηκεύοντας την στην global μεταβλητή *currentWord*

```
285 currentWord = lex()
```

Κανονικά στη γραμματική το ID είναι μεταβλητή και έχει δικό του κανόνα, αλλά προτιμήσαμε να μην δημιουργήσουμε καινούργια συνάρτηση *ID()* και το ενσωματώνουμε κάθε φορά στην συνάρτηση που χρειάζεται. Αυτό που μας ενδιαφέρει για το ID, δηλαδή το όνομα του κύριου προγράμματος, είναι να μην ανήκει σε κάποια από τις δεσμευμένες λέξεις (πχ *print*, *function* κτλ) και να είναι έγκυρο το όνομά του (δηλαδή να αρχίζει από κάποιο γράμμα του αγγλικού αλφάβητου, και στην συνέχεια να ακολουθούν μόνο γράμματα ή αριθμοί).

Οι δεσμευμένες λέξεις είναι αποθηκευμένες σε μια global λίστα που ονομάζεται **standardWords**, οπότε το πρόγραμμα ελέγχει αν το *currentWord* ανήκει σε αυτήν την λίστα, και αν ανήκει πετάει σχετικό μήνυμα σφάλματος, τερματίζοντας το πρόγραμμα. Αμέσως κάτω γίνεται ο έλεγχος της εγκυρότητας του ονόματος, απλά ελέγχοντας αν το *currentType* είναι “Word”, το οποίο εξηγήσαμε νωρίτερα πως παίρνει τιμή μέσα στην *lex()* και εκτελεί τις ίδιες ενέργειες σε περίπτωση σφάλματος

```
292 if currentWord in standardWords:  
293     print("ERROR! at line " + str(line) + ". Word " +  
294         exit() # false  
295  
296 elif not currentType == "Word":  
297     print("ERROR! at line " + str(line) + ". Word " +  
298         exit() # false  
299
```

Στη συνέχεια βλέπουμε στη γραμματική ότι ακολουθεί η μεταβλητή `block`, οπότε στον κώδικα καλούμε την αντίστοιχη συνάρτηση `block()`.

```
300      if block() == 1:
301          error()
302      exit()
```

Κατά το κάλεσμά της, γίνεται ταυτόχρονα ο έλεγχος για την τιμή που θα επιστρέψει η συνάρτηση. Αν επιστρέψει 1, σημαίνει ότι βρέθηκε σφάλμα σε κάποια απ' τις επόμενες συναρτήσεις. Σε αυτή την περίπτωση καλεί την συνάρτηση `error()` για να τυπώσει το σχετικό μήνυμα λάθους και τερματίζει το πρόγραμμα.

Αφού τερματίσει η `block()`, καλείται πάλι ο `lex()` και περιμένουμε η τιμή που θα επιστρέψει να είναι ".". Αλλιώς εκτελεί την διαδικασία για μήνυμα λάθους.

```
304      if lex() != ".":
305          print("ERROR! at li
306      exit()
```

Τέλος, αφού ολοκληρώθηκε η διαδικασία για τον κανόνα `print`, πριν τερματίσει η συνάρτηση, καλεί για μοναδική φορά την `mk()` όπου θα γίνει ο έλεγχος αν υπάρχει κώδικας μετά το πέρας του προγράμματος, και στην συνέχεια θα επιστρέψει τυπώνοντας σχετικό μήνυμα

```
307
308      mk()
309
310      print("Program finished. No errors detected.")
311      return 0 # Reached EOF ( "." found)
```

-`block()`:

Στην συνάρτηση `block()` (γραμμή 314) ελέγχεται η κύρια δομή του προγράμματος, ή η δομή μιας συνάρτησης (υποπρογράμματος).

```
block      →      {
                                declarations
                                subprograms
                                blockstatements
                                }
```

Αρχικά καλείται ο `lex()` και ελέγχεται αν η τιμή που θα επιστρέψει είναι το σύμβολο "{", αλλιώς επιστρέφει την τιμή 1, η οποία τιμή θα φτάσει στην `program()` και θα τυπώσει

μήνυμα σφάλματος

```
321         if lex() != "{":
322             return 1
```

Στην συνέχεια καλείται ο *lex()* και η μεταβλητή *currentWord* παίρνει την τιμή που θα επιστρέψει. Στον κανόνα της γραμματικής ακολουθεί η μεταβλητή *declarations* που υλοποιείται απ' την ομώνυμη συνάρτηση και ελέγχει αν ορίζονται σωστά οι μεταβλητές. Υπάρχει όμως και το ενδεχόμενο, το πρόγραμμα ή το υποπρόγραμμα το οποίο ελέγχουμε να μην ορίζει μεταβλητές, οπότε υπάρχει και η περίπτωση να μην χρειαστεί να κληθεί η *declarations()*. Για αυτόν τον λόγο γίνεται ο έλεγχος αν το *currentWord* ισούται με την λέξη "declare", όπου τότε καλείται η *declarations()* και αυτομάτως αν η τιμή που επιστρέφει είναι 1, επιστρέφεται επίσης 1 και από την *block()*.

```
324         currentWord = lex()
325
326         if currentWord == "declare":
327             if declarations() == 1:
328                 return 1
```

Παρόμοια διαδικασία ακολουθούμε και για το subprograms. Ενδέχεται να μην υπάρχουν υποπρογράμματα, άρα συνεπώς δε θα υπάρχουν οι λέξεις "function" ή "procedure" που τα ορίζουν. Για αυτό τον λόγο ελέγχουμε αν το *currentWord* ισούται με κάποια απ' τις δυο αυτές λέξεις, έτσι ώστε να κληθεί η *subprograms()* ή αν δεν ισχύει, να προχωρήσει παρακάτω

```
330         if currentWord == "function" or currentWord == "procedure":
331             if subprograms() == 1:
332                 return 1
```

Επίσης με τον ίδιο τρόπο διαχειριζόμαστε το blockstatements. Κάθε εντολή στην Cimple θα αρχίζει είτε με κάποιο όνομα μεταβλητής για ανάθεση τιμής (*x := 2* κτλ) ή με κάποια εντολή (*if*, *while* κτλ). Τις εντολές αυτές τις αποθηκεύουμε στην global λίστα *statementWords*, όπως επίσης και τα ονόματα των μεταβλητών αποθηκεύονται στην global λίστα **varNames**.

Επομένως ελέγχεται αν το *currentWord* ανήκει σε κάποια από αυτές τις λίστες, και αντίστοιχα συνεχίζεται η διαδικασία όπως με τις προηγούμενες δυο συναρτήσεις

```
343         if currentWord in statementWords or currentWord in varNames:
344             if blockstatements() == 1:
345                 return 1
```

Τέλος, πριν επιστρέψει η τρέχουσα συνάρτηση, ελέγχει αν το *currentWord* ισούται με "}". Αν δεν ισχύει αυτό καλεί την *error()* για να διαχειριστεί το λάθος, αλλιώς επιστρέφει την τιμή 0.

```
359         if currentWord != "}":
360             error()
```

Παρατηρήσεις:

1. Η συνάρτηση *error()* καλείται μόνο στην *program()* (με εξαιρετικά ελάχιστες εξαιρέσεις που μπορεί να κληθεί και αλλού)... TODO

-declarations:

Η *declarations()* υλοποιεί τον κανόνα της γραμματικής υπεύθυνο για το μέρος του κώδικα που ορίζονται οι μεταβλητές.

declarations → **(declare varlist ;)***

Το τερματικό σύμβολο “declare” έχει ήδη ελεγχθεί αν υπάρχει από την *block()*, οπότε καλείτε απευθείας η συνάρτηση *varlist()* και με την βοήθεια της τοπικής μεταβλητής *retvar*, γίνεται ο έλεγχος για το αν χρειαστεί να επιστρέψει 1.

```
375      retvar = varlist()
376
377      if retvar == 1:
378          return 1
```

Στην συνέχεια το *currentWord*, στο οποίο είναι αποθηκευμένη η τελευταία λεκτική μονάδα που επέστρεψε ο *lex()*, ελέγχεται αν έχει την τιμή “;”, ώστε να πάει σύμφωνα με τον κανόνα. Να σημειωθεί εδώ ότι ο *lex()* δεν χρειάστηκε να κληθεί, διότι τον κάλεσε ήδη η *varlist()* όσες φορές χρειάστηκε.

```
380      if currentWord != ";":
381          expectation = ";"
382          return 1
```

Ελέγξαμε και το τελευταίο τερματικό σύμβολο αλλά δεν ολοκληρώθηκε η διαδικασία, καθώς υπάρχει Kleene Star στον κανόνα. Εφόσον το Kleene Star αφορά ολόκληρο το δεξί μέρος, δηλαδή με άλλα λόγια επαναλαμβάνεται όλη η συνάρτηση, αυτό σημαίνει ότι μπορεί να υλοποιηθεί με αναδρομή.

Επομένως αφού κληθεί ο *lex()* και πάρει την τιμή το *currentWord*, ελέγχεται αν η τιμή αυτή είναι “declare”. Αν ισχύει αυτό, τότε η *declarations()* καλεί τον εαυτό της, διαφορετικά επιστρέφει 0.


```

384     currentWord = lex()
385     if currentWord == "declare":
386         if declarations() == 1:
387             return 1
388     elif currentWord == ";":
389         return 1
390     else:
391         return 0

```

Ο βρόγχος elif παρέμεινε στον κώδικα εκ παραδρομής και είναι περιττός, διότι ελέγχει μια περίπτωση σφάλματος η οποία ελέγχεται έτσι κι αλλιώς την στιγμή που πρέπει.

-varlist:

Στο varlist() ελέγχεται η εγκυρότητα των ονομάτων των μεταβλητών και το αν ορίζονται ορθά μετά την Cimple εντολή declare.

```

varlist      →      ID
               ( , ID ) *
               |
               ε

```

Η Cimple επιτρέπει το ενδεχόμενο να υπάρξει η εντολή declare χωρίς απαραίτητα να ορίζεται κάποια μεταβλητή μετά. Επομένως αφού κληθεί ο lex(), ελέγχεται αν το currentType (το οποίο όπως προαναφέρθηκε, κρατάει αν η τιμή που επιστρέφει ο lex() είναι "word" ή "number") έχει την τιμή "Word". Αν δεν αληθεύει αυτό σημαίνει ότι δεν ορίζεται κάποια μεταβλητή (ή ορίζεται με μη έγκυρο όνομα, το οποίο σφάλμα καλύπτεται στην πορεία) και επιστρέφει 0.

```

399     currentWord = lex()
400
401     if currentType != "Word":
402         return 0 # No variables

```

Εφόσον η συνάρτηση συνεχίσει και δεν επιστρέψει, σημαίνει ότι έχουμε τουλάχιστον μια μεταβλητή. Το αν ακολουθεί κι άλλη μεταβλητή, θα φανεί απ' το αν η επόμενη τιμή που θα επιστρέψει ο lex() θα είναι κόμμα ",".

Λόγω του Kleene Star, θα χρειαστεί ένας βρόγχος με την χρήση while. Στην αρχή του while θα χρειαστεί να προσθέσουμε στην global λίστα varNames με το όνομα της νέας μεταβλητής, και μιας που είναι η τελευταία τιμή που επέστρεψε ο lex() είναι αποθηκευμένο

στο currentWord

```
406 while True:
407
408     varNames.append(currentWord)
```

Στην συνέχεια, κάνουμε τις εξής ενέργειες: Καλούμε τον lex() και ελέγχουμε αν η τιμή που επέστρεψε είναι “,”. Αν δεν είναι, σημαίνει ότι δεν υπάρχει άλλη μεταβλητή και επιστρέφουμε με 0. Αν είναι ξανακαλούμε τον lex() για να πάμε στην επόμενη λεκτική μονάδα, η οποία θα πρέπει να είναι όνομα μεταβλητής. Όπως και πριν ελέγχουμε για την εγκυρότητα του ονόματος με το αν το currentType είναι “word”

```
413 currentWord = lex()
414
415 if currentWord != ",":
416     return 0
417
418 currentWord = lex()
419
420 if currentType != "Word":
421     return 1
```

Έπειτα επιστρέφει στην αρχή του βρόγχου και ξανακάνει τα ίδια βήματα μέχρι α) είτε να μην βρει κόμμα μετά από μεταβλητή ή β) να βρει μη έγκυρο όνομα μεταβλητής.

-subprograms:

Η subprograms() λειτουργεί σαν μεσάζοντας μεταξύ της block() και των υποπρογραμμάτων

subprograms → (subprogram)*

Χρησιμοποιήσαμε και εδώ την μέθοδο της αναδρομής. Αρχικά καλείται η subprogram() και προφανώς επιστρέφεται 1 σε περίπτωση σφάλματος. Με παρόμοια λογική με πριν, μπορούμε να πούμε ότι δεν είναι υποχρεωτικό να γίνει αναδρομή, καθώς ενδέχεται να μην υπάρχει κάποιο υποπρόγραμμα. Αυτό εντοπίζεται εύκολα καθώς εκεί που ορίζονται τα υποπρογράμματα προηγούνται οι λέξεις “function” ή “procedure”. Επομένως αν η τιμή του currentWord είναι κάποια από αυτές τις λέξεις κάνει την αναδρομή, αλλιώς επιστρέφει με 0.

```

427         if subprogram() == 1:
428             return 1
429         currentWord = lex()
430
431         if currentWord == "function" or currentWord == "procedure":
432             if subprograms() == 1:
433                 return 1
434
435         return 0

```

-subprogram():

Στην συνάρτηση subprogram() ελέγχεται αν ορίζεται σωστά μια συνάρτηση ή διεργασία.

```

subprogram    →    function ID ( formalparlist )
                    block
                |
                procedure ID ( formalparlist )
                    block

```

Έχει γίνει ήδη ο έλεγχος στην subprograms() και ξέρουμε ότι έχει γραφτεί ορθά ένα εκ των “function” ή “procedure”. Πριν καλέσουμε τον lex() και αλλάξει τιμή το currentWord, θα αποθηκεύσουμε την τωρινή τιμή του στην τοπική μεταβλητή flag, διότι θα μας χρειαστεί σε επόμενη φάση. Με την κλήση του lex() έχουμε το όνομα του υποπρογράμματος, και ελέγχουμε με την χρήση της currentType αν είναι έγκυρο.

```

449         flag = currentWord
450
451         currentWord = lex()
452
453         if currentType != "Word":
454             return 1

```

-formalparlist:

Αυτός ο κανόνας αντιπροσωπεύει το σύνολο των παραμέτρων, όταν ορίζεται μια συνάρτηση ή διαδικασία

```

formalparlist → formalparitem
                ( , formalparitem ) *
                |
                ε

```

Η λογική για την υλοποίηση αυτής της συνάρτησης, είναι ακριβώς η ίδια με αυτή της varlist(). Δηλαδή αρχίζουμε καλώντας τον lex(), και στην συνέχεια ελέγχουμε αν υπάρχει τουλάχιστον μια παράμετρος. Οι παράμετροι στην Cimple ορίζονται με το πρόθεμα “in” ή

“inout”, οπότε αν η τιμή του `currentWord` δεν είναι κάποιο απ’ τις δυο αυτές λέξεις, σημαίνει ότι δεν υπάρχει παράμετρος και η `formalparitem()` επιστρέφει 0.

```
493     currentWord = lex()
494
495     if not (currentWord == "in" or currentWord == "inout"):
496         return 0 # No variables
```

Αν τελικά υπάρχει παράμετρος, δεν θα μπει στην διακλάδωση του `if` και θα συνεχίσει καλώντας την `formalparitem()` μαζί με την γνωστή διαδικασία

```
500     if formalparitem() == 1:
501         return 1
502
```

Τέλος, λόγω του Kleene Star δημιουργείτε ο βρόγχος `while` όπου αρχίζει με την κλήση του `lex()` και στην συνέχεια ελέγχει αν το `currentWord` έχει την τιμή “,”. Αν δεν την έχει, σημαίνει ότι δεν υπάρχουν άλλοι παράμετροι και επιστρέφει με 0. Αλλιώς συνεχίζει ξανά με κλήση του `lex()` και η καλεί την `formalparitem()`.

```
503     while True:
504
505
506         currentWord = lex()
507
508         if currentWord != ",":
509             return 0
510
511         currentWord = lex()
512
513         if formalparitem() == 1:
514             return 1
515
```

Έπειτα εφόσον επιστρέψει απ’ την `formalparitem()` με κάποια τιμή διάφορη του 1, θα συνεχίσει τον βρόγχο μέχρι την μη εμφάνιση “,” ή σφάλματος.

-`formalparitem`:

Σε αυτήν την συνάρτηση ελέγχεται μια παράμετρος αν ακολουθεί την δομή που χρειάζεται

```
formalparitem →    in ID
                  |    inout ID
```

Η συνάρτηση αυτή ακολουθεί μια σχετικά απλή διαδικασία, καθώς αρχικά ελέγχει αν το `currentWord` είναι "in" ή "inout" όπως και στον κανόνα της γραμματικής, και έπειτα καλεί τον `lex()`, και ελέγχει την ορθότητα του ονόματος με την χρήση του `currentType`

```
524     if not (currentWord == "in" or currentWord == "inout"):
525         expectation = "in or inout"
526         return 1
527
528     currentWord = lex()
529
530     if (lastWord == "inout"):
531         varNames = varNames + [currentWord]
532
533     if currentType != "Word":
534         return 1
```

Να σημειωθεί εδώ ότι στην γραμμή 530 γίνεται **λανθασμένα** η προσθήκη της παραμέτρου στην λίστα `varNames` που κρατά τα ονόματα των μεταβλητών, μόνο αν είναι με αναφορά. Αυτή η ενέργεια έχει ως αποτέλεσμα το πρόγραμμα να πετάει μήνυμα σφάλματος σε περίπτωση που γίνει ανάθεση τιμής (`x := 0`) σε παράμετρο με τιμή. Τι α κάνς τωρα

-blockstatements:

Αυτή η συνάρτηση υλοποιεί τον κανόνα όπου ελέγχεται αν διαχωρίζονται ορθά οι εντολές του κυρίως προγράμματος ή ενός υποπρογράμματος στην Cimple

blockstatements → **statement**
(; statement)*

Πάλι έχουμε παρόμοια λογική με το `varlist()` όπως και με το `formalparlist()`, με την διαφορά ότι στον κανόνα αυτόν δεν υπάρχει κενό σύμβολο, επομένως καλείται κατευθείαν η `statement()`, χωρίς να ελέγξουμε αν χρειάζεται να κληθεί.

```
556     if statement() == 1:
557         return 1
558
```

Γενικά, κάθε εντολή (`statement`) της Cimple θα πρέπει να καταλήγει σε ";", με εξαίρεση την τελευταία εντολή ενός block η οποία δεν είναι υποχρεωτικό να έχει ";" αλλά εφόσον είναι η τελευταία εντολή του block, σίγουρα θα ακολουθείται από "}".

Επομένως, στην συνέχεια ανοίγει ο βρόγχος `while` λόγω του Kleene Star στον κανόνα, και ελέγχουμε αν το `currentWord` ισούται με ";". Αν δεν ισχύει, σημαίνει ότι ή α) βρισκόμαστε στην τελευταία εντολή του block, ή β) είναι σφάλμα. Έπειτα ελέγχουμε αν το `currentWord` είναι "}" και αν δεν ισχύει, σημαίνει ότι είναι σφάλμα οπότε επιστρέφει με 1, αλλιώς είναι

Όπως φαίνεται και στον κανόνα της γραμματικής, θα πρέπει να ακολουθεί ή το σύμβολο "{" ή αλλιώς να καλέσουμε την συνάρτηση *statement*.

Άρα, με την χρήση της εντολής *if* ελέγχουμε αν το *currentWord* δεν είναι ίσο με "{", που σε αυτήν την περίπτωση καλούμε την συνάρτηση *statement()*. Έπειτα, σύμφωνα με τον κανόνα, μετά το *statement* πρέπει να ακολουθεί το σύμβολο ";", οπότε γίνεται ξανά έλεγχος με *currentWord*, και αν είναι διάφορο του ";" επιστρέφει 0, αλλιώς επιστρέφει 1.

```
582      if currentWord != '{':
583          if statement() == 1:
584              return 1
585
586      if currentWord != ';':
587          expectation = ";"
588          return 1
589      else:
590          return 0
```

Αν όμως δεν ισχύει η παραπάνω συνθήκη, τότε έχουμε βρει το σύμβολο "{" (άρα είμαστε στην κάτω περίπτωση του κανόνα), οπότε καλούμε τον *lex()* για να πάμε στο επόμενο και καλούμε την συνάρτηση *statement()*.

```
594      currentWord = lex()
595
596      if statement() == 1:
597          return 1
```

Έπειτα στον κανόνα ακολουθεί τερματικό σύμβολο και μεταβλητή μέσα σε Kleene Star, οπότε επόμενο είναι να ανοίξει βρόγχος *while*.

Μέσα στο *while* γίνεται πρώτα ο έλεγχος για το αν ακολουθεί ";". Όπως προαναφέραμε και εξηγήσαμε στην *blockstatements()*, κάθε *statement* θα ακολουθείτε είτε από ";", είτε από "}". Οπότε αν το *currentWord* δεν έχει για τιμή το ";", στην συνέχεια ελέγχεται αν έχει για τιμή το "}". Αν δεν ισχύει ούτε αυτό, τότε υπάρχει συντακτικό σφάλμα και επιστρέφει με 1,

αλλιώς επιστρέφει με 0.

```
599      while True:
600
601          if currentWord != ";":
602
603              if currentWord != "}":
604                  expectation = ";\\" or "\\}"
605                  return 1
606              return 0
```

Αν όμως τελικά ισούται με ";" δεν θα μπει στην παραπάνω διακλάδωση, και θα μεταβεί στο else, όπου αρχικά εκτελείται ο lex() για να πάρουμε την επόμενη λεκτική μονάδα. Εφόσον βρήκαμε το σύμβολο ";", δεν γνωρίζουμε αν ακολουθεί κι άλλο statement στην συνέχεια, οπότε ελέγχουμε αν το currentWord ισούται με "}", διότι αν ακολουθεί "}" προφανώς δεν υπάρχει άλλο statement και επιστρέφει με 0

```
608      else:
609          currentWord = lex()
610          if currentWord == "}":
611              return 0
```

Τέλος, εφόσον υπάρχει το ενδεχόμενο να ακολουθεί κι άλλο statement, καλούμε την συνάρτηση statement(), και στην συνέχεια επαναλαμβάνεται ο βρόγχος while από την αρχή

```
613      if statement() == 1:
614          return 1
```

-statement:

Ο κανόνας αυτός εκφράζει όλους του διαφορετικούς τρόπους με τους οποίους που μπορεί να γραφτεί ένα statement στην Cimple, δηλαδή όλες τις διαφορετικές εντολές.

statement	→	assignStat	
		ifStat	callStat
		whileStat	returnStat
		switchcaseStat	inputStat
		forcaseStat	printStat
		incaseStat	ε

Αφού λοιπόν φτάσαμε σε σημείο όπου βρέθηκε κάποιο statement, ήρθε η στιγμή να δούμε ποιο statement είναι συγκεκριμένα.

Αρχικά για την `assignStat`, που είναι η περίπτωση ανάθεσης τιμής (δηλαδή `x:= 0`), πρέπει η τωρινή λεκτική μονάδα, δηλαδή το `currentWord`, να είναι κάποια μεταβλητή του προγράμματος. Υπενθυμίζουμε ότι όλες αυτές οι μεταβλητές αποθηκεύονται στην global λίστα `varNames` για αυτόν ακριβώς τον σκοπό, οπότε ελέγχουμε αν το `currentWord` ανοίκει στο `varNames`, και αν ναι, καλούμε την `assignStat()`

```
if currentWord in varNames:
    if assignstat() == 1:
        return 1
```

Στις υπόλοιπες περιπτώσεις είναι πολύ πιο απλά τα πράγματα, καθώς για κάθε μια περίπτωση statement υπάρχει μια “λέξη κλειδί” που την χαρακτηρίζει

Στην περίπτωση του `ifStat` για παράδειγμα, η “λέξη κλειδί” είναι το “`if`”, οπότε αν το `current word` ισούται με “`if`”, καλείται η συνάρτηση `ifStat()`

```
625 elif currentWord == "if":
626     if ifstat() == 1:
627         return 1
```

Στα παρακάτω στιγμιότυπα φαίνεται αναλυτικά κάθε περίπτωση, ποια λέξη κλειδί χρειάζεται να βρει, και ποια συνάρτηση θα καλέσει, με τον ίδιο ακριβώς τρόπο όπως και στην περίπτωση της “`if`”.

```
629 elif currentWord == "while":
630     if whilestat() == 1:
631         return 1
632
633 elif currentWord == "switchcase":
634     if switchcasestat() == 1:
635         return 1
636
637 elif currentWord == "forcase":
638     if forcasestat() == 1:
639         return 1
640
641 elif currentWord == "incase":
642     if incasestat() == 1:
643         return 1
644
645 elif currentWord == "call":
646     if callstat() == 1:
647         return 1
648
649 elif currentWord == "return":
650     if returnstat() == 1:
651         return 1
652
653 elif currentWord == "input":
654     if inputstat() == 1:
655         return 1
656
657 elif currentWord == "print":
658     if printstat() == 1:
659         return 1
```

Στο τέλος, σε περίπτωση που δεν βρει καμία “λέξη κλειδί” σημαίνει ότι υπάρχει κάποιο συντακτικό λάθος, οπότε επιστρέφει 1. Εναλλακτικά, αν μπει αν σε κάποια απ’ τις διακλαδώσεις, αφού επιστρέψει η συνάρτηση που κάλεσε, η *statement()* θα επιστρέψει 0

```

661         else:
662             return 1
663
664     return 0
665

```

-assignStat:

Η *assignStat()* εκτελεί τον κανόνα του statement για την ανάθεση τιμής

assignStat → **ID := expression**

Η *statement()* έχει ήδη ελέγξει την μεταβλητή ID του κανόνα, οπότε τώρα η *assignStat()* πρέπει να ελέγξει το τερματικό σύμβολο ":=". Οπότε καλούμε τον *lex()* για να πάμε στην επόμενη λεκτική μονάδα, και ελέγχουμε αν η τιμή του *currentWord* ισούται με ":", αν όχι επιστρέφει με 1 για σφάλμα.

```

675     currentWord = lex()
676
677     if currentWord != "(":
678         expectation = ":"
679     return 1
680

```

Εφόσον βρέθηκε το ":", καλείτε ξανά ο *lex()*, και στην συνέχεια καλείται η συνάρτηση *expression()* που αντιστοιχεί στην ομώνυμη μεταβλητή του κανόνα της γραμματικής. Η μεταβλητή *E_place* θα χρειαστεί στο στάδιο του ενδιαμέσου κώδικα, οπότε την αγνοούμε προς το παρόν

```

681     currentWord = lex()
682
683     E_place = expression()
684     if E_place == 1:
685         return 1
686

```

-ifStat:

Σε αυτή την φάση ελέγχεται η περίπτωση όπου το statement είναι η εντολή διακλάδωσης if της Cimple.

ifStat → if (condition)
statements
elsepart

Ο έλεγχος για το τερματικό σύμβολο “if” έχει γίνει ήδη από την συνάρτηση που κάλεσε την ifStat(), οπότε καλούμε τον lex() και ελέγχουμε αν το currentWord ισούται με το επόμενο σύμβολο που είναι το “(”. Προφανώς επιστρέφει με 1 αν δεν ισχύει

```
698      currentWord = lex()
699
700      if currentWord != "(":
701          expectation = "("
702      return 1
```

Έπειτα στον κανόνα της γραμματικής εμφανίζεται η μεταβλητή condition, επομένως θα κληθεί πάλι ο lex() και στην συνέχεια θα κληθεί η συνάρτηση condition(). Η μεταβλητή condition_total θα αγνοηθεί προς το παρόν, όπως και γενικά οι μεταβλητές που παίρνουν την τιμή που επιστρέφει κάποια συνάρτηση θα αγνοούνται, καθώς θα επεξηγηθούν στα πλαίσια του ενδιαμέσου κώδικα.

```
704      currentWord = lex()
705
706      condition_total = condition()
707      if condition_total == 1:
708          return 1
```

Αφού επιστρέψει απ’ το condition(), θα χρειαστεί να ελεγχθεί αν το currentWord ισούται με “)”. Επίσης ο lex() δεν χρειάζεται να κληθεί εδώ καθώς η κλήση του έγινε μέσα στον condition()

```
713      if currentWord != ")":
714          expectation = ")"
715      return 1
```

Όπως φαίνεται και στον κανόνα, μετά απ’ το σύμβολο “)” ακολουθούν δύο μεταβλητές, επομένως θα χρειαστεί να κληθούν οι δύο συναρτήσεις statements() και elsepart()

```
721     currentWord = lex()
722     if statements() == 1:
723         return 1
```

Και προφανώς στο τέλος η συνάρτηση επιστρέφει με 0.

Η `elsepart` ελέγχει την διακλάδωση με την εντολή `else`, που αποτελεί συνέχεια της `if`

Η διαδικασία εδώ είναι σχετικά απλή καθώς ελέγχουμε μόνο ένα τερματικό σύμβολο και μια μεταβλητή. Υπάρχει όμως και το ενδεχόμενο του κενού συμβόλου, όπως φαίνεται και στον κανόνα. Οπότε αρχικά καλούμε τον `lex()` για να μεταβούμε στην επόμενη λεκτική μονάδα, και ελέγχουμε αν το `currentWord` είναι διάφορο του “else”. Αν είναι όντως διάφορο, τότε ισχύει το ενδεχόμενο με το κενό σύμβολο (δηλαδή δεν υπάρχει else μετά το if), οπότε επιστρέφει με 0. Αλλιώς αν δεν μπει στην διακλάδωση σημαίνει ότι υπάρχει “else” οπότε ξανακαλείται ο `lex()`.

Και αυτό που έμεινε είναι να κληθεί το `statements()`, και αν δεν εμφανιστεί σφάλμα, να επιστρέψει η συνάρτηση με 0

ΑΥΤΑ ΒΑΡΙΕΜΑΙ ΝΑ ΤΑ ΚΑΝΩ ΤΩΡΑ, ΣΚΙΠΑΡΩ ΚΑΙ ΠΑΩ ΣΤΟ ACTUALPARLIST()

Εδώ ελέγχεται η περίπτωση όπου το statement είναι η εντολή επανάληψης “while” της Cimple.

```
whileStat    →    while ( condition )
                        statements
```

Ο έλεγχος για το τερματικό σύμβολο “while” έχει γίνει ήδη από την συνάρτηση που κάλεσε την *whileStat()*, οπότε καλούμε τον *lex()* και ελέγχουμε αν το *currentWord* ισούται με το επόμενο σύμβολο που είναι το “(”. Προφανώς επιστρέφει 1 αν δεν ισχύει, τυπώνοντας το αντίστοιχο σφάλμα μέσω του *expectation*.

```

756         currentWord = lex()
760         if currentWord != "(":
761             expectation = "("
762             return 1

```

Στη συνέχεια, στον κανόνα της γραμματικής εμφανίζεται η μεταβλητή *condition*, επομένως θα κληθεί πάλι ο *lex()* και στη συνέχεια θα κληθεί η συνάρτηση *condition()*.

```
764     currentWord = lex()
765
766     condition_total = condition()
767     if condition_total == 1:
768         return 1
```

Αφού επιστρέψει απ' το condition(), θα χρειαστεί να ελέγξει αν το currentWord ισούται με "). Επίσης ο lex() δε χρειάζεται να κληθεί εδώ καθώς η κλήση του έγινε μέσα στον condition()

```
773     if currentWord != "":
774         expectation = ""
775         return 1
```

Όπως φαίνεται και στον κανόνα, μετά απ' το σύμβολο " $)$ " ακολουθεί η συνάρτηση *statements*, επομένως θα χρειαστεί να κληθεί κι η συνάρτηση *statements()*.

Άρα πρώτα εκτελείται ο lex() και στη συνέχεια καλείται η statements()

```
779     currentWord = lex()
780     if statements() == 1:
781         return 1
```

Τέλος, καλείται ο `lex()` για μια τελευταία φορά, ώστε να βγούμε από τη συνάρτηση ολοκληρωτικά, κι έπειτα επιστρέφουμε 0, για να μας γίνει γνωστό ότι η συνάρτηση "έκλεισε" με επιτυχία.

```

786         currentWord = lex()
787
788
789     return 0

```

-switchcaseStat:

Σ' αυτό το σημείο ελέγχεται η ορθότητα της εντολής "switchcase" του προγράμματος Cimple.

```

switchcaseStat →  switchcase
                  ( case ( condition ) statements ) *
                  default statements

```

Ως συνήθως, ο έλεγχος για "switchcase" έχει ήδη γίνει πριν εισέλθει το πρόγραμμα στη *switchcasestat()*.

Έτσι, καλείται ο *lex()* για να δούμε τι ακολουθεί:

```

798         currentWord = lex()

```

Υστερα λόγω του Kleene Star, εισερχόμαστε σε μια επανάληψη "while", όπως φαίνεται στην εικόνα:

```

800     while True:
801
802         if currentWord != "case":
803
804             if currentWord != "default":
805                 expectation = "default"
806             return 1

```

Αρχικά, ελέγχουμε την περίπτωση να απουσιάζει η λέξη "case", σε περίπτωση απουσίας της, περιμένουμε να δούμε "default", αλλιώς η σύνταξη της switchcase είναι εσφαλμένη κι επιστρέφει με 1.

```

808         currentWord = lex()
809         if statements() == 1:
810             return 1

```

Έπειτα, ξανακαλείται ο *lex()* κι η *statements()*, για να ελέγξουμε τη δικής της ορθότητα.

Σε περίπτωση που υπάρχει "case" καλείται ξανά ο *lex()* ώστε να ελέγξει αν υπάρχει "(".

```

815         currentWord = lex()
816
817     if currentWord != "(":
818         expectation = "("
819         return 1

```

Φυσικά, μετά από άνοιγμα παρένθεσης ελέγχουμε το condition, οπότε καλούμε την *condition()*.

```

821         currentWord = lex()
822
823         condition_total = condition()
824         if condition_total == 1:
825             return 1

```

Αφού επιστρέψει απ' το *condition()*, θα χρειαστεί να ελέγξει αν το *currentWord* ισούται με ")". Όπως και πριν, ο *lex()* δε χρειάζεται να κληθεί εδώ καθώς η κλήση του έγινε μέσα στον *condition()*.

```

830     if currentWord != ")":
831         expectation = ")"
832         return 1

```

Όπως φαίνεται και στον κανόνα, μετά απ' το σύμβολο ")" ακολουθεί η συνάρτηση *statements*, επομένως θα χρειαστεί να κληθεί κι η συνάρτηση *statements()*.

Στη συνέχεια, λοιπόν, καλείται πάλι ο *lex()*, καθώς κι η *statements()* προκειμένου να βρούμε τυχόν σφάλματα.

```

836         currentWord = lex()
837         if statements() == 1:
838             return 1

```

Το μόνο που απομένει είναι να κάνουμε ένα τελευταίο *lex()* ώστε να γνωρίζει η *block()* τι ακολουθεί και να λάβει τα κατάλληλα μέτρα.

```

846         currentWord = lex()

```

-forcasestat:

Εδώ γίνεται έλεγχος της ορθότητας της εντολής "forcasestat" του προγράμματος Cimple.


```
forcaseStat → forcase
               ( case ( condition ) statements ) *
               default statements
```

Όμοια με τη `switchcasestat()`, ξεκινάμε κάνοντας `lex()`, αποθηκεύοντας το αποτέλεσμα στη μεταβλητή `currentWord`.

```
853      currentWord = lex()
```

Ξανά, όπως και στη `switchcasestat()` και λόγω του Kleene Star, εισερχόμαστε σε μια επανάληψη “while”, όπως φαίνεται στην εικόνα:

```
858      while True:
859
860          if currentWord != "case":
861
862              if currentWord != "default":
863                  expectation = "default"
864                  return 1
```

Ελέγχουμε αν λείπει η λέξη “case”, και σε περίπτωση που ισχύει αυτό, γνωρίζουμε ότι μετά απαιτείται “default”, για το οποίο γίνεται κι έλεγχος, αλλιώς η σύνταξη της `switchcase` είναι εσφαλμένη και ξανά, επιστρέφει με 1, τυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
866      currentWord = lex()
867      if statements() == 1:
868          return 1
```

Ξανά όμοια με παραπάνω, ξανακαλείται ο `lex()` κι η `statements()`, για να ελέγξουμε τη δικής της ορθότητα.

Σε περίπτωση που υπάρχει “case” καλείται ξανά ο `lex()` προκειμένου να ελέγξει αν υπάρχει “(“.

```
873      currentWord = lex()
874
875      if currentWord != "(":
876          expectation = "("
877          return 1
```

Φυσικά, μετά από άνοιγμα παρένθεσης ελέγχουμε το `condition`, οπότε καλούμε την `condition()`, αφού φυσικά καλέσουμε άλλη μια φορά τον `lex()`, προκειμένου να προχωρήσουμε στο αρχείο.

```

879         currentWord = lex()
880
881         condition_total = condition()
882         if condition_total == 1:
883             return 1

```

Αφού επιστρέψει απ' το condition(), θα χρειαστεί να ελέγξει αν το currentWord ισούται με ")". Φυσικά, ο lex() δε χρειάζεται να κληθεί εδώ καθώς η κλήση του έγινε μέσα στον condition().

```

887         if currentWord != ")":
888             expectation = ")"
889             return 1

```

Όπως φαίνεται και στον κανόνα, μετά απ' το σύμβολο ")" ακολουθεί η συνάρτηση statements, επομένως θα χρειαστεί να κληθεί κι η συνάρτηση statements().

Άρα πρώτα εκτελείται ο lex() και στη συνέχεια καλείται η statements()

```

893         currentWord = lex()
894         if statements() == 1:
895             return 1

```

Το μόνο που απομένει είναι να κάνουμε ένα τελευταίο lex() ώστε να βγούμε από τη συνάρτηση.

```

903         currentWord = lex()

```

-incasestat:

Ψάχνουμε για πιθανά σφάλματα στη χρήση της εντολής "incasestat" του προγράμματος Cimple.

```

incaseStat    →    incase
                  ( case ( condition ) statements ) *

```

Όμοια με τις παραπάνω, ξεκινάμε κάνοντας lex(), αποθηκεύοντας το αποτέλεσμα στη μεταβλητή currentWord.

```

911         currentWord = lex()

```

Ξανά, όπως και παραπάνω και λόγω του Kleene Star, εισερχόμαστε σε μια επανάληψη "while", όπως φαίνεται στην εικόνα:

```

918     while True:
925         currentWord = lex()
926
927         if currentWord != "(":
928             expectation = "("
929             return 1

```

Ωστόσο, αυτή τη φορά δεν ελέγχουμε για την απουσία του “case” (τουλάχιστον όχι για έλεγχο ορθότητας) επειδή η “incase” δε γίνεται να έχει την τιμή “default”.

Οπότε ελέγχουμε για “(” απευθείας.

Τα υπόλοιπα είναι ίδια, έχουμε δηλαδή κάλεσμα των *lex()* και *condition()*, όπως φαίνεται παρακάτω:

```

931         currentWord = lex()
932
933         condition_total = condition()
934         if condition_total == 1:
935             return 1

```

Και φυσικά αφού επιστρέψει απ’ το *condition()*, θα χρειαστεί πάλι να ελέγξει αν το *currentWord* ισούται με “)”. Βέβαια, ο *lex()* δε χρειάζεται να κληθεί εδώ καθώς η κλήση του έγινε μέσα στον *condition()*.

```

940         if currentWord != ")":
941             expectation = ")"
942             return 1

```

Όπως φαίνεται και στον κανόνα, μετά απ’ το σύμβολο “)” ακολουθεί η συνάρτηση *statements*, επομένως θα χρειαστεί να κληθεί κι η συνάρτηση *statements()*.

Οπότε πρώτα εκτελείται ο *lex()* και στη συνέχεια καλείται η *statements()*

```

946         currentWord = lex()
947         if statements() == 1:
948             return 1

```

Και στο τέλος κάνουμε κι ένα τελευταίο *lex()* ώστε να βγούμε από τη συνάρτηση.

```

954     currentWord = lex()

```

-returnStat:

Έλεγχος για την ορθότητα της “returnstat”:

returnStat → **return(expression)**

Ξεκινάμε κάνοντας *lex()*, αφού έχει ήδη βρεθεί “return” και γι’ αυτό καλέστηκε η *returnstat()*. Αμέσως μετά, ελέγχουμε για “(“.

```
962         currentWord = lex()
963         if currentWord != '(':
964             expectation = "("
965         return 1
```

Στη συνέχεια, κάνουμε ακόμη μια φορά *lex()*, κι αφού τώρα φτάσαμε στο “expression”, καλούμε την αντίστοιχη ομώνυμη συνάρτηση *expression()*.

```
969         returnVal = expression()
970         if returnVal == 1:
971             return 1
```

Δεν καλούμε πάλι τον *lex()*, αφού το κάνει η *expression()* για μας, οπότε απευθείας ελέγχουμε για τον επόμενο χαρακτήρα, που περιμένουμε να είναι “)”

```
976         if currentWord != ')':
977             expectation = ")"
978         return 1
```

Τέλος, κάνουμε *lex()* άλλη μια τελευταία φορά, ενώ επιστρέφουμε και 0 απλά για να πούμε στο πρόγραμμα ότι βγήκε από τη συνάρτηση επιτυχώς.

```
980         currentWord = lex()
981
982         return 0
```

-callstat:

Εδώ φυσικά γίνεται έλεγχος για την ορθότητα της συνάρτησης *callstat*

callStat → **call ID(actualparlist)**

Όπως και στις υπόλοιπες συναρτήσεις, ξεκινάμε κάνοντας ένα *lex()*, με σκοπό να προχωρήσουμε στο πρόγραμμα, αφού έχουμε ήδη βρει την εντολή “call”.

```
991         currentWord = lex()
992
993         if currentType != "Word":
994             return 1
```

Με την κλήση του *lex()* εδώ θα προκύψει το όνομα του υποπρογράμματος, οπότε ελέγχουμε μέσω του *currentType* αν είναι έγκυρο.

```

996     check = searchEntity(currentWord)
997     if check == -1:
998         print("Error in line " + str(line) + ": No function found named \"" + currentWord + "\"")
999         exit()
1000     elif check[0].__class__.__name__ != "Procedure":
1001         print("Error in line " + str(line) + ": Procedure type was expected after 'call'. \"" + currentWord + "\" is \"" + check[0].__class__.__name__ + " type")
1002         exit()

```

Αυτά σόρι, αλλά μάλλον θα χρειαστεί να τα εξηγήσεις εσύ... #Βασίλης_OP

Μετά, καλούμε πάλι τον *lex()*, με σκοπό να δούμε αν ανοίγει παρένθεση, οπότε ψάχνουμε για "("

```

1006         currentWord = lex()
1007         if currentWord != '(':
1008             expectation = "("
1009             return 1

```

Στη συνέχεια, καλούμε την *actualparlist()*, για να ελεγχθούν όλες οι παράμετροι.

```

1011         if actualparlist() == 1:
1012             return 1

```

Αμέσως μετά, ελέγχουμε το *currentWord* για ")", χωρίς νέα κλήση του *lex()*, αφού ξέρουμε ότι αυτό έχει φροντίσει να το κάνει η *actualparlist()*.

```

1014         if currentWord != ')':
1015             expectation = ")"
1016             return 1

```

Τέλος, καλούμε μια τελευταία φορά τον *lex()* κι επιστρέφουμε 0 για να ξέρουμε ότι η συνάρτηση τερμάτισε σωστά.

```

1022         currentWord = lex()
1023
1024         return 0

```

-printStat:

Εδώ θα ελεγχθεί η ορθότητα της "print" της γλώσσας Cimple.

printStat → **print(expression)**

Όπως στις περισσότερες συναρτήσεις, έτσι κι εδώ ελέγχουμε για "(", αφού κάνουμε πρώτα ένα *lex()*, για να πάμε στην επόμενη λεκτική μονάδα.

```

1032         currentWord = lex()
1033         if currentWord != '(':
1034             expectation = "("
1035             return 1

```

Στη συνέχεια, ακολουθώντας την παραπάνω εικόνα, για το μοντέλο της *printStat* κι αφού καλέσουμε τον *lex()* ακόμη μια φορά, καλούμε την *expression()*, για να ελέγξουμε τα περιεχόμενα της παρένθεσης.

```

1037         currentWord = lex()
1038
1039         E_place = expression()
1040         if E_place == 1:
1041             return 1

```

Φυσικά, σε περίπτωση σφάλματος η συνάρτηση θα επιστρέψει 1.

Η expression θα καλέσει τον lex(), οπότε αμέσως μετά ελέγχουμε για “)”

```

1043         if currentWord != ')':
1044             expectation = ")"
1045             return 1

```

Ξανά, καλούμε τελευταία φορά τον lex() πριν επιστρέψουμε 0 και βγούμε απ’ τη συνάρτηση.

```

1050         currentWord = lex()
1051         return 0

```

-inputStat:

Ψάχνουμε για τυχόν σφάλματα στη δομή της inputstat:

inputStat → input(ID)

Καταρχάς, το πρόγραμμα ελέγχει αν ακολουθεί “(”, έχοντας φυσικά καλέσει τον lex() πρώτα, ώστε να προχωρήσει στο πρόγραμμα.

```

1059         currentWord = lex()
1060         if currentWord != '(':
1061             expectation = "("
1062             return 1

```

Μετά, καλεί τον lex() ξανά προκειμένου να ελέγξει αν μέσα στην παρένθεση υπάρχει κάποια “ανωμαλία”, δηλαδή δεν είναι λέξη. Αυτό επιτυγχάνεται με το *currentType*:

```

1064         currentWord = lex()
1065
1066         if currentType != "Word":
1067             return 1

```

```

1068         check = searchEntity(currentWord)
1069         if check == -1:
1070             print("Error in line " + str(line) + ": No variable found named \"" + currentWord + "\"")
1071             exit()
1072         elif check[0].__class__.__name__ != "Variable" and check[0].__class__.__name__ != "Parameter":
1073             print("Error in line " + str(line) + ": Variable type was expected after 'call'. \"" + currentWord + "\" is \"" + check[0].__class__.__name__ + " type")
1074             exit()
1075

```

Whatever that is. #Βασίλης_OP

Χεστο αυτό, έχει να κάνει με πίνακα συμβόλων, το ξηγούμε κάτω

Το επόμενο βήμα είναι ο έλεγχος για “)”, αφού προηγηθεί ένας `lex()`, αφού δε μεσολαβεί κάποια άλλη συνάρτηση γι’ αυτό.

```
1079      currentWord = lex()
1080
1081      if currentWord != ')':
1082          expectation = ")"
1083          return 1
```

Τέλος, κλείνουμε με `lex()` και “return 0” για τους συνήθεις λόγους.

```
1089      currentWord = lex()
1090      return 0
```

-actualparlist:

Η `actualparlist` κάνει τον έλεγχο για την σωστή σειρά και διαχωρισμό των παραμέτρων. Θεωρητικά, η διαφορά της με την `formalparlist()`, είναι ότι εδώ αναφερόμαστε στην περίπτωση όπου καλείται μια συνάρτηση, και όχι όταν ορίζεται. Στο συντακτικό κομμάτι όμως δεν έχει κάποια διαφορά.

`actualparlist` → `actualparitem`
 `(, actualparitem)*`
 |
 ϵ

Οπότε, όπως και στην `formalparitem()`, έτσι κι εδώ αρχικά κοιτάμε αν υπάρχει τουλάχιστον μια παράμετρος, το οποίο φαίνεται με την εμφάνιση ενός εκ των “in” και “inout”. Οπότε αφού κληθεί ο `lex()` ελέγχεται αν το `currentWord` ισούται με κάποια απ’ τις δυο αυτές λέξεις. Το if ελέγχει την περίπτωση που δεν ισχύει αυτό, δηλαδή δεν υπάρχει παράμετρος, άρα επιστρέφει με 0

```
1103      currentWord = lex()
1104
1105      if not (currentWord == "in" or currentWord == "inout"):
1106          return 0 # No variable
```

Εφόσον τελικά υπάρχει μια παράμετρος, στην καλείται η `actualparitem()` που αντιπροσωπεύει την μεταβλητή `actualparitem` της γραμματικής

```
1115      parTotal = actualparitem()
1116      if parTotal == 1:
1117          return 1
```

Παρακάτω στην γραμματική υπάρχει το Kleene Star οπότε προφανώς θα χρειαστεί να ξεκινήσει βρόγχος while. Για να υπάρχει και επόμενη παράμετρος, χρειάζεται να ακολουθεί κόμμα (“,”), οπότε ελέγχουμε αν το `currentWord` είναι διάφορο του “,”, και αν ισχύει επιστρέφει με 0. Οι υπόλοιπες εντολές που φαίνονται μέσα στην διακλάδωση αφορούν τον

ενδιάμεσο κώδικα και θα εξηγηθούν όταν χρειαστεί.

```
1126     while True:
1127
1128         if currentWord != ",":
1129
1130             if len(thisFunc[0].formalParameters) != 1:
1131
1132                 for i in range(0, len(parNamesList)):...
```

Ένα δεν μπει στον βρόγχο if, σημαίνει ότι υπάρχει ",", άρα σύμφωνα με την γραμματική θα χρειαστεί να κληθεί ο `lex()` και στην συνέχεια η `actualparitem()`. Έπειτα επιστρέφει στην αρχή του βρόγχου

```
1144         currentWord = lex()
1145
1146         parTotal = actualparitem()
1147         if parTotal == 1:
1148             return 1
```

-actualparitem:

```
actualparitem → in expression
               | inout ID
```

-condition:

```
condition → boolterm
           ( or boolterm )*
```

-boolterm:

`boolterm` \rightarrow `boolfactor`
 `(and boolfactor)*`

-boolfactor:

`boolfactor` \rightarrow `not [condition]`
 $|$ `[condition]`
 $|$ `expression REL_OP expression`

-expression:

`expression` \rightarrow `optionalSign term`
 `(ADD_OP term)*`

-term:




`term` \rightarrow `factor`
 `(MUL_OP factor)*`

-factor:

`factor` \rightarrow `INTEGER`
 $|$ `(expression)`
 $|$ `ID idtail`

-idtail:

`idtail` \rightarrow `(actualparlist)`
 $|$ ϵ

ΕΛΑ Τι άλλο κάνω; Χμμμμ   



<https://youtu.be/dmiyE6y0Oyl>



Μισο να δω και την αλλη✓

Ναι

Το blocknames είναι πάλι για τον ενδιαμεσο, αλλα για την εντολή begin block

Άρα curentcalling όταν call bluh

Blocknames όταν begin_block bluh



Κομπλέ, μην τα σβήνεις αν είναι και θα το συνεχίσω αύριο



Δεν σβηνω :harí: τι ωραια που εχουμε νιτρο εδώ



Ελα Έλα

Λοιπον

Λεω να ερθω κι εγω στον ενδιαμεσο

Το θεμα είναι ότι άφησα καποια από πανω που δεν τα εκανα

Τα πάνω είναι από 1η φάση;

Ναι, απλα νιωθω ότι γραφω τα ιδια συνεχεια, γιατι επαναλαμβανονται

Αν νομιζεις οτι μπορεις να το συνεχισεις feel free to go

Κοίτα. Σίγουρα θα ξέρω περισσότερα από ενδιάμεσο

Τελεια

Κοιτα τι εχω κανει μεχρι τωρα και πως τα λεω
Στην ουσηα μιμούμαι το λέγγην του Μανη
Οποτε δοκίμασε να κανεις το ιδιο καπως
Θα κοιτάξω απ' τα δικά σου και λογικά θα προσαρμοστώ
Α μπραβο δεν αντεχα άλλο
Με αυτά τα..... βαρετα πραγματα
ΒΑΡΕΤΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑΑ



:jjjlaugh: Έγινε ρε συ

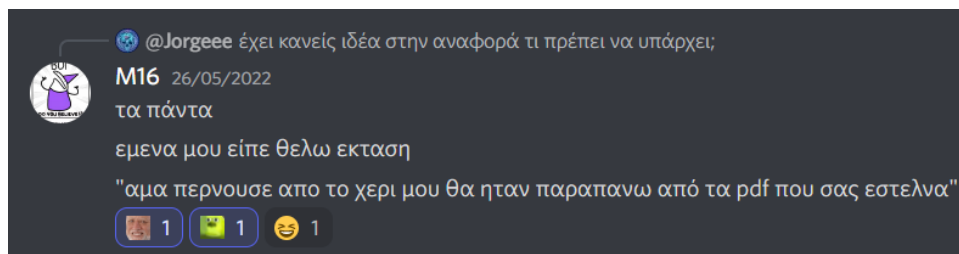
Αχχ το word παιζει να είναι το αγαπημενο πραμα για chat, παρ' όλο που δεν ειναι καν
αυτος ο σκοπος -ναι ξεκάθαρα

Τεσπα

Γενικα αν κολλάς κάπου στα θεωρητικά, αστα και θα τα συμπληρώσω μετα

Ελα Έλα **ELA**

Μισο, να βρω ακριβως το μήνυμα πως το ειπε



ELAELELA Γενικα νομιζω είναι η βαση αυτές οι συναρτήσεις. Όσα πιο πολλά γράψουμε
τοσο καλύτερα παντως. Αρκει να εξηγούμε πράγματα και να μην ξεφεύγουμε

Οκε, κομπλέ

ELA Μπορείς να τσεκάρεις μια να δεις αν είναι ελλιπείς;

Οκ παμε απανω Yeh

Καλα το πας. :hari:

Εγω απλα ειχα μεγαλυτερες εικονες. Δηλαδη σε μια εικονα ειχα πιο πολύ κωδικα και δεν
τον εσπαγα τοσο. Αλλα δεν τρεχει κατι Τρέχει μόνο η γλώσσα μας;



Αλλά ναι, έπρεπε να το κόψω επειδή είχε backpatch και τέτοια :sadge:

δεν χωρεύει :sad:



Ναι, ok όπως νομίζεις, σιγά, κομπλέ. Let's get back to it. Ikuzo

3) Ενδιάμεσος Κώδικας

Ο ενδιάμεσος κώδικας παράγεται κατά την διάρκεια της διαδικασίας του συντακτικού αναλυτή. Δηλαδή οι εντολές ή οι συναρτήσεις που είναι υπεύθυνες για την παραγωγή του, καλούνται μέσα στις συναρτήσεις του συντακτικού αναλυτή.

Για την υλοποίησή του χρησιμοποιήθηκαν κυρίως οι βοηθητικές συναρτήσεις (υπορουτίνες) που δόθηκαν στις διαφάνειες του μαθήματος, καθώς και κάποιες νέες μεταβλητές για να αποθηκεύουν ορισμένα δεδομένα.

Όπως είναι ήδη γνωστό, ο ενδιάμεσος κώδικας είναι στην ουσία μια σειρά από τετράδες. Για την αποθήκευση των τετράδων αυτών δημιουργήθηκε η κλάση Quad, με τέσσερα πεδία, που αντιστοιχούν στα τέσσερα στοιχεία της τετράδας.

```
8 class Quad:
9     def __init__(self, operator, operand1, operand2, operand3):...
```

Εφόσον έχουμε την δομή, θα χρειαστούμε και μια λίστα για να αποθηκεύει συνολικά όλες τις τετράδες. Αυτή θα είναι η global λίστα quadz.

```
69 quadz = [] # here we save the quadz
```

Όπως είναι προφανές, εκεί θα αποθηκεύονται όλες οι τετράδες όσο τρέχει το πρόγραμμα, με απώτερο σκοπό να τις πάρουμε από αυτήν την λίστα και να γραφτούν σε αρχείο με κατάληξη `.int`.

Επιπλέον, ορίζουμε τις εξής μεταβλητές:

```
101 t_name = 0
102 labelslist = []
103 blocknames = []
104 currentCalling = []
105
106 flagForCfile = 0
```

- **t_name:** Υπάρχει το ενδεχόμενο να χρειαστεί να δημιουργηθούν κάποιες προσωρινές μεταβλητές στον ενδιάμεσο κώδικα (T_1, T_2 κτλ). Στην t_name λοιπόν κρατάμε τον αριθμό της τελευταίας προσωρινής μεταβλητής του ενδιάμεσου κώδικα. Αν πχ η τελευταία προσωρινή μεταβλητή ήταν η T_3, το t_name θα έχει την τιμή t_name = 3
- **labelslist:** Κάθε τετράδα πρέπει να αντιστοιχεί σε μια ετικέτα, οπότε η labelslist είναι η λίστα όπου αποθηκεύονται οι ετικέτες που αντιστοιχούν στις τετράδες του

quadz. Κάθε ετικέτα αντιστοιχεί στην τετράδα που βρίσκεται στην ίδια θέση με αυτήν στον *quadz*. Πχ η *labelslist[i]* είναι η ετικέτα της τετράδας *quadz[i]*

- **blocknames**: Αποθηκεύει τα ονόματα των block με τη σειρά που εμφανίζονται, ώστε να γνωρίζουμε σε ποιο βρισκόμαστε και να τα τυπώνουμε με ευκολία στον πίνακα συμβόλων ως “begin_block” και μετά το όνομα.
- **currentCalling**: Κρατάει την τωρινή συνάρτηση που καλέστηκε για να δημιουργηθεί η κατάλληλη τετράδα με operator “call” και το όνομα αυτής.
- **flagForCFile**: Στο κάποια φάση της εργασίας χρειάζεται να δημιουργηθεί ένα αρχείο που θα έχει μετατρέψει τον ενδιαμέσο κώδικα σε γλώσσα C. Το αρχείο αυτό όμως θα δημιουργηθεί μόνο αν δεν υπάρχουν υποπρογράμματα στον κώδικα, οπότε Η μεταβλητή *flagForCFile* λειτουργεί σαν flag που χρησιμοποιείται για την αναγνώριση δημιουργίας αρχείου σε γλώσσα C. Αν το flag έχει την τιμή 1, τότε δεν θα δημιουργηθεί το αρχείο αυτό, ενώ αν είναι 0, τότε θα δημιουργηθεί. Και στις δύο περιπτώσεις θα τυπωθεί αντίστοιχο μήνυμα.

Αφού είδαμε τις μεταβλητές, ας δούμε τώρα τις καινούργιες συναρτήσεις αναλυτικά:

nextQuad() : Επιστρέφει το label της επόμενης τετράδας που θα παραχθεί. Αυτό επιτυγχάνεται όπως φαίνεται στην εικόνα:

```
1496 def nextQuad():
1497     return len(quadz) + 1
```

Στην ουσία το label είναι ο αριθμός της τετράδας που παράχθηκε, όπου η πρώτη τετράδα θα έχει label = 1, η δεύτερη label = 2 κ.ο.κ. Άρα η επόμενη τετράδα που θα παραχθεί θα έχει για label το μήκος της λίστας *quadz* που κρατάει όλες τις τετράδες, συν ένα.

genQuad(operator, operand1, operand2, operand3) : Φτιάχνει μια τετράδα ως κλάση Quad με τις 4 τιμές που παίρνει ως ορίσματα και την προσθέτει στο τέλος της λίστας *quadz* με την εντολή “append” ως εξής:

```
1492 def genQuad(operator, operand1, operand2, operand3):
1493     quadz.append(Quad(operator, operand1, operand2, operand3))
1494     return 0
```

Με το “return 0” απλώς σιγουρευόμαστε ότι τερματίζει η συνάρτηση.

Τα ορίσματα της συνάρτησης, operator, operand1, operand2 και operand3 θα εξηγηθούν παρακάτω.

newTemp() : Η *newTemp()* εκτελεί τις δυο εξής λειτουργίες:

1. Δημιουργεί μια νέα προσωρινή μεταβλητή, ανάλογα με το όνομα της προηγούμενης. Υπενθυμίζουμε ότι μια προσωρινή μεταβλητή στον ενδιαμέσο κώδικα έχει την μορφή *T_x*, όπου x κάποιος ακέραιος αριθμός. Για το όνομα της νέας μεταβλητής χρησιμοποιείται η *t_name* στην οποία αναφερθήκαμε παραπάνω, όπου αν η *t_name* έχει για παράδειγμα την τιμή 3, σημαίνει ότι η προηγούμενη προσωρινή μεταβλητή ήταν η *T_3*, άρα τώρα θα δημιουργήσει την *T_4*
2. Επιστρέφει τη μεταβλητή που μόλις δημιούργησε, η οποία θα προστεθεί αργότερα σε τετράδα μέσω του *genQuad()*.

```

1499 def newTemp():
1500     global t_name
1501     global totaloffset
1502
1503     t_name = t_name + 1
1504     totaloffset = totaloffset + 4
1505     addNewRegister("TemporaryVariable", "T_" + str(t_name), totaloffset)
1506     return "T_" + str(t_name)

```

Η μεταβλητή **totaloffset**, καθώς κι η συνάρτηση **addNewRegister()** θα εξηγηθούν αργότερα.

emptyList(): Όπως φαίνεται κι απ' το όνομά της, δημιουργεί μια κενή λίστα και μετά την επιστρέφει, (προκειμένου να τη γεμίσουμε μετά με το *genQuad()*)???

```

1508 def emptyList():
1509     definitelyEmptyList = []
1510     return definitelyEmptyList

```

makeList(label): Παίρνει σαν όρισμα την παράμετρο label, και δημιουργεί μια λίστα όπου η label θα είναι το μόνο της στοιχείο.

```

1512 def makeList(label):
1513     definitelyLabel = [label]
1514     return definitelyLabel

```

mergeList(list1, list2): Δημιουργεί μια νέα λίστα ενώνοντας τις λίστες list1 και list2. Έπειτα την επιστρέφει.

```

1516 def mergeList(list1, list2):
1517     mergedList = list1 + list2
1518     return mergedList

```

backpatch(list,label): Η συνάρτηση *backpatch()* υλοποιείται ως εξής:

Η λίστα list, που παίρνει σαν όρισμα η συνάρτηση, αποτελείται από labels που αντιστοιχούν σε τετράδες τύπου Quad που είναι στοιχεία της global λίστας quadz, οι οποίες έχουν ασυμπλήρωτο το τελευταίο πεδίο τους. Ύστερα αναζητάει μια προς μια τις τετράδες αυτές στην quadz, και εκεί βάζει στο τελευταίο πεδίο τους την τιμή του ορίσματος label.

```

1520 def backpatch(list, label):
1521     global labelslist, quadz
1522
1523     counter = 0
1524     for i in quadz:
1525         for j in list:
1526             if labelslist[counter] == j:
1527                 i.operand3 = label
1528                 counter = counter + 1
1529     return 0

```

Πρώτα αρχικοποιούμε τη μεταβλητή *counter* στο 0. Η μεταβλητή Αυτή θα κρατάει την θέση του στοιχείου της *quadz* που θα θέλουμε να μεταβάλλουμε.

Αυτό που θέλουμε τώρα είναι να βρούμε μέσα στην *quadz*, εκείνες τις τετράδες των οποίων τα *labels* υπάρχουν μέσα στο όρισμα *list* της συνάρτησης. Με τη πρώτη “for” διατρέχουμε τη λίστα *quadz*, και για κάθε στοιχείο της, διατρέχουμε την *list*.

Η *labelslist* είναι μια λίστα που κρατάει όλες τις ετικέτες στην κατάλληλη θέση για να γνωρίζουμε ακριβώς το σημείο που θα βρούμε την τετράδα της *quadz* που χρειαζόμαστε κάθε φορά.

Φυσικά, προκειμένου να δουλεύει σωστά το πρόγραμμα, οι παραπάνω συναρτήσεις καλούνται σε πολλές από τις προηγούμενες συναρτήσεις.

Έτσι, έχουμε:

Στη *block()*:

```

335     labelslist = labelslist + makeList(nextQuad())
336     genQuad('begin_block', blocknames[-1], '_', '_')

```

Γραμμή 335: Προστίθεται στη *labelslist* η επόμενη ετικέτα.

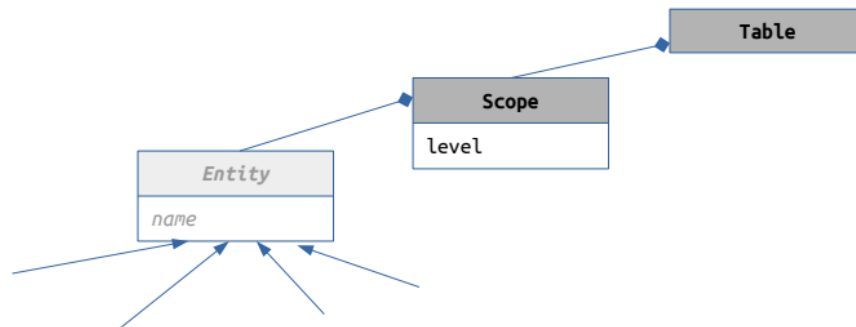
Γραμμή 336: Δημιουργείται τετράδα με operator “begin_block” που υποδεικνύει την αρχή ενός block, operand1 **blocknames**[-1], που είναι το τελευταίο στοιχείο του *blocknames*, το οποίο κρατάει τα ονόματα των blocks, operand2 και 3 “_”, αφού δεν υπάρχει κάποια πληροφορία να προστεθεί εκεί προς το παρόν.

Αυτό θα μπει πιο μετά, γι’ αυτό το έβαλε με χρώμα πεθαμενί

4) Πίνακας συμβόλων

Ο πίνακας συμβόλων αποθηκεύει τα ονόματα μεταβλητών, συναρτήσεων κτλ καθώς και ορισμένες πληροφορίες σχετικά με αυτά

Είναι γνωστό από την θεωρία ότι ο πίνακας συμβόλων χωρίζεται σε επίπεδα, όπου κάθε επίπεδο αποτελείται από διάφορες οντότητες, όπως φαίνεται στο παρακάτω διάγραμμα



Για να υλοποιηθεί αυτό σε κώδικα, θα χρησιμοποιήσουμε τις κλάσεις της python.

Για τον πίνακα, θα ορίσουμε την κλάση Table

```
17 class Table:
18     def __init__(self, scopes):
19         self.scopes = scopes
```

Όπου, όπως φαίνεται και στο διάγραμμα, χρειάζεται να έχει ως πεδίο μόνο την λίστα scopes όπου θα κρατάει όλα τα επίπεδα

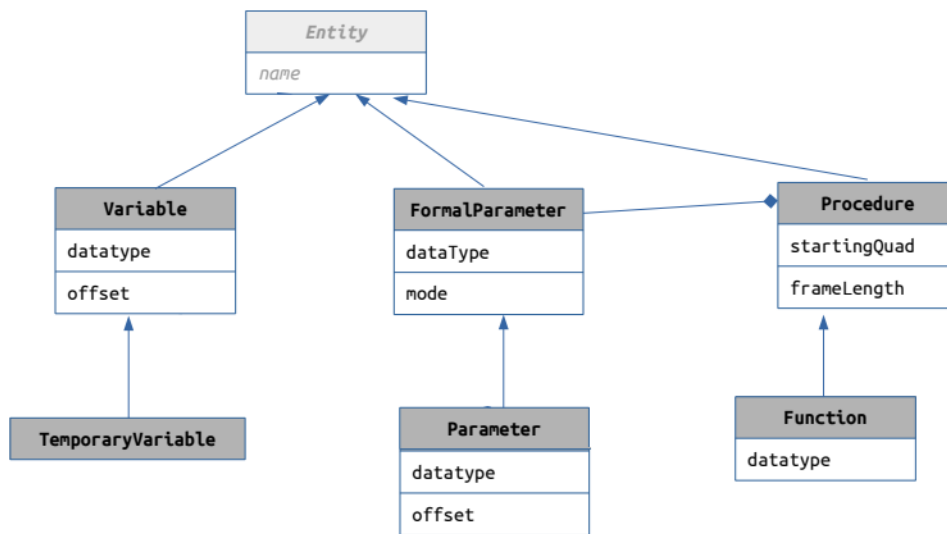
Το κάθε επίπεδο θα είναι μια κλάση της μορφής Scope, με πεδία την μεταβλητή level όπου θα κρατάει τον αριθμό του επιπέδου, και την λίστα entities που θα έχει μέσα οντότητες

```
21 class Scope:
22     def __init__(self, level, entities):
23         self.level = level
24         self.entities = entities
```

Οι οντότητες με την σειρά τους θα υλοποιούνται κι αυτές από μια κλάση που ονομάζεται Entity, και θα έχει ως πεδίο την μεταβλητή name που θα κρατάει το όνομα της οντότητας

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


Όπως είναι λογικό θα υπάρχουν πολλών ειδών οντότητες, και οι συσχετίσεις τους μπορούν να φανούν στο παρακάτω γράφημα



Όπως φαίνεται απ' το γράφημα, όλες οι οντότητες κληρονομούν έμεσα ή άμεσα απ' την κλάση **Entity**, επομένως όλες θα έχουν πεδίο `name`.

Ας τις αναλύσουμε μια προς μια

-Variable:

Μια απ' την πιο βασικές οντότητες προφανώς θα είναι η μεταβλητή, οποία θα υλοποιείται με την κλάση **Variable**

```
30 class Variable(Entity):
31     def __init__(self, name, datatype, offset):
32         super().__init__(name)
33         self.datatype = datatype
34         self.offset = offset
```

Όπως βλέπουμε και στην εικόνα, κληρονομεί το πεδίο `name` απ' το **Entity**, και έχει άλλα δυο έξτρα πεδία, το `datatype` και το `offset`. Το **datatype** κανονικά κρατάει το είδος της μεταβλητής, πχ αν είναι `integer`, `double` κτλ. Στην **Cimple** όλες οι μεταβλητές είναι τύπου `integer` επομένως το `datatype` είναι περιττό, αλλά το κρατήσαμε για τυπικούς λόγους. Το **offset** κρατάει την απόσταση της μεταβλητής απ' την αρχή του εγγραφήματος δραστηριοποίησης, το οποίο θα εξηγηθεί παρακάτω.

-TemporaryVariable:

Η κλάση αυτή αντιπροσωπεύει τις προσωρινές μεταβλητές, των οποίων η φύση είναι ακριβώς ίδια με τις κανονικές μεταβλητές, αλλά χρειάζεται να τις έχουμε χωριστά

```
59 class TemporaryVariable(Variable):
60     def __init__(self, name, datatype, offset):
61         super().__init__(name, datatype, offset)
```

Όπως φαίνεται κληρονομεί όλα τα πεδία της κλάσης Variable

-Procedure:

Σε αυτήν την κλάση υλοποιείται η οντότητα της διαδικασίας. Όπως όλες οι οντότητες, έτσι και αυτή κληρονομεί το πεδίο name απ' το Entity.

```
class Procedure(Entity):
    def __init__(self, name, startingQuad, framelength, formalParameters):
        super().__init__(name)
        self.startingQuad = startingQuad
        self.framelength = framelength
        self.formalParameters = formalParameters
```

Σχετικά με τα υπόλοιπα πεδία, έχουμε την μεταβλητή **startingQuad**, η οποία κρατάει τον αριθμό του label της πρώτης εντολής της διαδικασίας. Το **framelength**, που κρατάει το μήκος σε bytes της διαδικασίας του οποίου η χρήση και η σημασία θα φανεί καλύτερα στην πορεία, και τέλος, την λίστα **formalParameters**, που είναι αποθηκευμένες όλες οι παράμετροι της διαδικασίας σε μορφή μιας κλάσης που θα αναλυθεί λίγο παρακάτω.

-Function:

Η κλάση Function υλοποιεί την οντότητα της συνάρτησης και κληρονομεί τα στοιχεία της Procedure.

```
54 class Function(Procedure):
55     def __init__(self, name, datatype, startingQuad, framelength, formalParameters):
56         super().__init__(name, startingQuad, framelength, formalParameters)
57         self.datatype = datatype
58
```

Απ' ότι φαίνεται στον κώδικα, δεν χρησιμοποιούμε πουθενά το FormalParameter, αλλά μόνο το Parameter σκέτο. Εδώ είναι μια στιγμή παραφροσύνης

```
36 class FormalParameter(Entity):
37     def __init__(self, name, datatype, mode):
38         super().__init__(name)
39         self.datatype = datatype
40         self.mode = mode # in/ref/ret
```

```

42 class Parameter(FormalParameter):
43     def __init__(self, name, datatype, mode, offset):
44         super().__init__(name, datatype, mode)
45         self.offset = offset

```

Αφού έχουμε εξηγήσει πως υλοποιούνται οι οντότητες σε κλάσεις, ας δούμε μερικές global μεταβλητές που αφορούν τον ενδιάμεσο

```

109 myTable = Table([])
110 totaloffset = 0
111 offsetCheckPoint = []

```

- **myTable:** Αυτή η μεταβλητή θα κρατάει τον πίνακα συμβόλων του προγράμματος. Αρχικοποιείται ως κλάση τύπου Table και παίρνει σαν όρισμα μια κενή λίστα για το πεδίο scores, διότι δεν έχει δημιουργηθεί ακόμη κάποιο επίπεδο
- **totaloffset:** Εδώ θα κρατάμε το μήκος σε bytes του εγγραφήματος δραστηριοποίησης, έτσι ώστε αν για παράδειγμα εμφανιστεί μια νέα μεταβλητή, το totaloffset θα έχει αποθηκευμένο το offset της προηγούμενης, και θα μπορούμε να υπολογίσουμε το νέο offset προσθέτοντας 4.
- **offsetCheckPoint:** Είναι μια λίστα που λειτουργεί σαν LIFO ουρά. Όταν καλείται κάποια συνάρτηση ή διαδικασία δημιουργείτε ένα νέο επίπεδο, και θα πρέπει κάπως να γνωρίσουμε ποιο ήταν το offset για όταν επιστρέψουμε. Έτσι, πριν κάθε δημιουργία του νέου επιπέδου, το offsetCheckPoint προσθέτει την τιμή του offset, και με την διαγραφή του νέου επιπέδου την αφαιρεί.

Προχωράμε λοιπόν στις συναρτήσεις

`addNewRegister(entityType, name, extraOp):` τι χρωμα είναι αυτο γτστ

Όπως λέει και το όνομα, ο ρόλος της συνάρτησης αυτής είναι να δημιουργήσει μια καινούργια οντότητα και να την προσθέσει στον πίνακα συμβόλων, στο τελευταίο επίπεδο.

Με άλλα λόγια, να φτιάξει μια οντότητα τύπου μιας συγκεκριμένης κλάσης, και να πάει στο myTable (που όπως είπαμε είναι τύπου κλάση Table), και συγκεκριμένα στην λίστα scores που την έχει ως πεδίο, στο τελευταίο στοιχείο της λίστας (που είναι κλάση Score), το οποίο με την σειρά του έχει την λίστα entities ως πεδίο, όπου θα προστεθεί η νέα οντότητα.

Bluh bluh bluh κατι κατι νατο

```

def addNewRegister(entityType, name, extraOp):
    global myTable
    if entityType == "Variable":
        myTable.scopes[-1].entities.append(Variable(name, "Int", extraOp))
    elif entityType == "FormalParameter":
        myTable.scopes[-1].entities.append(FormalParameter(name, "Int", extraOp))
    elif entityType == "Procedure":
        # name, startingQuad, framelength, formalParameters
        myTable.scopes[-1].entities.append(Procedure(name, "", "", []))
    elif entityType == "Function":
        # name, datatype, startingQuad, framelength, formalParameters
        myTable.scopes[-1].entities.append(Function(name, "Int", "", "", []))
    elif entityType == "Parameter":
        # name, datatype, mode, offset:
        myTable.scopes[-1].entities.append(Parameter(name, "Int", extraOp[0], extraOp[1]))
    elif entityType == "TemporaryVariable":
        myTable.scopes[-1].entities.append(TemporaryVariable(name, "Int", extraOp))
    else:
        return 1

    return 0

```

```
def addLayer():
```

Αυτή η συνάρτηση δημιουργεί ένα καινούργιο επίπεδο στον πίνακα συμβόλων. Προγραμματιστικά μιλώντας, ο πίνακας αντιπροσωπεύεται από το myTable, και το scopes που είναι πεδίο του, κρατάει τα επίπεδα, άρα προσθέτει ένα καινούργιο στοιχείο στο scopes.

Πιο συγκεκριμένα:

Αρχικά, ελέγχουμε αν το length της λίστας scopes είναι 0 (δηλαδή η λίστα είναι κενή), που αν ισχύει, σημαίνει ότι το επίπεδο που θα προσθέσει η συνάρτηση θα είναι το πρώτο. Σε αυτήν την περίπτωση δημιουργείται μια κλάση Scope και οι τιμές των πεδίων θα είναι για το level = 0, διότι είμαστε στο πρώτο επίπεδο, και το entities θα ισούται με μια κενή λίστα καθώς δεν υπάρχουν ακόμα οντότητες. Αμέσως στην ίδια εντολή, το πεδίο scopes του myTable παίρνει ως τιμή το Scope που μόλις δημιουργήθηκε

```

1555 def addLayer():
1556     global myTable
1557
1558     if len(myTable.scopes) == 0:
1559         myTable.scopes = [Scope(0, [])]
1560     else:
1561         tmp = myTable.scopes[-1].level
1562         myTable.scopes.append(Scope(tmp+1, []))
1563
1564     return 0

```

Αν υπάρχουν ήδη επίπεδα στον πίνακα συμβόλων (δηλαδή στοιχεία στην λίστα scopes), τότε πηγαίνει στην διακλάδωση του else. Εκεί αρχικά σε μια μεταβλητή tmp,

αποθηκεύουμε τον αριθμό του τελευταίου επιπέδου, δηλαδή το πεδίο level απ' το τελευταίο στοιχείο του scores. Έπειτα με την χρήση append προσθέτουμε νέο στοιχείο στο scores, το οποίο θα είναι κλάση Score, με τιμές πεδίων tmp+1 για το level (λογικό, εφόσον θεωρητικά το νέο επίπεδο βρίσκεται αμέσως πάνω απ' το προηγούμενο), και μια κενή λίστα για το entities καθώς δεν υπάρχουν ακόμα οντότητες για το επίπεδο.

```
def removeLayer():
```

Σε αυτήν εδώ την συνάρτηση αφαιρούμε ένα επίπεδο απ' τον πίνακα, και συγκεκριμένα το τελευταίο.

Η διαδικασία είναι απλή. Αρχικά ελέγχουμε αν το μήκος της λίστας scores (πεδίο του myTable) είναι μηδεν, τότε δεν έχουμε κάποιο επίπεδο να αφαιρέσουμε, και προφανώς δεν θα γίνει κάποια ενέργεια και θα επιστρέψει η συνάρτηση

```
1566 def removeLayer():
1567     global myTable
1568
1569     printBoard()
1570
1571     if len(myTable.scores) == 0:
1572         return 0
1573     else:
1574         myTable.scores.pop()
1575
1576     return 0
```

Αλλιώς, θα μπει στον κλάδο esle, και με την χρήση του pop() της python, αφαιρεί το τελευταίο στοιχείο του scores, το οποίο είναι το τελευταίο επίπεδο

```
def updateEntity(flag, value):
```

Όπως ίσως προαναφέρθηκε, πολλές φορές όταν δημιουργούμε μια οντότητα, δεν γνωρίζουμε πάντα εκείνη την στιγμή όλα τα πεδία της. Γιαυτό τον λόγο υπάρχει η συνάρτηση updateEntity που ενημερώνει μια οντότητα, ανάλογα με το πεδίο που της λείπει.

Αρχικά υπάρχουν δύο περιπτώσεις που θα χρειαστεί η updateEntity(). Η μία είναι να θέλει ενημέρωση το startingQuad, και η άλλη να θέλει ενημέρωση το framelength. Για να γνωρίζουμε κάθε φορά σε ποια περίπτωση είμαστε, η συνάρτηση παίρνει σαν όρισμα το **flag** όπου παίρνει τις τιμές 0 και 1, για κάθε περίπτωση αντίστοιχα. Επομένως εάν το flag

είναι 0 μπαίνει στην διακλάδωση του if για να ενημερώσει το startingQuad, αλλιώς αν είναι 1 μπαίνει στην διακλάδωση του else για το framelength

```
1578 def updateEntity(flag, value):
1579     global myTable
1580
1581     if flag == 0:
1582         myTable.scopes[-2].entities[-1].startingQuad = value
1583     elif flag == 1:
1584         myTable.scopes[-2].entities[-1].framelength = value
1585     return 0
```

Εφόσον όμως δεν παίρνουμε κάπου το Entity σαν όρισμα, πως γνωρίζουμε ποιανού τα πεδία θα ενημερώσουμε; Δεν σου λεω

```
def addParameter():
```

Κάθε κλάση τύπου Function ή Procedure, έχει ως πεδίο μία λίστα formalParameters, η οποία κρατάει όλες τις παραμέτρους της συνάρτησης/διαδικασίας της μόρφης κλάση τύπου FormalParameter. Η συνάρτηση αυτή λοιπόν προσθέτει μια παράμετρο σε αυτήν την λίστα

Όπως θα δούμε στην συνέχεια, η διαδικασία είναι με τέτοιον τρόπο, ώστε όταν βρούμε την πρώτη παράμετρο, έχει δημιουργηθεί ήδη νέο επίπεδο. Επομένως η συνάρτηση είναι αποθηκευμένη στην προτελευταία θέση του scopes.

Άρα πρώτα σε μια τοπική μεταβλητή **currentParameter** αποθηκεύουμε την παράμετρο, η οποία είναι η τελευταία οντότητα στο τελευταίο επίπεδο, συνεπώς τελευταίο στοιχείο του entities, που είναι πεδίο του τελευταίου στοιχείου του scopes

```
1587 def addParameter():
1588     global myTable
1589     currentParameter = myTable.scopes[-1].entities[-1]
1590     myTable.scopes[-2].entities[-1].formalParameters.append(currentParameter)
1591
1592     return 0
```

Στην συνέχεια πάμε στο προτελευταίο στοιχείο του scopes, όπου όπως εξηγήσαμε είναι αποθηκευμένη η συνάρτηση, το οποίο έχει ως πεδίο την λίστα entities, απ' την οποία το τελευταίο στοιχείο είναι η συνάρτηση που θέλουμε. Η συνάρτηση αυτή έχει ως πεδίο την λίστα formalParameters όπου αποθηκεύει τις παραμέτρους, και εκεί προσθέτουμε και την currentParameter.

```
def searchEntity(ent_name):
```

Αυτή είναι μια συνάρτηση που αφορά κυρίως την διαδικασία παραγωγής του τελικού κώδικα, αλλά παρ' όλα αυτά χρησιμοποιεί τον πίνακα συμβόλων. Στην ουσία ελέγχει αν το όνομα μιας οντότητας που δίνει ως όρισμα στο **ent_name** υπάρχει στον πίνακα συμβόλων, δηλαδή στο myTable.

Η διαδικασία αναζήτησης γίνεται μέσα σε 2 βρόγχους for, ο ένας φωλιασμένος μέσα στον άλλον. Λόγω του ότι έχουν προτεραιότητα οι οντότητες στα τελευταία επίπεδα, ο πρώτος βρόγχος που ψάχνει απ' το τέλος του scopes στο myTable, με βήμα -1. Και για κάθε Scope, ο δεύτερος βρόγχος ψάχνει μέσα στην λίστα entities του κάθε επιπέδου. Αν το όνομα της οντότητας (δηλαδή το πεδίο name) είναι το ίδιο με το όρισμα ent_name, τότε επιστρέφει μια λίστα που το πρώτο στοιχείο της είναι η οντότητα, και το δεύτερο είναι ο αριθμός του επιπέδου στο οποίο βρέθηκε

```
1594 def searchEntity(ent_name):
1595     global myTable
1596     for i in range(len(myTable.scopes)-1, 0, -1):
1597         for j in myTable.scopes[i].entities:
1598             if j.name == ent_name:
1599                 return [j, myTable.scopes[i].level]
```

Έτσι όπως υλοποιήθηκε η παραπάνω αναζήτηση προέκυψαν θέματα και δεν φτάνει ποτέ στο αρχικό επίπεδο, που είναι το Scope με level = 0. Έτσι αμέσως μετά υλοποιείται άλλη μια αναζήτηση με if, ειδικά για αυτό το επίπεδο

```
1601     for j in myTable.scopes[0].entities:
1602         if j.name == ent_name:
1603             return [j, 0]
```

Η λογική είναι ακριβώς η ίδια

Τέλος, σε περίπτωση που δεν βρεθεί η οντότητα, επιστρέφει -1

```
1604
1605     return -1
1606
```

Συνοψίζοντας,

Έχουμε εξηγήσει όλες τις συναρτήσεις και τις global μεταβλητές που αφορούν τον ενδιάμεσο κώδικα, καθώς και αυτές που αφορούν τον τελικό. Παρακάτω θα δούμε με ποιον τρόπο αυτές χρησιμοποιούνται στις συναρτήσεις του συντακτικού αναλυτή.

Οι συναρτήσεις του συντακτικού αναλυτή έχουν μεν ήδη εξηγηθεί, οπότε ναι

-program()

Αρχικά προσθέτουμε στο blocknames το ID του προγράμματος, το οποίο πήραμε απ' το currentWord καθώς το έχει επιστρέψει ο lex(). Έτσι, πλέον ξέρουμε σε ποιο block

βρισκόμαστε (θα χρειαστεί αργότερα όταν έρθει η ώρα να δημιουργηθεί η εντολή `begin_block`)

Η `program()` είναι η πρώτη συνάρτηση που καλείται στο πρόγραμμα. Επομένως σημαίνει ότι δεν υπάρχουν ακόμα επίπεδα. Έτσι, δημιουργούμε το πρώτο επίπεδο του πίνακα καλώντας την συνάρτηση `addLayer`, η οποία θα βάλει στην λίστα `scopes` του `myTable`, το πρώτο της στοιχείο τύπου κλάσης `Scope` με πεδίο `level = 0`.

Γενικά, θέλουμε η πρώτη οντότητα που θα εμφανιστεί να έχει `offset = 12`. Όταν θα έρθει η ώρα να πάρει τιμή το `offset` της οντότητας αυτής, θα πάρει την τιμή της `global` μεταβλητής `totaloffset` και θα προσθέσει 4. Επομένως για να γίνει αυτό, δίνουμε στο `totaloffset` την τιμή 8, έτσι ώστε η πρώτη οντότητα που θα εμφανιστεί να έχει `offset = 8 + 4 = 12`

```
269 def program():
270
271     if lex() != "program":
272         print("ERROR! at line " + str
273             exit() # false
274
275     finalFile.write("\t.data\nstr_n\l:
276     finalFile.write("\tj L_main_\n")
277
278     currentWord = lex()
279     ID = currentWord
280     blocknames = blocknames + [ID]
281
282     addLayer()
283     totaloffset = 8
```

`-block()`

Στον ενδιαμέσο κώδικα δεν θέλουμε φωλιασμένα `block` μέσα σε άλλα, επομένως οι παρακάτω εντολές είναι τοποθετημένες αφού έχουν οριστεί τα υποπρογράμματα (αν υπάρχουν) του παρόντος `block`.

Εφόσον μπήκαμε μόλις στο `block`, χρειάζεται να δημιουργηθεί η εντολή `begin_block` και στην συνέχεια ακολουθεί το όνομα του `block`. Πριν από αυτό δημιουργούμε το `label` της εντολής με την συνάρτηση `nextQuad()`, και την τιμή που επιστρέφει την περνάμε ως όρισμα στο `makeList`, έτσι ώστε να είναι συμβατή για να αποθηκευτεί στην μεταβλητή `labelsList`. Και από κάτω με την χρήση `genQuad()` δημιουργούμε την τετράδα για την εντολή του

begin_block, μαζί με το όνομα του που είναι το τελευταίο στοιχείο του blocknames.

```
334         # endiamesos
335         labelslist = labelslist + makeList(nextQuad())
336         genQuad('begin_block', blocknames[-1], '_', '_')
```

Σε αυτή την φάση επίσης ξέρουμε το label που αντιστοιχεί στον πρώτη εντολή ενδιάμεσου κώδικα του block, επομένως μπορούμε να το να δεν έχω ιδέα τι γίνεται εδώ, θα το δω μετά

```
339
340         if len(myTable.scopes) > 1:
341             updateEntity(0, labelslist[-1]+1)
342
```

Αμέσως μετά, στα πλαίσια της συντακτικής ανάλυσης καλείται η blockstatements() και επιστρέφει, οπότε πλέον δεν υπάρχουν άλλες εντολές στο συγκεκριμένο block.

Ισχύει ότι όταν τελειώνει το block του κύριου προγράμματος, στον ενδιάμεσο κώδικα υπάρχει η εντολή halt ___. Για να ξέρουμε αν βρισκόμαστε στο κυρίως block, ελέγχουμε πόσα στοιχεία υπάρχουν στο blocknames. Λειτουργεί σαν FILO ουρά, οπότε αν έχει μόνο ένα στοιχείο (length = 1) σημαίνει ότι βρισκόμαστε σίγουρα σε αυτήν την περίπτωση. Έτσι, αν μπει στον βρόγχο if, δημιουργεί καινούργιο label με την ίδια διαδικασία με πριν, και έπειτα κάνει χρήση της genQuad για να παράγει την εντολή "halt"

```
347         # endiamesos
348         if len(blocknames) == 1:
349             labelslist = labelslist + makeList(nextQuad())
350             genQuad('halt', '_', '_', '_')
```

Αφού λοιπόν έχουμε φτάσει στο τέλος του block, χρειάζεται να δημιουργηθεί η εντολή end_block. Αρχικά δημιουργούμε νέο label με την γνωστή διαδικασία, χρησιμοποιώντας makeList() και nextQuad(), αποθηκεύοντας το στην λίστα labelslist. Για να δημιουργήσουμε την εντολή τώρα χρειαζόμαστε το όνομα του (υπο) προγράμματος του block, το οποίο είναι το τελευταίο στοιχείο της λίστα blocknames. Το αποθηκεύουμε στην τοπική μεταβλητή, και ταυτόχρονα το αφαιρούμε απ' την blocknames με την χρήση του .pop(). Και τέλος με την genQuad δημιουργούμε την τετράδα

```
352         labelslist = labelslist + makeList(nextQuad())
353         tmpname = blocknames.pop()
354         genQuad('end_block', tmpname, '_', '_')
```

Το πάνω δεν θυμάμαι τι είναι... μετά εφόσον ολοκληρώθηκε το block, σημαίνει ότι πρέπει να αφαιρέσουμε το επίπεδο του απ' τον πίνακα συμβόλων, και για αυτόν τον λόγο καλούμε

την removeLayer()

```
362         if len(myTable.scopes) > 1:
363             totaloffset = totaloffset + 4
364             updateEntity(1, totaloffset)
365
366         removeLayer()
```

varlist():

Όπως είχαμε αναφέρει και νωρίτερα εδώ είναι το σημείο που ορίζονται οι μεταβλητές

Επομένως, αφού ο lex() επιστρέψει το όνομα κάποιας μεταβλητής, το πρώτο που κάνουμε είναι να υπολογίσουμε το offset της. Εφόσον η global μεταβλητή totaloffset κρατάει το offset της προηγούμενης οντότητας, το αυξάνουμε κατά 4 και έχουμε το offset που θέλουμε.

Αμέσως κάνουμε προσθέτουμε την νέα οντότητα τύπου κλάση Variable στον πίνακα συμβόλων στην κατάλληλη θέση με την χρήση του addNewRegister

```
410         totaloffset = totaloffset + 4
411         addNewRegister("Variable", currentWord, totaloffset)
```

Ελα δεν σε ειδα, τι εγινε; Όταν μπορείς ψάξε μια για Βασίλης_OP

subprogram():

Ο κώδικας που μας ενδιαφέρει ξεκινάει απ' το σημείο που ο lex() έχει επιστρέψει ένα εκ των "function" ή "procedure". Την τιμή αυτή την αποθηκεύουμε στην τοπική μεταβλητή **flag** διότι μας ενδιαφέρει να ξέρουμε αν πρόκειται για συνάρτηση ή διαδικασία

```
449         flag = currentWord
```

Παρακάτω αφού προχωρήσει η συντακτική ανάλυση, το currentWord θα έχει το όνομα της συνάρτησης που ορίζεται. Αρχικά αποθηκεύουμε το όνομα αυτό στην λίστα blocknames διότι πρόκειται να ανοίξει καινούργιο block. Και παρακάτω ελέγχουμε αν το flag έχει την τιμή Function ή Procedure, και αναλόγως την περίπτωση, καλούμε την addNewRegister να προσθέσουμε την νέα οντότητα, με ορίσματα το τύπο κλάσης της οντότητας (αναλόγως το flag), το όνομα του υποπρογράμματος (είναι η τιμή του currentWord) και 0 ως τελευταίο όρισμα καθώς δεν ξέρουμε άλλη πληροφορία ακόμα

```
456         blocknames = blocknames + [currentWord] # endiamesos
457
458         if flag == "function":
459             addNewRegister("Function", currentWord, 0)
460         else:
461             addNewRegister("Procedure", currentWord, 0)
```

Από την στιγμή που ορίζεται καινούργια συνάρτηση ή διαδικασία, σημαίνει ότι θα χρειαστούμε καινούργιο επίπεδο στον πίνακα συμβόλων.

Αρχίζουμε με την εντολή `addLayer` για να δημιουργηθεί το καινούργιο επίπεδο. Το πρόβλημα εδώ είναι ότι στο νέο επίπεδο, το `totaloffset` θα χρειαστεί να πάρει πάλι την τιμή 8. Για να μην χάσουμε την παρούσα τιμή του, πριν την ανάθεση τιμής 8 στο `totaloffset`, την αποθηκεύουμε στην global λίστα `offsetCheckPoint`, το οποίο επίσης λειτουργεί σαν FILO ουρά.

```
463         addLayer()
464         offsetCheckPoint = offsetCheckPoint + [totaloffset]
465         totaloffset = 8
```

Και αργότερα, αφού με το καλό επιστρέψει η `block()`, το `totaloffset` θα πάρει πάλι την τιμή που είχε στο παρόν block, και θα διαγραφθεί απ' την λίστα `offsetCheckPoint` με την χρήση του `pop()`.

```
480         if block() == 1:
481             return 1
482
483
484         totaloffset = offsetCheckPoint.pop()
485
```

`formalparitem()`:

Είμαστε τώρα στο σημείο που έχουμε μια παράμετρο του υποπρογράμματος. Πρέπει να ξέρουμε αν η παράμετρος είναι περασμένη με τιμή (in) ή με αναφορά (inout). Λόγω του ότι το `currentWord` έχει για τιμή το όνομα της παραμέτρου, θα χρησιμοποιήσουμε το `lastWord` που κρατάει την προηγούμενη λεκτική μονάδα που επέστρεψε ο `lex()`. Επομένως, αν το `lastWord` έχει την τιμή "in", η τοπική μεταβλητή `mode` θα πάρει την τιμή "cv", διαφορετικά θα πάρει την τιμή "ref"

```
536         if lastWord == "in":
537             mode = "cv"
538         else:
539             mode = "ref"
```

Η παράμετρος είναι νέα οντότητα, επομένως θα αυξηθεί το offset κατά 4, και στην συνέχεια με την συνάρτηση `addNewRegister`, θα αποθηκεύσουμε την παράμετρο στον πίνακα συμβόλων, ως τύπο κλάση `Parameter`. Τα ορίσματα που θα πάρει η συνάρτηση για να συμπληρωθούν τα πεδία είναι το `currentWord` ως όνομα οντότητας (name), και μια λίστα με δυο στοιχεία, το `mode` ("cv" ή "ref") και το `totaloffset` που θα είναι το πεδίο offset

```
541         totaloffset = totaloffset + 4
542         addNewRegister("Parameter", currentWord, [mode, totaloffset])
543         addParameter()
```

Και τέλος, επειδή είναι παράμετρος, θα κληθεί και η συνάρτηση `addParameter()` για να προστεθεί στην αντίστοιχη λίστα της συνάρτησης/διαδικασίας που ανοίκει

`assignStat()`:

Σε αυτό το σημείο θα γραφτεί σε ενδιάμεσο κώδικα η εντολή ανάθεσης τιμής

Το `currentWord` αυτή την στιγμή έχει το όνομα της μεταβλητής, στην οποία θα ανατεθεί κάποια τιμή. Μέχρι όμως να βρεθεί ποια θα είναι αυτή η τιμή, το όνομα της μεταβλητής θα έχει χαθεί. Για αυτό, πριν κάνουμε οποιαδήποτε ενέργεια το αποθηκεύουμε στην τοπική μεταβλητή **`assignVar`**.

```
667 def assignstat():
668
669     global currentWord
670     global expectation
671     global labelslist
672
673     assignVar = currentWord
```

Η τιμή που θα πάρει η μεταβλητή στο `assignVar` μπορεί να είναι είτε μεταβλητή του προγράμματος της Cimple, είτε σταθερά, είτε κάποια τοπική μεταβλητή του ενδιάμεσου κώδικα. Σε κάθε περίπτωση, θα επιστραφεί από την `expression()` και θα αποθηκευτεί στην τοπική μεταβλητή `E_place`. Παρακάτω γίνεται η διαδικασία δημιουργίας νέου label, και έπειτα δημιουργείται και η εντολή ανάθεσης τιμής, που έχει ως πρώτο στοιχείο το σύμβολο “:=”, δεύτερο στοιχείο την τιμή του `E_place`, και τελευταίο την τιμή του `assignVar`.

```
683     E_place = expression()
684     if E_place == 1:
685         return 1
686
687     # endiamesos
688     labelslist = labelslist + makeList(nextQuad())
689     genQuad(':=', E_place, '_', assignVar)
```