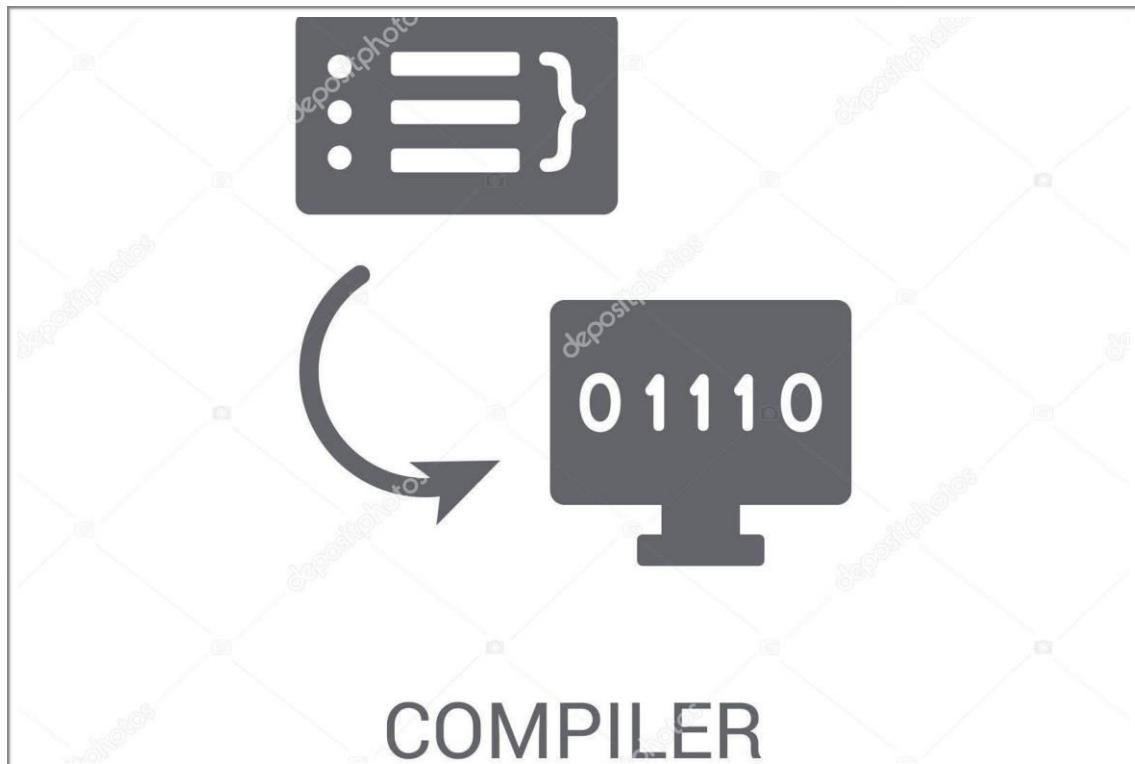


Compilers Report

Πλήρως λειτουργικός μεταγλωττιστής



Σεραφείμ-Ηλίας Αντωνίου - 2640

Σοφία Πασόη - 2798

Άνοιξη 2019

Περιεχόμενα - Contents:

• Εισαγωγή Introduction.....	3
• Γλώσσα Starlet Starlet Language.....	9
• Λεκτικός Αναλυτής VerbalAnalyst.....	16
• Συντακτικός Αναλυτής SyntaxAnalyst.....	29
• Παραγωγή ενδιάμεσου κώδικα Intermediate Code Production... ..	64
• Πίνακας Συμβόλων Symbol Matrix.....	98
• Παραγωγή τελικού κώδικα Final Code Production.....	111

Εισαγωγή - Introduction:

Στην αναφορά μας, θα αναλύσουμε τα βήματα υλοποίησης που ακολουθήσαμε για την κατασκευή ενός πλήρως λειτουργικού μεταγλωττιστή με βάση την προγραμματιστική γλώσσα "Starlet" που κατασκεύασε ο κ. Μανής στα πλαίσια του μαθήματος των Μεταφραστών.

Μεταγλωττιστής - Compiler:

Αρχικά, ένας μεταγλωττιστής δέχεται σαν είσοδο ένα αρχικό πρόγραμμα (source code) το οποίο σκοπεύουμε να μεταγλωττίσουμε (γραμμένο σε κάποια γλώσσα προγραμματισμού), το οποίο υφίσταται επεξεργασία και στη συνέχεια επιστρέφει κάποια διαγνωστικά μηνύματα όπως μηνύματα λάθους (ERROR messages) ή ακόμα και προειδοποιητικά μηνύματα (warnings), ενώ στο τέλος παράγει το τελικό πρόγραμμα (object program) συνήθως γραμμένο σε γλώσσα μηχανής κάποιου επεξεργαστή (π.χ. Assembly -> MIPS).



Γλώσσα Προγραμματισμού - Programming Language:

Το **αρχικό πρόγραμμα** γράφεται σε μία X γλώσσα προγραμματισμού (Starlet, ...) η οποία αποτελεί και το εφαλτήριο της υλοποίησής μας.

Το **τελικό πρόγραμμα** παράγεται στην τελική γλώσσα (τελικός κώδικας), η οποία αποτελεί την γλώσσα στην οποία θέλουμε να μεταγλωττιστεί το πρόγραμμα. Τις περισσότερες φορές η τελική γλώσσα είναι συνήθως γλώσσα μηχανής κάποιου επεξεργαστή. Στην δική μας περίπτωση η γλώσσα μηχανής είναι η Assembly και ο επεξεργαστής ο MIPS.

Ο **μεταγλωττιστής** είναι και αυτός υλοποιημένος σε κάποια γλώσσα προγραμματισμού (στην περίπτωσή μας είναι σε Python).

Το σχεδιάγραμμα της διαδικασίας απεικονίζεται στο παρακάτω σχήμα:



Ανάπτυξη Μεταγλωτιστή - Compiler Development:

Η πρώτη φάση της υλοποίησής μας πραγματοποιήθηκε σε γλώσσα προγραμματισμού Python.

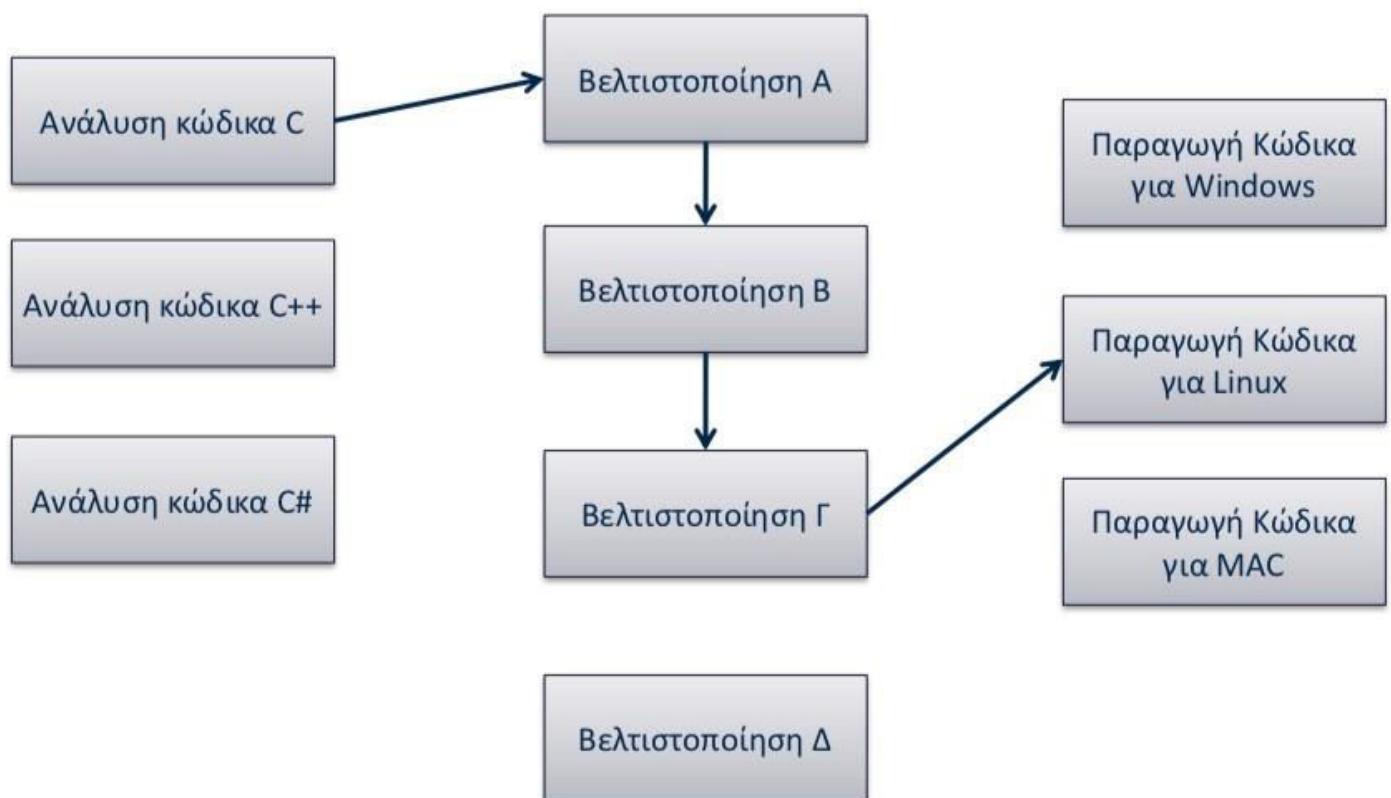
Σχεδιάγραμμα υλοποίησης (με την χρήση εργαλείων):



Σύνθεση από τμήματα λογισμικού - Software Parts Composition:

Γενικώς, έχουμε ένα **σύνολο από λειτουργίες** λογισμικού κάθε μία από τις οποίες επιτελεί μία συγκεκριμένη λειτουργία. Αναλόγως με τις απαιτήσεις μας, **επιλέγουμε** ποιες λειτουργίες θα τεθούν σε χρήση. Δημιουργούμε καινούργιες εάν κρίνεται απαραίτητο ή **προσαρμόζουμε** τις υπάρχουσες στις νέες απαιτήσεις. Επιπρόσθετα, δεν διστάζουμε να **συνδυάσουμε** τμήματα που μας είναι απαραίτητα για την εξασφάλιση της εύρυθμης λειτουργίας του μεταγλωτιστή.

Παράδειγμα σύνθεσης από τμήματα λογισμικού:



Χρήσιμη Ορολογία - Useful Terminology:

- **Διαμεταγλωττιστές (cross-compilers):** Ο υπολογιστής στον οποίο γίνεται η μετάφραση είναι διαφορετική αρχιτεκτονική από τον υπολογιστή στον οποίο θα εκτελεστεί το τελικό πρόγραμμα.
- **Προεπεξεργαστές (preprocessors):** Από αρχική γλώσσα πάλι σε αρχική γλώσσα αφού έχει κάποιες τροποποιήσεις, π.χ. αντικατάσταση συμβολικών ονομάτων των σταθερών με τις πραγματικές τιμές τους.
- **Συμβολομεταφραστές (assemblers):** Από συμβολική γλώσσα μηχανής σε γλώσσα μηχανής – μία προς μία αντιστοίχιση εντολών.
- **Βιβλιοθήκες χρόνου εκτέλεσης (run-time libraries):** Υποπρογράμματα και συναρτήσεις σε μεταγλωττισμένη μορφή.
- **Συνδέτες (linkers):** Έχουν πια ενσωματωθεί στη διαδικασία μεταγλωττισης, παλαιότερα ήταν χωριστά προγράμματα. Δέχονται σαν είσοδο μεταγλωττισμένα αρχεία και βιβλιοθήκες χρόνου εκτέλεσης και παράγουν ως έξοδο ένα εκτελέσιμο πρόγραμμα. Αναλύουν τις σχετικές διευθύνσεις των μεταγλωττισμένων αρχείων και των προγραμμάτων βιβλιοθήκης και τα ενοποιούν ώστε να είναι όλες με βάση μία σχετική διεύθυνση.
- **Φορτωτές (loaders):** Δέχονται σαν είσοδο ένα εκτελέσιμο πρόγραμμα και το τοποθετούν στη μνήμη για να είναι έτοιμο προς εκτέλεση. Μετατρέπουν τις σχετικές διευθύνσεις σε απόλυτες διευθύνσεις. Αποτελεί μέρος του λειτουργικού συστήματος.
- **Εκδότες προγραμμάτων (program editors):** Χρησιμοποιούνται στη συγγραφή, διόρθωση προγραμμάτων και αποθήκευσή τους στη περιφερειακή μονάδα του υπολογιστή.
- **Εντοπιστές σφαλμάτων (debuggers):** Βηματική εκτέλεση προγραμμάτων. Παρακολούθηση τιμών μεταβλητών. Τοποθέτηση σημείων διακοπής εκτέλεσης.
- **Στατιστικοί αναλυτές (profiles):** Δίνουν πληροφορίες σχετικά με την εκτέλεση ενός προγράμματος, π.χ. απασχόληση των επεξεργαστών όταν χρησιμοποιούμε ένα υπολογιστικό σύστημα πολλών επεξεργαστών.

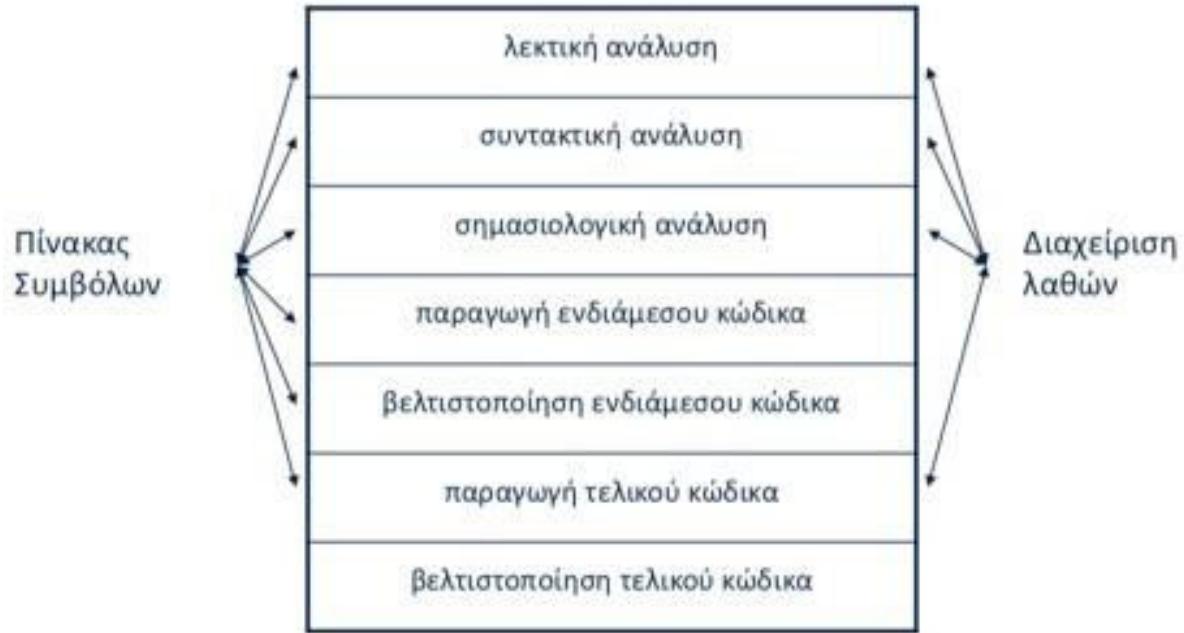
Διερμηνείς- Interpreters:

- Μετάφραση και **εκτέλεση εντολή προς εντολή** σε αντίθεση με τους μεταγλωττιστές που μεταφράζουν μία φορά το πηγαίο πρόγραμμα το αποθηκεύουν στο δίσκο και το εκτελούν από εκεί.
- **Ευκολότερη ανάπτυξη λογισμικού** από ότι με τους μεταγλωττιστές.
- **Ευκολότερη** η βήμα προς βήμα εκτέλεσή άρα και η αποσφαλμάτωση.
- Σημαντικά **πτιο αργοί** από τους μεταγλωττιστές.
- **Ασφάλεια** εκτέλεσης σε σχέση με τους μεταγλωττιστές.

Απαιτήσεις Μεταγλωττιστή - Compiler Conditions:

- **Εύρυθμη λειτουργία.**
- Να συμμορφώνεται με τις **προδιαγραφές/απαιτήσεις** της αρχικής και τελικής γλώσσας.
- Να μεταγλωττίζει προγράμματα **αυθαίρετα μεγάλου μήκους**.
- Να παράγει **αποδοτικό κώδικα**.
- Να έχει **μικρό χρόνο εκτέλεσης**.
- Να έχει **μικρές απαιτήσεις μνήμης** κατά την μεταγλώττιση.
- Να επιστρέψει **σωστά διαγνωστικά μηνύματα**.
- Να έχει την δυνατότητα **συνέχισης** μετά από τον εντοπισμό σφαλμάτων.
- Να είναι **μεταφέρσιμος**.

Στάδια της Μεταγλώττισης - Compiling Stages:



Γλώσσα προγραμματισμού Starlet - Starlet Programming Language:

Η Starlet είναι μια μικρή γλώσσα προγραμματισμού φτιαγμένη με βάση τις ανάγκες της προγραμματιστικής άσκησης του μαθήματος. Παρόλο που οι προγραμματιστικές της ικανότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από άλλες γλώσσες, καθώς και κάποιες πρωτότυπες. Η Starlet υποστηρίζει συναρτήσεις, μετάδοση παραμέτρων με αναφορά, τιμή και αντιγραφή, αναδρομικές κλήσεις και άλλες ενδιαφέρουσες δομές. Επίσης, επιτρέπει φώλιασμα στη δήλωση συναρτήσεων κάτι που λίγες γλώσσες υποστηρίζουν (το υποστηρίζει η Pascal, δεν το υποστηρίζει η C). Από την άλλη όμως πλευρά, η Starlet δεν υποστηρίζει βασικά προγραμματιστικά εργαλεία όπως η δομή *for*, ή τύπους δεδομένων όπως οι πραγματικοί αριθμοί και οι συμβολοσειρές. Οι παραλήψεις αυτές έχουν γίνει ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή, μία απλούστευση όμως που έχει να κάνει μόνο με τη

μείωση των γραμμών κώδικα και όχι με τη δυσκολία κατασκευής του ή την εκπαιδευτική αξία της άσκησης.

Παρακάτω παρουσιάζεται μία περιγραφή της γλώσσας:

Λεκτικές μονάδες - Verbal Units:

Το αλφάριθμο της Starlet αποτελείται από:

- **Μικρά & κεφαλαία γράμματα της λατινικής αλφαριθμήτου:** “A”...”Z” & “a”...”z”.
- **Τα αριθμητικά ψηφία:** “0”...”9”.
- **Τα τελεστικά σύμβολα πράξεων:** “+”, “-”, “*”, “/”.
- **Τους τελεστές συσχέτισης:** “<”, “>”, “<=”, “>=”, “<>”.
- **Το σύμβολο ανάθεσης:** “:=“.
- **Τους διαχωριστές:** “:”, “;”, “,”.
- **Σύμβολα ομαδοποίησης:** “(“, “)”, “[“, “]”.
- **Διαχωρισμό σχολίων:** “/*”, “*/“, “//”.

Δεσμευμένες λέξεις - Reserved words:

program, endprogram

declare

if then else endif

while endwhile dowhile enddowhile

loop, endloop, exit

forcase, endforcase, incase, end incase, when, default, end default

function, endfunction, return, in, inout,inandout

and, or, not

input, print

Οι λέξεις αυτές δεν μπορούν να χρησιμοποιηθούν ως μεταβλητές. Οι σταθερές της γλώσσας είναι ακέραιες σταθερές που αποτελούνται από προαιρετικό πρόσημο και από μία ακολουθία αριθμητικών ψηφίων.

Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα. Οι λευκοί χαρακτήρες (tab, space, return)) αγνοούνται και μπορούν να

χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια να μην βρίσκονται μέσα σε δεσμευμένες λέξεις, αναγνωριστικά, σταθερές. Το ίδιο ισχύει και για τα σχόλια, τα οποία πρέπει να βρίσκονται μέσα στα σύμβολα /* και */ ή να βρίσκονται μετά το σύμβολο // και ως το τέλος της γραμμής. Απαγορεύεται να ανοίξουν δύο φορές σχόλια, πριν τα πρώτα κλείσουν. Δεν υποστηρίζονται εμφωλευμένα σχόλια.

Μορφή προγράμματος - Starlet program form:

program id

 declarations

 subprograms

 statements

endprogram

Τύποι & δηλώσεις μεταβλητών - Types & statements of variables:

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η Starlet είναι οι ακέραιοι αριθμοί. Οι ακέραιοι αριθμοί πρέπει να έχουν τιμές από –32767 έως 32767. Η δήλωση γίνεται με την εντολή **declarations**. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές και χωρίς να είναι αναγκαίο να βρίσκονται στην ίδια γραμμή. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης αναγνωρίζεται με το ελληνικό ερωτηματικό. Επιτρέπεται να έχουμε περισσότερες των μία συνεχόμενες χρήσεις της **declarations**.

Τελεστές & εκφράσεις - Operands & expressions:

Η προτεραιότητα των τελεστών από την μεγαλύτερη προς την μικρότερη είναι:

- (1) Μοναδιαίοι λογικοί: “not”
- (2) Πολλαπλασιαστικοί: “*”, “/“
- (3) Μοναδιαίοι προσθετικοί: “+”, “-“
- (4) Δυαδικοί προσθετικοί: “+”, “-“
- (5) Σχεσιακοί: “=”, “<”, “>”, “<>”, “<=”, “>=“

(6) Λογικό “and”

(7) Λογικό “or”

Δομή της γλώσσας - Language structure:

Εκχώρηση:

id := expression

Χρησιμοποιείται για την ανάθεση της τιμής μίας μεταβλητής ή μίας σταθεράς, ή μίας έκφρασης σε μία μεταβλητή.

Απόφαση if:

if (condition) **then**

 statements

[else

 statements]

endif

Η εντολή απόφασης **if** εκτιμάει εάν ισχύει η συνθήκη **condition** και εάν πράγματι ισχύει, τότε εκτελούνται οι εντολές που ακολουθούν το **then** έως ότου συναντηθεί **else** ή **endif**. Το **else** δεν αποτελεί υποχρεωτικό τμήμα της εντολής και γι' αυτό βρίσκεται σε αγκύλη. Οι εντολές που το ακολουθούν εκτελούνται εάν η συνθήκη **condition** δεν ισχύει. Το **endif** είναι υποχρεωτικό τμήμα της εντολής.

Επανάληψη while:

while (condition)

 statements

endwhile

Η εντολή επανάληψης **while** επαναλαμβάνει συνεχώς τις εντολές **statements** που βρίσκονται ανάμεσα στο **while** και στο **endwhile**, όσο η συνθήκη **condition** ισχύει. Αν την πρώτη φορά που θα αποτιμηθεί η condition, το αποτέλεσμα της αποτίμησης είναι ψευδές, τότε οι **statements** δεν εκτελούνται ποτέ.

Επανάληψη dowhile - enddowhile:

dowhile

 statements

enddowhile (condition)

Η εντολή επανάληψης **dowhile-enddowhile** επαναλαμβάνει συνεχώς τις εντολές **statements** που βρίσκονται ανάμεσα στο **dowhile** και στο **enddowhile**, όσο η συνθήκη **condition** ισχύει. Οι **statements** εκτελούνται τουλάχιστον μία φορά, πριν αποτιμηθεί η **condition**.

Επανάληψη loop:

```
loop
    statements
endloop
```

Η εντολή επανάληψης **loop** επαναλαμβάνει για πάντα τις εντολές **statements** που βρίσκονται ανάμεσα στο **loop** και στο **endloop**. Έξοδος από το βρόχο γίνεται όταν κληθεί η εντολή **exit**.

Επανάληψη forcase:

```
forcase
    (when (condition): statements)*
    default: statements enddefault
endforcase
```

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

Επανάληψη incase:

```
incase
    (when (condition): statements)*
endincase
```

Η δομή επανάληψης **incase** ελέγχει τις **condition** που βρίσκονται μετά τα **when**, εξετάζοντας τες κατά σειρά. Για κάθε μία από αυτές που η αντίστοιχη **condition** ισχύει, εκτελούνται οι **statements** που ακολουθούν το σύμβολο ":". Θα εξεταστούν όλες οι **condition** και θα εκτελεστούν όλες οι **statements** των οποίων οι **condition** ισχύουν. Αφότου εξεταστούν όλες οι **when** ο έλεγχος μεταβαίνει έξω από τη δομή **incase** εάν καμία από τις **statements** δεν έχει εκτελεστεί ή μεταβαίνει στην αρχή της **incase**, έαν έστω και μία από τις **statements** έχει εκτελεστεί.

Επιστροφή τιμής:

return expression

Χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης.

Έξοδος:

print expression

Εμφανίζει στην οθόνη το αποτέλεσμα της αποτίμησης του **expression**.

Είσοδος:

input id

Ζητάει από τον χρήστη να δώσει μία τιμή μέσα από το πληκτρολόγιο.

Υποπρογράμματα - Subprograms:

Η Starlet υποστηρίζει συναρτήσεις:

function id (formal_pars)

declarations

subprograms

statements

endfunction

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

Μετάδοση παραμέτρων - Parameter passing:

Η Starlet υποστηρίζει 3 τρόπους μετάδοσης παραμέτρων:

- **Με σταθερή τιμή:** Δηλώνεται με τη λεκτική μονάδα **in**. Αλλαγές στην τιμή της δεν επιστρέφονται στο πρόγραμμα που κάλεσε τη συνάρτηση.
- **Με αναφορά:** Δηλώνεται με τη λεκτική μονάδα **inout**. Κάθε αλλαγή στη τιμή της μεταφέρεται αμέσως στο πρόγραμμα που κάλεσε τη συνάρτηση.

- Με **αντιγραφή**: Δηλώνεται με τη λεκτική μονάδα **inandout**. Κάθε αλλαγή στη τιμή της μεταφέρεται στο πρόγραμμα που κάλεσε τη συνάρτηση, όταν ολοκληρώνεται η εκτέλεση της συνάρτησης.

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

Κατάληξη - Ending:

Τα αρχεία της Starlet έχουν κατάληξη **.stl**

Λεκτικός Αναλυτής - Verbal Analyst:

Ο λεκτικός αναλυτής καλείται ως **συνάρτηση** από τον συντακτικό αναλυτή. Στη συνέχεια, διαβάζει γράμμα προς γράμμα το πηγαίο πρόγραμμα (στην περίπτωσή μας: Starlet Language). Κάθε φορά που καλείται επιστρέφει την **επόμενη λεκτική μονάδα** (verbal unit). Στον συντακτικό αναλυτή επιστρέφει έναν **ακέραιο** αριθμό που ταυτοποιεί την λεκτική μονάδα και την ίδια την λεκτική μονάδα.

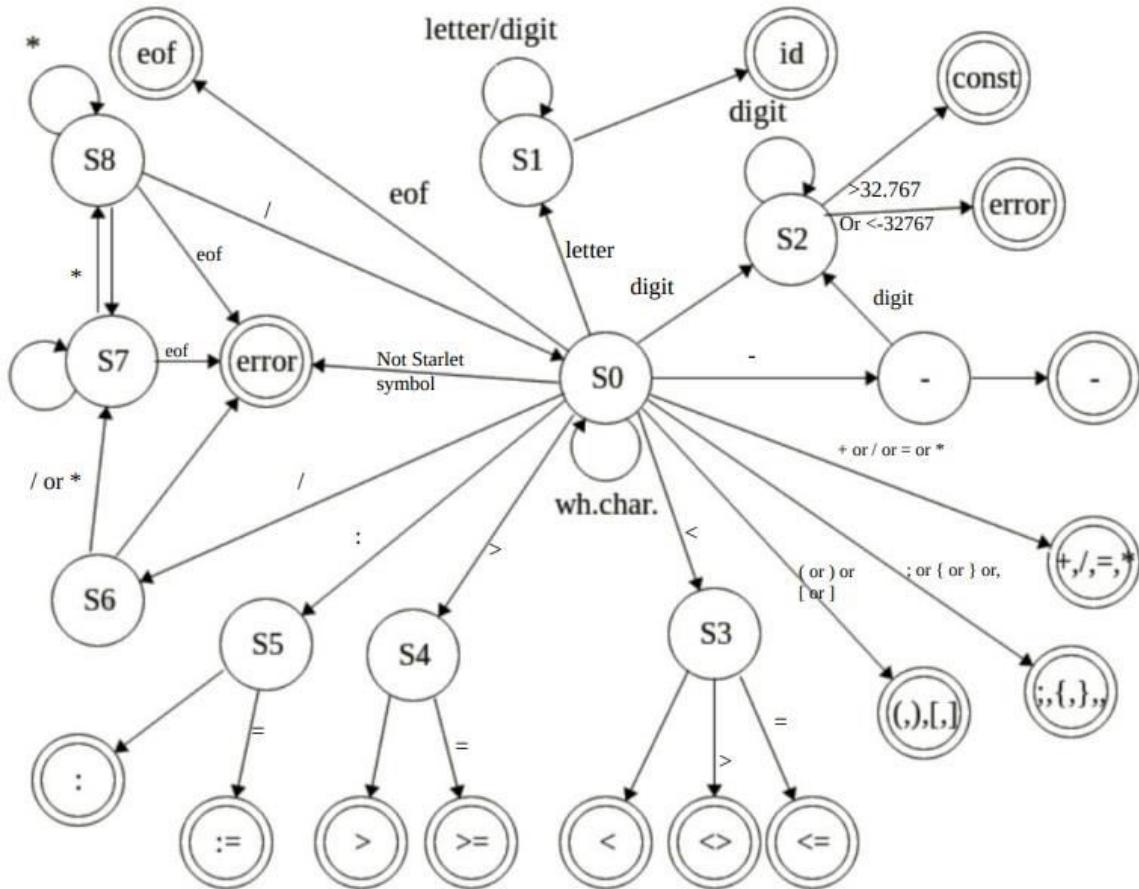


Εσωτερική Λειτουργία Λεκτικού Αναλυτή - Inside Function of the Verbal Analyst:

Ο λεκτικός αναλυτής (*Iex*) εσωτερικά λειτουργεί σαν ένα **αυτόματο καταστάσεων** το οποίο ξεκινά από μία αρχική κατάσταση, με την είσοδο κάθε χαρακτήρα αλλάζει κατάσταση έως ότου συναντήσει μία τελική κατάσταση. Το αυτόματο καταστάσεων αναγνωρίζει:

1. **Δεσμευμένες λέξεις:** if, for, while, ...
2. **Τελεστικά σύμβολα της γλώσσας:** "+", "-", "=" "*", "<>", ...
3. **Αναγνωριστικά και σταθερές:** counter, a12, 32768
4. **Λάθη:** Απαγορευμένους χαρακτήρες, συντακτικά λάθη, ...

Αυτόματο - Automata:



Με βάση το αυτόματο που κληθήκαμε να υλοποιήσουμε, ξεκινούμε από κάποιες αρχικές καταστάσεις (states) και στη συνέχεια αναλόγως με το τι θα “διαβάσει” ο λεκτικός αναλυτής το πρόγραμμα προχωράει αναλόγως. Όσο η κατάσταση στην οποία βρισκόμαστε δεν είναι λάθους, τέλους του αρχείου (EOF) ή απλά πεπερασμένη (OK state), λεκτικός αναλυτής (lex) διαβάζει συνεχώς νέους χαρακτήρες από το αρχείο.

Οι αρχικές καταστάσεις με βάση το πεπερασμένο αυτόματο:

```
# Marks: (strictly based on the Professor's outline) #
state_0 = 0      #Return of symbols
state_1 = 1      #Identifiers --> consist of one letter and a letter or digit
state_2 = 2      #Integer numbers
state_3 = 3      #Symbol: '<'
state_4 = 4      #Symbol: '>'
state_5 = 5      #Symbol: ':'
# --- Comment marks: --- #
state_6 = 6      #Comment symbol '//' 
state_7 = 7      #OPENING comment symbol '/*'
state_8 = 8      #CLOSING comment symbol '*/'
ERROR_state = -1
OK_state = -2
EOF_state = -3
neg_mark = 0     #In case of negative integer numbers
```

Αν ο αναλυτής διαβάσει κάποιο χαρακτήρα που δεν ανηκει στη Starlet τότε από την **κατάσταση_0** μεταβαίνει στην κατάσταση λάθους (Error state). Στην περίπτωση που διαβάσει **eof** τότε από την **κατάσταση_0** μεταβαίνει στην κατάσταση τερματισμού. Κάθε φορά που ο Λεκτικός Αναλυτής καλείται, ξεκινά πάλι από την αρχική κατάσταση, όμως διαβάζει τους χαρακτήρες από το σημείο στο οποίο είχε σταματήσει. Αντιστοίχως κάθε φορά που ολοκληρώνει την αναγνώριση μίας λεκτικής μονάδας την αποθηκεύει σε μια μεταβλητή, την επιστρέφει στον **συντακτικό αναλυτή** (syntax analyst) και τερματίζει.

```
# --- EOF state: --- #
if (state == EOF_state):
    rtrn_verbal_unit = ['EOF'] + [62]
```

Αρχική κατάσταση 0: Ο λεκτικός αναλυτής βρίσκεται στην αρχική κατάσταση 0. Ενόσω διαβάζει λευκούς χαρακτήρες (space, tab, new line) παραμένει σε αυτή. Αν διαβάσει “\n” τότε αυξάνει μία γραμμή και επιστρέφει στη θέση ανάγνωσης 0. Εάν διαβάσει tab, τότε απλώς αυξάνει τη θέση ανάγνωσης πάνω στο αρχείο κατά 4. Αν αναγνώσει γράμμα που ανήκει στους χαρακτήρες ASCII μεταβαίνει στην **κατάσταση_1**, αν αναγνώσει ψηφίο μεταβαίνει στην **κατάσταση_2**. Αν διαβάσει έναν από τους συγκριτικούς τελεστές “>” ή “<” μεταβαίνει στην **κατάσταση_3** και **κατάσταση_4**, αντίστοιχα. Όταν αναγνώσει το σύμβολο “:” μεταβαίνει στην **κατάσταση_5**, με την ανάγνωση του συμβόλου “/” μεταβαίνει στην **κατάσταση_6**, με το πάτημα του κουμπιού ENTER επιστρέφει στην αρχική **κατάσταση_0**.

```
# --- State_0: Initial state: --- #
if (state == state_0):
    if (input == ' ' or input == '\n' or input == '\t'):      #cc
        if (input == '\n'):
            line += 1
            in_line_position = 0
        if (input == '\t'):
            in_line_position += 4      #1 Tab = 4 spaces
        state = state_0
    elif (input in string.ascii_letters):    #input --> letter cl
        state = state_1
    elif (input in string.digits):      #input --> integer nummt
        state = state_2
    elif (input == '<'):
        state = state_3
    elif (input == '>'):
        state = state_4
    elif (input == ':'):
        state = state_5
    elif (input == '/'):
        state = state_6
    elif (input == '\r'):      #Hitting the enter button
        state = state_0
    elif (input == ''):          state = EOF_state
    else:
        state = OK_state
        verbal_unit = input
        identifier = "".join(verbal_unit)

        if (input == '+'):
            rtrn_verbal_unit += [identifier] + [3]
        elif (input == '-'):
            # WARNING: negative integer number check
            input = file_pointer.read(1)
            in_line_position += 1
            if (input in string.digits):
                neg_mark = 1
                state = state_2
            else:
                file_pointer.seek(file_pointer.tell()-2)  #Go back
                in_line_position -= 2
                input = file_pointer.read(1)
                in_line_position += 1
                rtrn_verbal_unit += [identifier] + [4]
        elif (input == '*'):
            rtrn_verbal_unit += [identifier] + [5]
        elif (input == '/'):
            rtrn_verbal_unit += [identifier] + [6]
        elif (input == ';'):
            rtrn_verbal_unit += [identifier] + [7]
        elif (input == ','):
            rtrn_verbal_unit += [identifier] + [8]
        elif (input == '{'):
            rtrn_verbal_unit += [identifier] + [9]
        elif (input == '}'):
            rtrn_verbal_unit += [identifier] + [10]
        elif (input == '('):
            rtrn_verbal_unit += [identifier] + [11]
        elif (input == ')'):
            rtrn_verbal_unit += [identifier] + [12]
        elif (input == '['):
            rtrn_verbal_unit += [identifier] + [13]
        elif (input == ']'):
            rtrn_verbal_unit += [identifier] + [14]
        elif (input == '='):
            rtrn_verbal_unit += [identifier] + [15]
        else:
            state = ERROR_state
            print('ERROR MSG: Unknown character detected: %c' % input)
            print('Line -> %d:%d' % (line, in_line_position))
    
```

Κατάσταση_1: Κατάσταση_0 + Γράμμα:

Όταν διαβάσει γράμμα από την **κατάσταση_0** μεταβαίνει στην **κατάσταση_1**, η οποία θα τον οδηγήσει σε ένα **αναγνωριστικό** (identifier). Για να προστεθεί κάτι στο αναγνωριστικό πρέπει πρώτα να έχει διαβαστεί. Όσο ο αναλυτής λαμβάνει γράμμα ή ψηφίο παραμένει στην **κατάσταση_1** προσθέτοντας το γράμμα ή το ψηφίο στο αναγνωριστικό που θα καταλήξει. Όμως ο αναλυτής μπορεί να διαβάσει μόνο τους πρώτους **30** χαρακτήρες. Αν ξεπεράσει αυτό το όριο τότε το αναγνωριστικό ολοκληρώνεται, ο αναλυτής το επιστρέφει και μεταβαίνει σε τελική κατάσταση. Ομοίως, αν ληφθεί κάτι διαφορετικό από γράμμα ή ψηφίο. Μετά την ολοκλήρωση του αναγνωριστικού ο αναλυτής αναζητά αν η λέξη που διάβασε ανήκει σε κάποια από τις δεσμευμένες λέξεις της Starlet. Στην περίπτωση που δεν ανήκει τη θεωρεί ως id, την κρατάει σε μια λεκτική μονάδα την οποία επιστρέφει και τερματίζει.

```
# --- State_1: State_0 -> Letter --- #
if (state == state_1):
    if (input in string.ascii_letters or input in string.digits):          #Checking if i
        letter_count += 1
        if (letter_count > 30):           #Compiler reads only the first 30 letters
            mark_count = letter_count - 30
        if (mark_count >= 1):
            identifier = "".join(verbal_unit)
            rtrn_verbal_unit += [identifier] + [1]
        else:
            verbal_unit += input
            state = state_1
    else:
        state = OK_state
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit = [identifier] + [1]

        if (mark_count >= 1):
            print('WARNING: Word exceeded the 30 character limit!')
            print('Line -> %d:%d' % (line, in_line_position))

file_pointer.seek(file_pointer.tell()-1)      #Go back 1 position
in_line_position -= 1
```

Κατάσταση_2: Κατάσταση_0 + Ψηφίο:

Όταν διαβάσει ψηφίο από την **κατάσταση_0** μεταβαίνει στην **κατάσταση_2**, η οποία μας οδηγεί σε μία **αριθμητική σταθερά**. Όσο ο αναλυτής λαμβάνει ψηφίο παραμένει στην **κατάσταση_2** προσθέτοντας το ψηφίο στην αριθμητική σταθερά που θα καταλήξει. Η Starlet υποστηρίζει μόνο ακέραιους με πεδίο ορισμού το [-32.767, 32.767]. Αν το ψηφίο που διαβάστηκε δεν ανήκει σε αυτό το διάστημα, τότε ο αναλυτής μεταβαίνει σε **κατάσταση λάθους**. Στην περίπτωση που ληφθεί κάτι διαφορετικό από ψηφίο τότε η αριθμητική σταθερά ολοκληρώνεται, ο αναλυτής την αποθηκεύει σε μία λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση.

```
# --- State_2: State_0 -> Digit --- #
if (state == state_2):
    if (input in string.digits):
        state = state_2
        verbal_unit += input
    else:
        state = OK_state
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [2]
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
# --- Digit check if it's between the limits: --- #
if (int(identifier) > 32767 or int(identifier) < (-32767)):
    state = ERROR_state
    if (int(identifier) > 32767):
        print('ERROR: Number %d exceeded the 32767 limit!' % int(identifier))
        print('Line -> %d:%d' % (line, in_line_position))
    else:
        print('ERROR: Number %d is under the -32767 limit!' % int(identifier))
        print('Line -> %d:%d' % (line, in_line_position))
```

Κατάσταση_3: Κατάσταση_0 + σύμβολο “<“:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “<“ μεταβαίνει στην **κατάσταση_3**. Εκεί υπάρχουν 3 υποπεριπτώσεις:

1. **“<“ + “=“ = “<=“**: Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “<=”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση.
2. **“<“ + “>“ = “<>“**: Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “<>“ ως το διάφορο, το αποθηκεύει σε μια λεκτική μονάδα, την επιστρέφει και μεταβαίνει σε τελική κατάσταση.
3. **“<“ + “<“ = “<“**: Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “<“ ως το μικρότερο, το αποθηκεύει σε μια λεκτική μονάδα, την επιστρέφει και μεταβαίνει σε τελική κατάσταση.

```
# --- State_3: State_0 -> '<' symbol ----- #
if (state == state_3):
    state = OK_state
    input = file_pointer.read(1)
    in_line_position += 1
    if (input == '='):
        verbal_unit = '<='
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [16]
    elif (input == '>'):
        verbal_unit = '<>'
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [17]
    else:
        verbal_unit = '<'
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [18]
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
```

Κατάσταση_4: Κατάσταση_0 + σύμβολο “>”:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “>” μεταβαίνει στην **κατάσταση_4**. Εκεί υπάρχουν 2 υποπεριπτώσεις:

1. **“>” + “=“ = “>=“:** Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “>=”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση.
2. **“>” + “>“ = “>“:** Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “>”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση.

```
# ---- State_4: State_0 -> '>' symbol ---- #
if (state == state_4):
    state = OK_state
    input = file_pointer.read(1)
    in_line_position -= 1
    if (input == '='):
        verbal_unit = '>='
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [19]
        #file_pointer.seek(file_pointer.tell() - 1)
        #in_line_position -= 1
    else:
        verbal_unit = '>'
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [20]
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
```

Κατάσταση_5: Κατάσταση_0 + σύμβολο “:”:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “:” μεταβαίνει στην **κατάσταση_5**. Εκεί υπάρχουν 2 υποπεριπτώσεις:

1. **“:” + “=“ = “:=“:** Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “:=”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση.
2. **“:” + ERROR = ERROR:** Ο λεκτικός αναλυτής αναγκαστικά “πέφτει” σε περίπτωση λάθους, επειδή εάν συναντηθεί το σύμβολο “:” η μοναδική περίπτωση είναι να καταλήξουμε στο “:=“.

```
# ---- State_5: State_0 -> ':' symbol ---- #
if (state == state_5):
    input = file_pointer.read(1)
    in_line_position += 1
    if (input == '='):
        state = OK_state
        verbal_unit = ':='
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [21]
    else:
        state = OK_state
        verbal_unit = ':'
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
        identifier = "".join(verbal_unit)
        rtrn_verbal_unit += [identifier] + [22]
```

Κατάσταση_6: Κατάσταση_0 + σύμβολο “/”:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “/” μεταβαίνει στην **κατάσταση_6**. Εκεί υπάρχουν 2 υποπεριπτώσεις:

1. **“/” + “*” = “/*”**: Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “/*”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση. Στην περίπτωση αυτή πρόκειται για την έναρξη σχολίων, οπότε οτιδήποτε ακολουθεί αναγνωρίζεται σαν σχόλια.
2. **“/” + “//” = “//”**: Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “//”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση. Έπειτα μεταβαίνει στην **κατάσταση_7**. Στην περίπτωση αυτή πρόκειται για την έναρξη σχολίων μίας γραμμής, οπότε οτιδήποτε ακολουθεί αναγνωρίζεται σαν σχόλια έως ότου διαβάσει χαρακτήρα νέας γραμμής.

```
# --- State_6: State_0 -> '/*' symbol --- #
if (state == state_6):
    input = file_pointer.read(1)
    in_line_position += 1
    if (input == '*'):
        state = state_7
        input = file_pointer.read(1)
        in_line_position += 1
    # One-line comments '//' #
    elif (input == '/'):
        state = state_7
        input = file_pointer.read(1)
        in_line_position += 1
    else:
        state = ERROR_state
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
        print('Syntax Error: Missing "*" in comments.')
        print('Line -> %d:%d' % (line, in_line_position))
```

Κατάσταση_7: Κατάσταση_0 + σύμβολο “/*”:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “/ *” μεταβαίνει στην **κατάσταση_7**.

- 1. “/” + “*” = “/*”:** Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “/*”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση. Στην περίπτωση αυτή πρόκειται για την έναρξη σχολίων, οπότε οτιδήποτε ακολουθεί αναγνωρίζεται σαν σχόλια. Αν μέσα στα σχόλια αναγνωρίσει tab τότε μετατίθεται +4 θέσεις δεξιά, ενώ αν αναγνωρίσει \n αλλάζει γραμμή και επιστρέφει στην αρχική κατάσταση και στην αρχή της γραμμής ανάγνωσης. Αν τα σχόλια παραμένουν ανοιχτά και ο λεκτικός αναλυτής διαβάσει EOF τότε μεταβαίνει σε κατάσταση λάθους.

```
# --- State_7: State_0 -> '/*' symbol --- #
if (state == state_7):
    if (input == '*'):
        state = state_8
    elif (input == ''):
        state = ERROR_state
        print('Syntax Error: Please make sure to close your comments!')
        print('Line -> %d:%d' % (line, in_line_position))
        in_line_position += 1
    elif (input == '\t'):
        in_line_position += 4
    elif (input == '\n'):
        state = state_0
        line += 1
        in_line_position = 0
    else:
        input = file_pointer.read(1)      #EOF checkpoint
        in_line_position += 1
        if (input == ''):
            state = ERROR_state
            print('Syntax Error: Please make sure to close your comments!')
            print('Line -> %d:%d' % (line, in_line_position))
        else:
            file_pointer.seek(file_pointer.tell()-1)
            in_line_position -= 1
```

Κατάσταση_8: Κατάσταση_0 + σύμβολο “*/”:

Με βάση το αυτόματο, όταν ο λεκτικός αναλυτής διαβάσει το σύμβολο “*/” μεταβαίνει στην **κατάσταση_8**.

1. **“*” + “/” = “*/”:** Ο λεκτικός αναλυτής αναγνωρίζει το σύμβολο “*/”, το αποθηκεύει σε μια λεκτική μονάδα την οποία επιστρέφει και μεταβαίνει σε τελική κατάσταση. Στην περίπτωση αυτή πρόκειται για την λήξη σχολίων, οπότε οτιδήποτε ακολουθεί **ΔΕΝ** αναγνωρίζεται σαν σχόλια. Αν αναγνωρίσει “\n” αλλάζει γραμμή και επιστρέφει στην αρχική κατάσταση και στην αρχή της γραμμής ανάγνωσης, αν αναγνωρίσει “*” μετατίθεται μία θέση προς τα πίσω. Αν τα σχόλια παραμένουν ανοιχτά και ο λεκτικός αναλυτής διαβάσει EOF τότε μεταβαίνει σε κατάσταση λάθους.

```
# --- State_8: State_0 -> '*/' symbol --- #
if (state == state_8):
    input = file_pointer.read(1)
    in_line_position += 1
    if (input == '/'):
        state = state_0
    elif (input == '\n'):
        state = state_0
    elif (input == ''):
        state = ERROR_state
        print('Syntax Error: Please make sure to close your comments!')
        print('Line -> %d:%d' % (line, in_line_position))
    elif (input == '*'):
        state = state_8
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
    else:
        state = state_7
        file_pointer.seek(file_pointer.tell()-1)
        in_line_position -= 1
if (state == ERROR_state):
    verbal_unit = '-1'
    rtrn_verbal_unit = ['ERROR'] + [-1]
```

Προκειμένου να εξασφαλιστεί η εύρυθμη λειτουργία του Λεκτικού Αναλυτή και κατ' επέκταση του Μεταγλωττιστή, πραγματοποιείται έλεγχος για να διαπιστωθεί εάν το αναγνωριστικό ταυτίζεται με κάποια από τις δεσμευμένες λέξεις της γλώσσας Starlet:

```
# Checking to see if the ID matches one of the Starlet reserved keywords #
for i in range (len(res_words)):
    if (identifier == res_words[i][0]):
        rtrn_verbal_unit.pop()
        rtrn_verbal_unit += [res_words[i][1]]
        break

file_pointer.close()
return rtrn_verbal_unit[0], rtrn_verbal_unit[1]
```

Συντακτικός Αναλυτής - Syntax Analyst:



Ο συντακτικός αναλυτής λειτουργεί ως εξής:

- Γίνεται έλεγχος για να διαπιστωθεί εάν το πηγαίο πρόγραμμα **ανήκει ή όχι** στη γλώσσα.
- Δημιουργεί το **κατάλληλο «περιβάλλον»** μέσα από το οποίο αργότερα θα κληθούν οι σημαντικές ρουτίνες.
- Υπάρχουν πολλοί τρόποι για να κατασκευαστεί ένας συντακτικός αναλυτής.
- Θα προτιμήσουμε τη συντακτική ανάλυση με **αναδρομική κατάβαση**.
- Βασίζεται σε **γραμματική LL(1)***.



*Η Γραμματική LL(1):

- **L** : Left to right
- **L** : Leftmost derivation
- **(1)** : One look-ahead symbol

Η γραμματική LL(1) αναγνωρίζει από **αριστερά στα δεξιά**, την **αριστερότερη δυνατή παραγωγή** και όταν βρίσκεται σε δίλλημα ποιον κανόνα να ακολουθήσει της αρκεί να κοιτάξει το **αμέσως επόμενο σύμβολο** στην συμβολοσειρά εισόδου.

Παράδειγμα:

```
S ::= while(condition) S  
S ::= print(expression)  
S ::= input(id)  
S ::= { S }
```

Όταν πρέπει να αναγνωριστεί το S τότε:

- ακολουθούμε τον πρώτο κανόνα αν η επόμενη λεκτική μονάδα στη είσοδο είναι το **while**,
- τον δεύτερο εάν είναι το **print**,
- τον τρίτο εάν είναι το **input**,
- και τον τέταρτο εάν είναι το **άνοιγμα αγκίστρου**.

Εσωτερική λειτουργία γραμματικής:

- Για κάθε έναν από τους **κανόνες** της γραμματικής, φτιάχνουμε και ένα αντίστοιχο **υποπρόγραμμα**.
- Όταν συναντάμε **μη τερματικό** σύμβολο **καλούμε** το αντίστοιχο υποπρόγραμμα.
- Όταν συναντάμε **τερματικό** σύμβολο, τότε:
 - Εάν και ο λεκτικός αναλυτής **επιστρέφει λεκτική μονάδα** που αντιστοιχεί στο τερματικό αυτό σύμβολο έχουμε αναγνωρίσει **επιτυχώς** τη λεκτική μονάδα.

- Αντίθετα εάν ο λεκτικός αναλυτής δεν επιστρέψει τη λεκτική μονάδα που περιμένει ο συντακτικός αναλυτής, έχουμε λάθος και καλείται ο διαχειριστής σφαλμάτων.
- **Όταν αναγνωριστεί και η τελευταία λέξη** του πηγαίου προγράμματος, τότε η συντακτική ανάλυση έχει στεφτεί με επιτυχία.

Η Γραμματική της Starlet:

```
<program>      ::= program id <block> endprogram
<block>        ::= <declarations> <subprograms> <statements>
<declarations> ::= (declare <varlist>;)*
<varlist>       ::= ε | id ( , id )*
<subprograms>  ::= (<subprogram>)*
<subprogram>   ::= function id <funcbody> endfunction
<funcbody>     ::= <formalpars> <block>
<formalpars>   ::= ( <formalparlist> )
<formalparlist> ::= <formalparitem> ( , <formalparitem> )* | ε
<formalparitem> ::= in id | inout id | inandout id
<statements>   ::= <statement> ( ; <statement> )*
<statement>    ::= ε |
                  <assignment-stat> |
                  <if-stat> |
                  <while-stat> |
                  <do-while-stat> |
                  <loop-stat> |
                  <exit-stat> |
                  <forcase-stat> |
                  <incase-stat> |
                  <return-stat> |
                  <input-stat> |
                  <print-stat>

<assignment-stat> ::= id := <expression>
<if-stat>      ::= if (<condition>) then <statements> <elsepart> endif
<elsepart>     ::= ε | else <statements>
<while-stat>   ::= while (<condition>) <statements> endwhile
<do-while-stat> ::= dowhile <statements> enddowhile (<condition>)
<loop-stat>    ::= loop <statements> endloop
<exit-stat>    ::= exit
```

```

<forcase-stat> ::= forcase
                  ( when (<condition>) : <statements> )*
                  default: <statements> enddefault
                  endforcase

<incase-stat> ::= incase
                  ( when (<condition>) : <statements> )*
                  endincase

<return-stat> ::= return <expression>
<print-stat> ::= print <expression>
<input-stat> ::= input id
<actualpars> ::= ( <actualparlist> )
<actualparlist> ::= <actualparitem> ( , <actualparitem> )* | ε
<actualparitem> ::= in <expression> | inout id | inandout id
<condition> ::= <boolterm> (or <boolterm>)*
<boolterm> ::= <boolfactor> (and <boolfactor>)*
<boolfactor> ::= not [<condition>] | [<condition>] |
                   f<expression> <relational-oper> <expression>
<expression> ::= <optional-sign> <term> ( <add-oper> <term> )*
<term> ::= <factor> (<mul-oper> <factor>)*
<factor> ::= constant | (<expression>) | id <idtail>
<idtail> ::= ε | <actualpars>
<relational-oper> ::= = | <= | >= | > | < | <>
<add-oper> ::= + | -
<mul-oper> ::= * | /
<optional-sign> ::= ε | <add-oper>

```

Η συνάρτηση του συντακτικού αναλυτή **def syntax_analyst()** λειτουργεί ως εξής:

Με βάση την γραμματική της Starlet καλούμε τον λεκτικό αναλυτή μία φορά και το εναποθέτουμε σε μία οντότητα (token).

- <program> ::= program id <block> endprogram:

Ο συντακτικός αναλυτής ελέγχει αν το πρώτο κομμάτι του token ταυτίζεται με τη δεσμευμένη λέξη “program”. Αν δεν τη βρει μας ενημερώνει με κατάλληλο μήνυμα λάθους και πραγματοποιεί έξοδο. Έπειτα καλεί τον λεκτικό αναλυτή και ελέγχει αν μετά το πρώτο γράμμα ακολουθεί γράμμα ή ψηφίο. Αν όχι τότε μας ενημερώνει με μήνυμα λάθους και πραγματοποιεί έξοδο. Αφού περάσει τους ελέγχους καλεί τον λεκτικό αναλυτή και τη συνάρτηση block(). Τέλος αφού εκτελεστεί η block, πραγματοποιεί την έλεγχο για την παρουσία της δεσμευμένης λέξης endprogram, η οποία πρέπει να υπάρχει πάντα στο τέλος του πηγαίου κώδικα.

```
# <program> ::= program id <block> endprogram #
def program():
    global token

    if (token[0] == 'program'):
        token = lex()           #Refill
        if (token[1] == 1):      #Checking to see if after the 1st letter fo
            token = lex()
            block()
        else:
            print('ERROR: Expected the name of the program instead of "{0}"')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
        if (token[0] == 'endprogram'):
            token = lex()
        else:
            print('ERROR: Expected the keyword "endprogram" instead of "{0}"')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected the keyword "program" instead of "{0}".' .fo
        print('Line -> {0}:{1}' .format(line, in_line_position))
        exit(-1)
    return
```

- **<block> ::= <declare> <subprograms> <statements>** :

Η συνάρτηση block() καλείται από τις συναρτήσεις program() και funcbody() και αναφέρεται στα λογικά blocks του πηγαίου κώδικα του προγράμματος. Με τη σειρά της η block() καλεί 3 συναρτήσεις, την declare(), την subprograms() και την statements().

```
# <block> ::= <declare> <subprograms> <statements> #
def block():
    global token

    declare()
    subprograms()
    statements()

    return
```

- **<declare> ::= (declare <varlist>;)*:**

Χρησιμοποιείται για τον έλεγχο της δήλωσης των μεταβλητών. Ελέγχει για την παρουσία της δεσμευμένης λέξης “declare”. Αν δεν υπάρχει μας εμφανίζει μήνυμα λάθους και πραγματοποιεί έξοδο. Αν υπάρχει καλεί τον λεκτικό αναλυτή και έπειτα, τη συνάρτηση varlist(). Έπειτα γίνεται έλεγχος για την ύπαρξη του ";" στο τέλος της δήλωσης. Αν δεν υπάρχει μας εμφανίζει μήνυμα λάθους και πραγματοποιεί έξοδο. Το "*" σημαίνει ότι μπορεί να έχουμε περισσότερες από μία χρήσεις του declare.

```
# <declare> ::= (declare <varlist>;)* #
def declare():
    global token

    while (token[0] == 'declare'):
        token = lex()
        varlist()
        if (token[0] == ';'):
            token = lex()
            return
        else:
            print('ERROR: Expected ";" after after the variable "{0}".'.format(token[0]))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **<varlist> ::= ε | id (, id)***:

Η συνάρτηση varlist() ελέγχει αν το αναγνωριστικό ταυτίζεται με το token Και καλεί τον λεκτικό αναλυτή. Ενόσω “διαβάζει” το σύμβολο “,” καλεί τον λεκτικό αναλυτή ελέγχοντας πάντα την ταυτοποίηση αναγνωριστικού-token. Σε οποιαδήποτε άλλη περίπτωση ενημερώνει τον χρήστη με τα κατάλληλα μηνύματα λάθους.

```
# <varlist> ::= ε | id ( , id )* #
def varlist():
    global token

    if (token[1] == 1):      #identifier (a.k.a. tokenID)
        token = lex()
        while (token[0] == ','):
            token = lex()
            if (token[1] == 1):
                token = lex()
            else:
                print('ERROR: Variable was expected before "{}".'.format(token[0]))
                print('Line -> {}:{}' .format(line, in_line_position))
                exit(-1)

    return
```

- **<subprograms> ::= (<subprogram>)*:**

Ενόσω το token που διαβάζεται ταυτίζεται με την δεσμευμένη λέξη “function” η συνάρτηση subprograms() καλεί την subprogram()

```
# <subprograms> ::= (<subprogram>)* #
def subprograms():
    global token

    while (token[0] == 'function'):
        subprogram()
    return
```

- **<subprogram> ::= function id <funcbody> endfunction:**

Η συνάρτηση subprogram() ελέγχει αν το token ταυτίζεται με την δεσμευμένη λέξη “function”. Εάν ο έλεγχος αποτιμηθεί ως αληθής, καλεί τον λεκτικό αναλυτή (για να “ξαναγεμίσει”). Αφού ελέγχει και την ταυτότητα του token, καλεί την συνάρτηση funcbody(). Αν κατά τον έλεγχο ανιχνεύσει ότι το επόμενο token είναι η δεσμευμένη Starllet “endfunction”, ξανακαλεί τον Λεκτικό Αναλυτή και τερματίζει, Άλλιώς, αν κάποιος από αυτούς τους ελέγχους δεν πραγματοποιηθεί σωστά ή ο

χρήστης κατά την πληκτρολόγηση παραλείψει κάτι ή προσθέσει κάτι περιπτώ , θα λάβει το απαραίτητο προειδοποιητικό μήνυμα λάθους στην οθόνη του.

```
# <subprogram> ::= function id <funcbody> endfunction #
def subprogram():
    global token

    if (token[0] == 'function'):
        token = lex()
        if (token[1] == 1):      #tokenID
            token = lex()
            funcbody()
            if (token[0] == 'endfunction'):
                token = lex()
            else:
                print('ERROR: Keyword "endfunction" was expected or you missed a semicolon ";".')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: ID number expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Keyword "function" was expected.')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return
```

- **<funcbody> ::= <formalpars> <block>**:

Η συνάρτηση funcbody() εφόσον κληθεί, με τη σειρά της καλεί τις συναρτήσεις: formalpars() & block(), έπειτα τερματίζει.

```
# <funcbody> ::= <formalpars> <block> #
def funcbody():
    global token

    formalpars()          #same syntax principles as <block>
    block()

    return
```

- <formalpars> ::= (<formalparlist>):

Μόλις κληθεί η συνάρτηση formalpars() πραγματοποιείται έλεγχος για το αν το πρώτο κομμάτι (token) που θα αναγνωστεί είναι το σύμβολο της αριστερής παρένθεσης “(”, τότε καλείται ο Λεκτικός Αναλυτής για να συνεχίσει τον έλεγχο. Στην περίπτωση που το token που θα διαβαστεί είναι μία από τις δεσμευμένες λέξεις “in”, “inout”, “inandout”, η συνάρτηση formalpars() καλεί την συνάρτηση formalparlist(). Για την βεβαίωση της σωστής σύνταξης γίνεται έλεγχος για να διαπιστωθεί αν μετά την κλήση της formalparlis() ακολουθεί δεξιά παρένθεση, αν ο έλεγχος περάσει τότε καλείται μία τελευταία φορά ο Λεκτικός Αναλυτής για να “ξαναγεμίσει” και να ξεκινήσει ξανά η διαδικασία από την αρχή. Πάλι σε περίπτωση συντακτικού λάθους, ο χρήστης θα λάβει την απαραίτητη προειδοποίηση.

```
# <formalpars> ::= ( <formalparlist> ) #
def formalpars():
    global token

    if (token[0] == '('):
        token = lex()
        if (token[0] == 'in' or token[0] == 'inout' or token[0] == 'inandout'):
            formalparlist()
            if (token[0] == ')'):
                token = lex()
                return
            else:
                print('ERROR: Expected right parenthesis ")" instead of "{}".'.format(token[0]))
                print('Line -> {}:{}' .format(line, in_line_position))
                exit(-1)
        elif (token[0] == ')'):
            token = lex()
            return
        else:
            print('ERROR: Expected keywords "in" or "inout" or "inandout" after the comma ",".'.format(token[0]))
            print('Line -> {}:{}' .format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected left parenthesis "(" instead of "{}".'.format(token[0]))
        print('Line -> {}:{}' .format(line, in_line_position))
        exit(-1)
    return
```

- <formalparitem> ::= in id | inout id | inandout id:

Κατά την κλήση της συνάρτησης formalparitem(), ελέγχουμε αν το πρώτο token ταυτίζεται με μία από τις δεσμευμένες λέξεις “in”, “inout”, “inandout”. Εάν ταυτίζεται με κάποια από αυτές τις λέξεις καλεί τον Λεκτικό Αναλυτή, ελέγχει και το αναγνωριστικό (identifier) για τον αν μετά ακολουθεί το όνομα της μεταβλητής, αν όλα πάνε καλά τότε καλείται ο Λεκτικός μία ακόμη φορά και έπειτα τερματίζει (αυτό ισχύει και για τις 3 λέξεις). Στην περίπτωση που μετά από κάποια από τις δεσμευμένες λέξεις δεν ακολουθεί το όνομα της μεταβλητής, το πρόγραμμά μας θα εμφανίσει το σωστό μήνυμα λάθους.

```
# <formalparitem> ::= in id | inout id | inandout id #
def formalparitem():
    global token

    if (token[0] == 'in'):
        token = lex()
        if (token[1] == 1):
            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "in".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inout'):
        token = lex()
        if (token[1] == 1):
            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "inout".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inandout'):
        token = lex()
        if (token[1] == 1):
            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "inandout".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **<statements> ::= <statement> (; <statement>)*:**

Η συνάρτηση statements() μόλις κληθεί, με τη σειρά της καλεί την συνάρτηση statement(). Ενόσω η πρώτη οντότητα που διαβάζει ο Λεκτικός Αναλυτής είναι το σύμβολο semicolon “;”, καλείται ξανά ο Λεκτικός για ανάγνωση, μετά καλείται η συνάρτηση statement() και έπειτα η συνάρτηση τερματίζει.

```
# <statements> ::= <statement> ( ; <statement> )* #
def statements():
    global token

    statement()
    while (token[0] == ';'):
        token = lex()
        statement()
    return
```

- **<statement> ::= ε | <assingment-stat> | <if-stat> | <while-stat> | <do-while-stat> | <loop-stat> | <exit-stat> | <forcase-stat> | <incase-stat> | <return-stat> | <input-stat> | <print-stat>:**

Πρακτικά, η συνάρτηση statement() πραγματοποιεί πολλαπλούς ελέγχους για το statement που έκανε ο χρήστης. Αρχικά, ελέγχει αν καλείται η συνάρτηση assignment_stat() ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “if”, τότε καλεί την συνάρτηση if_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “while”, τότε καλεί την συνάρτηση while_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “dowhile”, τότε καλεί την συνάρτηση do_while_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “loop”, τότε καλεί την συνάρτηση loop_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “exit”, τότε καλεί την συνάρτηση exit_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “forcase”, τότε καλεί την συνάρτηση for_case_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “incase”, τότε καλεί την συνάρτηση in_case_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “return”, τότε καλεί την συνάρτηση return_stat(). Ή αν το πρώτο token ταυτίζεται με την

Δεσμεύμενη λέξη “input”, τότε καλεί την συνάρτηση input_stat(). Ή αν το πρώτο token ταυτίζεται με την δεσμεύμενη λέξη “print”, τότε καλεί την συνάρτηση print_stat(). Και έπειτα τερματίζει.

```
# <statement> ::= ε | <assignment-stat> | <if-stat> | <while-stat> | <do-while-stat>
#                           | <loop-stat> | <exit-stat> | <forcase-stat> | <incase-stat> |
#                           | <return-stat> | <input-stat> | <print-stat> #
def statement():
    global token

    if (token[1] == 1):
        assignment_stat()
    elif (token[0] == 'if'):
        if_stat()
    elif (token[0] == 'while'):
        while_stat()
    elif (token[0] == 'dowhile'):
        do_while_stat()
    elif (token[0] == 'loop'):
        loop_stat()
    elif (token[0] == 'exit'):
        exit_stat()
    elif (token[0] == 'forcase'):
        for_case_stat()
    elif (token[0] == 'incase'):
        in_case_stat()
    elif (token[0] == 'return'):
        return_stat()
    elif (token[0] == 'input'):
        input_stat()
    elif (token[0] == 'print'):
        print_stat()
    return
```

- <assignment-stat> ::= id := <expression>:

Η συνάρτηση assignment_stat() ελέγχει την ύπαρξη της ανάθεσης μίας τιμής ή έκφρασης σε μία μεταβλητή. Ξεκινώντας, καλείται ο Λεκτικός Αναλυτής προκειμένου να αρχίσει η ανάγνωση, αν κατά την ανάγνωση ανιχνεύσει το σύμβολο “:=”, τότε καλείται ξανά ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση expression() προκειμένου να περαστεί σωστά η μεταβλητή. Αν ο χρήστης αμελήσει να βάλει το σύμβολο της ανάθεσης πριν από την τιμή, θα ειδοποιηθεί καταλλήλως.

```
# <assignment-stat> ::= id := <expression> #
def assignment_stat():
    global token

    token = lex()
    if (token[0] == ':='):
        token = lex()
        expression()
        return
    else:
        print('ERROR: Expected ":" before "{}".'.format(token[0]))
        print('Line -> {}:{}' .format(line, in_line_position))
        exit(-1)
```

- <if-stat> ::= if (condition) then <statement> <elsepart> endif:

Η συνάρτηση if_stat(), πραγματοποιεί το λογικό μπλοκ ενός if statement. Καλείται στην αρχή της συνάρτησης ο Λεκτικός Αναλυτής, διότι πάντα θα υπάρχει ένα if token, αλλιώς ο έλεγχος δεν θα μπει ποτέ μέσα στην if_stat(). Ελέγχεται αν το token που διαβάζει είναι αριστερή παρένθεση “(“, εφόσον είναι προχωράει καλώντας τον Λεκτικό Αναλυτή και έπειτα την συνάρτηση condition(). Μετά, πραγματοποιείται έλεγχος για την ανίχνευση της δεξιάς παρένθεσης “)”, εφόσον βρεθεί ξανακαλεί τον Λεκτικό Αναλυτή. Αν το επόμενο token ταυτίζεται με την δεσμευμένη λέξη “then”, καλείται ο Λεκτικός Αναλυτής, η συνάρτηση statements() και η else_part() προκειμένου να πετύχουμε την σωστή δόμηση ενός λογικού if - block. Τέλος, ελέγχει αν μετά ακολουθεί η δεσμευμένη λέξη “endif”,

εφόσον ακολουθεί, καλεί τον lex() (Λεκτικό Αναλυτή) προκειμένου να ξαναγεμίσει και να είναι έτοιμος να αναγνώσει ξανά. Εάν δεν τηρηθούν οι απαραίτητες συντακτικές προϋποθέσεις, ο χρήστης θα ειδοοποιηθεί με μηνύματα λάθους καθώς και με το ακριβές σημείο του λάθους.

```
# <if-stat> ::= if (<condition>) then <statements> <elsepart> endif #
def if_stat():
    global token

# There is always an if token, otherwise it won't enter the function #
token = lex()
if (token[0] == '('):
    token = lex()
    condition()
    if (token[0] == ')'):
        token = lex()
        if (token[0] == 'then'):
            token = lex()
            statements()
            else_part()
            if (token[0] == 'endif'):
                token = lex()
            else:
                print('ERROR: Expected the keyword "endif" or a questionmark ";".')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected the keyword "then".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected right parenthesis ")" instead of "{0}".'.format(token[0]))
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
else:
    print('ERROR: Expected left parenthesis "(" instead of "{0}".'.format(token[0]))
    print('Line -> {0}:{1}'.format(line, in_line_position))
    exit(-1)
return
```

- <elsepart> ::= ε | else <statements>:

Η συνάρτηση else_part() πρακτικά βοηθά στην υλοποίηση του else-block του λογικού if που στοχεύουμε να υλοποιήσουμε. Κατά την κλήση της, η else_part() ελέγχει αν η οντότητα που μόλις αναγνώστηκε ταυτίζεται με την δεσμευμένη λέξη “else” που αυτομάτως σηματοδοτεί την συντακτική αλλά και λογική έναρξη του else-block. Αν και μόνο αν την ανιχνεύσει, καλείται ο Λεκτικός Αναλυτής, στη συνέχεια η συνάρτηση statements() και έπειτα τερματίζει.

```
# <elsepart> ::= ε | else <statements> #
def else_part():
    global token

    if (token[0] == 'else'):
        token = lex()
        statements()
    return
```

- <while-stat> ::= while (<condition>) <statements> <endwhile>:

Η συνάρτηση while_stat() υλοποιεί ένα while-loop. Πιο συγκεκριμένα, ελέγχει αν το πρώτο token είναι η δεσμευμένη λέξη “while”, αν ο έλεγχος αποτιμηθεί ως αληθής προχωρά και καλεί τον Λεκτικό Αναλυτή προκειμένου να συνεχίσει την ανάγνωση. Σε ένα βαθύτερο επίπεδο ελέγχει αν μετά τη λέξη “while” ακολουθεί το σύμβολο της αριστερής παρένθεσης “(”, εφόσον υπάρχει καλεί τον Λεκτικό Αναλυτή να διαβάσει ένα ακόμη token και εν συνεχεία καλείται η συνάρτηση condition() προκειμένου να παραμετροποιηθούν οι συνθήκες υπό τις οποίες θα λειτουργήσει ο while βρόγχος. Ο έλεγχος συνεχίζεται ψάχνοντας την δεξιά παρένθεση “)”, αν βρεθεί τότε σηματοδοτεί το “κλείσιμο” των συνθηκών και καλείται ξανά ο lex() προκειμένου να αναγνώσει τα απαραίτητα statements με την κλήση της συνάρτησης statements(). Τέλος, ελέγχει αν στη συνέχεια έπειται η δεσμευμένη λέξη “endwhile” που σηματοδοτεί το τέλος του while-loop, εάν βρεθεί τότε καλεί τον Λεκτικό μία ακόμη φορά για να “ξαναγεμίσει”. Σε περίπτωση που περιμένει να αναγνώσει το κατάλληλο σύμβολο και αυτό δεν συμβεί, η

έξοδος του προγράμματος θα ενημερώσει κατάλληλα τον χρήστη για το λάθος που έγινε αλλά και για την ακριβή του τοποθεσία κάνοντας έτσι την εκσφαλμάτωση πιο εύκολη.

```
# <while-stat> ::= while (<condition>) <statements> <endwhile> #
def while_stat():
    global token

    #token = lex()      na to dw to proxwraw askopa
    if (token[0] == 'while'):
        token = lex()
        if (token[0] == '('):
            token = lex()
            condition()
            if (token[0] == ')'):
                token = lex()
                statements()
                if (token[0] == 'endwhile'):
                    token = lex()
                else:
                    print('ERROR: Expected the keyword "endwhile" instead of "{0}.'.format(token[0]))
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected right parenthesis ")" instead of "{0}" in order to close the while-loop.'.format(token[0]))
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected left parenthesis "(" instead of "{0}" in order to open the while-loop.'.format(token[0]))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- <do-while-stat> ::= dowhile <statements> enddowhile (<condition>):

Η συνάρτηση do_while_stat() στην ουσία υλοποιεί ένα do-while-loop. Κατά την κλήση της, ελέγχει για την παρουσία της δεσμευμένης λέξης “dowhile”, εάν ανιχνευθεί καλεί τον Λεκτικό Αναλυτή και στη συνέχεια την συνάρτηση statements(). Αν αναγνωστεί η δεσμευμένη λέξη “enddowhile”, καλείται ο Λεκτικός Αναλυτής και ελέγχει για το αν ακολουθεί το σύμβολο της αριστερής παρένθεσης “(“ σηματοδοτώντας έτσι τη έναρξη των συνθηκών υπό των οποίων θα υλοποιηθεί το do-while-loop. Εφόσον υπάρχει παρένθεση, καλείται ξανά ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση condition(). Η ανίχνευση της δεξιάς παρένθεσης ”)” αντιπροσωπεύει την λήξη των συνθηκών και η συνάρτηση τερματίζει. Και σε αυτή την περίπτωση, το πρόγραμμά μας

είναι πλήρως εξοπλισμένο για να ειδοποιήσει τον χρήστη για τυχόν λάθη, όπως έλλειψη παρενθέσεων, απουσία κατάλληλων δεσμευμένων λέξεων κ.λ.π.

```
# <do-while-stat> ::= dowhile <statements> enddowhile (<condition>) #
def do_while_stat():
    global token

    if (token[0] == 'dowhile'):
        token = lex()
        statements()
        if (token[0] == 'enddowhile'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                condition()
                if (token[0] == ')'):
                    token = lex()
                    return
                else:
                    print('ERROR: Expected right parenthesis ")" instead of "{}" in order to close the while-loop.')
                    print('Line -> {}:{}' .format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "(" instead of "{}" in order to open the while-loop.')
                print('Line -> {}:{}' .format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected the keyword "while" instead of "{}' .format(token[0]))
            print('Line -> {}:{}' .format(line, in_line_position))
            exit(-1)
    return
```

- <loop-stat> ::= loop <statements> endloop:

Η συνάρτηση loop_stat() υλοποιεί ένα iteration, δηλαδή ένα σετ επαναλήψεων. Η συνάρτηση -εφόσον κληθεί- ξεκινά να ελέγχει για την ύπαρξη της δεσμευμένης λέξης “loop”. Εάν τη βρει, τότε καλεί τον Λεκτικό Αναλυτή και έπειτα την συνάρτηση statements() προκειμένου να μπουν οι απαραίτητες δηλώσεις για την επανάληψη. Το τέλος της επανάληψης σηματοδοτεί η δεσμευμένη λέξη “endloop”, η οποία μόλις αναγνωστεί έχει ως αποτέλεσμα την κλήση του Λεκτικού Αναλυτή ώστε να είναι έτοιμος για την επόμενη ανάγνωση. Σε περίπτωση απουσίας της λέξης “endloop”, πρόκειται για μία αέναη επανάληψη και ο χρήστης θα ειδοποιηθεί κατάλληλα για το λάθος αυτό.

```
# <loop-stat> ::= loop <statements> endloop #
def loop_stat():
    global token

    if (token[0] == 'loop'):
        token = lex()
        statements()
        if (token[0] == 'endloop'):
            token = lex()
        else:
            print('ERROR: OPEN LOOP – Expected the keyword "endloop".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **<exit-stat> ::= exit:**

Η συνάρτηση exit_stat() απλώς πραγματοποιεί έξοδο μόλις διαπιστώσει ότι το token ταυτίζεται με την δεσμευμένη λέξη “exit”.

```
# <exit-stat> ::= exit #
def exit_stat():
    global token

    token = lex()

    return
```

- <forcase-stat> ::= **forcase**

(**when** (<condition>) : <statements>)*

default: <statements> **enddefault**

endforcase:

Η συνάρτηση `for_case_stat()` είναι το αντίστοιχο for-loop. Ξεκινά ελέγχοντας για την ύπαρξη της δεσμευμένης λέξης “`forcase`”, εφόσον βρεθεί καλείται ο Λεκτικός Αναλυτής. Όσο το token που διαβάζει είναι η δεσμευμένη λέξη “`when`”, καλείται ο Λεκτικός Αναλυτής. Μέσα σε αυτόν τον έλεγχο, εμφωλευμένα ελέγχει αν ακολουθεί αριστερή παρένθεση “(”, εφόσον ο έλεγχος ολοκληρωθεί επιτυχώς, ξανακαλείται ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση `condition()` προκειμένου να εισαχθούν οι συνθήκες υπό τις οποίες θα γίνεται το for-loop. Έπειτα ελέγχει αν ακολουθεί δεξιά παρένθεση ")" που είναι σημάδι λήξης των συνθηκών και αν ο έλεγχος ολοκληρωθεί επιτυχώς τότε καλείται ξανά ο Λεκτικός Αναλυτής. Μετά διαβάζοντας, αν το τρέχον token είναι το σύμβολο ":" αυτό σημαίνει ότι ακολουθούν οι δηλώσεις οπότε καλείται ξανά ο `lex()` και έπειτα η συνάρτηση `statements()`. Οι έλεγχοι που πραγματοποιούνται σε κάθε επίπεδο μπορούν να ανιχνεύσουν και τυχόν λάθη, ενημερώνοντας έτσι τον χρήστη. Στη συνέχεια, αν διαβάσει την δεσμευμένη λέξη “`default`”, καλεί τον Λεκτικό Αναλυτή ο οποίος ελέγχει αν μετά ακολουθεί το σύμβολο “：“, καλώντας εν συνεχείᾳ την συνάρτηση `statements()`. Τέλος, αν το token που θα διαβαστεί είναι η δεσμευμένη λέξη “`enddefault`”, αυτομάτως σηματοδοτείται η λήξη των `default statements` και η συνάρτηση `for_case_stat()` τερματίζει. Και σε αυτό το σημείο σε περίπτωση λάθους, ο χρήστης θα ενημερωθεί κατάλληλα.

[for_case_stat\(\) screenshot](#):

```

# <forcase-stat> ::= forcase
#                               ( when (<condtion>) : <statements> )*
#                               default: <statements> enddefault
#                               endforcase #
def for_case_stat():
    global token

    if (token[0] == 'forcase'):
        token = lex()
        while (token[0] == 'when'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                condition()
                if (token[0] == ')'):
                    token = lex()
                    if (token[0] == ':'):
                        token = lex()
                        statements()
                    else:
                        print('ERROR: Expected ":" symbol.')
                        print('Line -> {0}:{1}'.format(line, in_line_position))
                        exit(-1)
                else:
                    print('ERROR: Expected right parenthesis ")" .')
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "(" .')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        if (token[0] == 'default'):
            token = lex()
            if (token[0] == ':'):
                token = lex()
                statements()
                if (token[0] == 'enddefault'):
                    token = lex()
                else:
                    print('ERROR: Keyword "enddefault" expected.')
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected ":" symbol.')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Keyword "default" expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return

```

- <incase-stat> ::= incase
(when <condition> : <statements>)*
endincase:

Κατά την κλήση της συνάρτησης `in_case_stat()` πραγματοποιείται έλεγχος για το αν το token είναι η δεσμευμένη λέξη “`incase`”. Αν ισχύει αυτή η συνθήκη, καλείται ο Λεκτικός Αναλυτής και ενόσω το token που διαβάζεται αυτή τη στιγμή είναι η δεσμευμένη λέξη “`when`”, καλείται ο Λεκτικός Αναλυτής. Στη συνέχεια, ελέγχει αν ακολουθεί αριστερή παρένθεση “(”, εφόσον συμβαίνει αυτό καλείται ξανά ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση `condtion()` έτσι ώστε να εισαχθούν οι συνθήκες του loop. Μετά ελέγχει αν ακολουθεί δεξιά παρένθεση ")" σηματοδοτώντας το “κλείσιμο” των συνθηκών και ξανακαλεί τον Λεκτικό Αναλυτή. Έπειτα πριν την κλήση της συνάρτησης `statements()` πρέπει να ελέγχει αν το token είναι το σύμβολο “：“. Στο σημείο αυτό, αν υπάρχει κάποια συντακτική δυσαναλογία, το πρόγραμμα θα εμφανίσει τα ανάλογα μηνύματα λάθους. Στη συνέχεια, αν διαβάσει την δεσμευμένη λέξη “`endincase`”, καλεί τον Λεκτικό Αναλυτή και η συνάρτηση `in_case_stat()` τερματίζει. Και σε αυτό το σημείο σε περίπτωση λάθους, ο χρήστης θα ενημερωθεί κατάλληλα.

[in_case_stat\(\) screenshot](#):

```

# <incase-stat> ::= incase
#                                     ( when <condition> : <statements> )*
#                                         endincase #
def in_case_stat():
    global token

    if (token[0] == 'incase'):
        token = lex()
        while (token[0] == 'when'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                condition()
                if (token[0] == ')'):
                    token = lex()
                    if (token[0] == ':'):
                        token = lex()
                        statements()
                    else:
                        print('ERROR: Expected ":" symbol.')
                        print('Line -> {0}:{1}'.format(line, in_line_position))
                        exit(-1)
                else:
                    print('ERROR: Expected right parenthesis ")" .')
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "(" .')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        if (token[0] == 'endincase'):
            token = lex()
            return
    else:
        print('ERROR: Expected the keyword "endincase".')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
return

```

- <return-stat> ::= return <expression>:

Η συνάρτηση return_stat() ελέγχει την επιστροφή τιμών από συναρτήσεις ή λειτουργίες. Ξεκινά ελέγχοντας αν το token είναι η δεσμευμένη Starlet λέξη “return”. Εάν η λέξη ανιχνευθεί επιτυχώς, καλείται ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση expression() προκειμένου να γίνει η αποτίμηση της έκφρασης που επιστρέφεται.

```
# <return-stat> ::= return <expression> #
def return_stat():
    global token

    if (token[0] == 'return'):
        token = lex()
        expression()
    return
```

- <print-stat> ::= print <expression>:

Η συνάρτηση print_stat() εκτυπώνει κομμάτι του κώδικα, ελέγχοντας αν το token είναι η δεσμευμένη Starlet λέξη “print”. Εάν η λέξη ανιχνευθεί επιτυχώς, καλείται ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση expression() προκειμένου να εκτυπωθεί το αποτέλεσμα.

```
# <print-stat> ::= print <expression> #
def print_stat():
    global token

    if (token[0] == 'print'):
        token = lex()
        expression()
    return
```

- <input-stat> ::= input id:

Κατά την κλήση της συνάρτησης `input()`, γίνεται έλεγχος για το αν το token που διαβάστηκε ταυτοποιείται με την δεσμευμένη λέξη “`input`”. Αν ο έλεγχος γίνει σωστά, καλείται στη συνέχεια ο Λεκτικός Αναλυτής. Αν το αναγνωριστικό είναι ίδιο τότε ξανακαλείται ο Λεκτικός Αναλυτής και η συνάρτηση τερματίζει. Σε περίπτωση λάθους, ο χρήστης ειδοποιείται για την απουσία ονόματος της λειτουργίας.

```
# <input-stat> ::= input id #
def input_stat():
    global token

    if (token[0] == 'input'):
        token = lex()
        if (token[1] == 1):
            token = lex()
            return
        else:
            print('ERROR: Expected the name of the function.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **<actualpars> ::= (<actualparlist>):**

Η συνάρτηση actual_pars() κατά την εκκίνησή της ελέγχει αν το token που διαβάζεται είναι η αριστερή παρένθεση “(”, εφόσον ισχύει αυτό καλείται στη συνέχεια ο Λεκτικός Αναλυτής. Έπειτα, ελέγχεται αν το token που ακολουθεί είναι μία από τις δεσμευμένες λέξεις “in”, “inout”, “inandout”, στην περίπτωση που ταυτίζεται με μία από αυτές καλείται η συνάρτηση actual_par_list(). Τέλος, ελέγχει ξανά αν ακολουθεί δεξιά παρένθεση “)” και εάν ισχύει αυτό καλείται ο Λεκτικός Αναλυτής. Σε περίπτωση που παραλείπεται η δεξιά παρένθεση το πρόγραμμα θα εμφανίσει το αντίστοιχο μήνυμα λάθους.

```
# <actualpars> ::= ( <actualparlist> ) #
def actual_pars():
    global token

    if (token[0] == '('):
        token = lex()
        if (token[0] == 'in' or token[0] == 'inout' or token[0] == 'inandout'):
            actual_par_list()
            if (token[0] == ')'):
                token = lex()
            else:
                print('ERROR: Expected right parenthesis ")".')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
    return
```

- **<actual_par_list> ::= <actualparitem> (, <actualparitem>)* | ε:**

Η συνάρτηση actual_par_list() καλεί την συνάρτηση actual_par_item(). Ενόσω η λεκτική μονάδα που διαβάζεται είναι το κόμμα “,”, καλείται ο Λεκτικός Αναλυτής και μετά η actual_par_item().

```
# <actualparlist> ::= <actualparitem> ( , <actualparitem> )* | ε #
def actual_par_list():
    global token

    actual_par_item()
    while (token[0] == ','):
        token = lex()
        actual_par_item()
    return
```

- **<actual_par_item> ::= in <expression> | inout id | inandout id:**

Κατά την κλήση της συνάρτησης actual_par_item(), αρχικά εφαρμόζονται πολλαπλοί ταυτόχρονοι έλεγχοι. Αν το token που διαβάζεται είναι η δεσμευμένη λέξη “in”, καλείται ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση expression(). Άλλιώς αν το token είναι η δεσμευμένη λέξη “inout” καλείται ο Λεκτικός Αναλυτής και ελέγχει για την ταυτοποίηση του αναγνωριστικού (identifier). Και η τρίτη περίπτωση είναι αν η δεσμευμένη λέξη είναι η “inandout”, τότε καλείται ο Λεκτικός Αναλυτής και ελέγχει επίσης για την ταυτοποίηση του αναγνωριστικού. Σε κάθε σημείο των ελέγχων, το πρόγραμμα μπορεί να ανιχνεύσει και να ειδοποιήσει τον χρήστη για τα αντίστοιχα λάθη.

```
# <actual_par_item> ::= in <expression> | inout id | inandout id #
def actual_par_item():
    global token

    if (token[0] == 'in'):
        token = lex()
        expression()
    elif (token[0] == 'inout'):
        token = lex()
        if (token[1] == 1):
            token = lex()
        else:
            print('ERROR: Token ID was expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inandout'):
        token = lex()
        if (token[1] == 1):
            token = lex()
        else:
            print('ERROR: Token ID was expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Missing one of the reserved words "in | inout | inandout".')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return
```

- **<condition> ::= boolterm (or <boolterm>)*:**

Μόλις κληθεί η συνάρτηση condition(), με την σειρά της καλεί την συνάρτηση bool_term(). Όσο συνεχίζει το token να ταυτίζεται πλήρως με την δεσμευμένη λέξη “or” καλείται ο Λεκτικός Αναλυτής και έπειτα η συνάρτηση bool_term().

```
# <condition> ::= <boolterm> (or <boolterm>)* #
def condition():
    global token

    bool_term()
    while (token[0] == 'or'):
        token = lex()
        bool_term()
    return
```

- **<boolterm> ::= <boolfactor> (and <boolfactor>)*:**

Κατά την κλήση της συνάρτησης bool_term(), καλείται πρώτα η συνάρτηση bool_factor(). Όσο συνεχίζει το token να ταυτίζεται πλήρως με την δεσμευμένη λέξη “and” καλείται ο Λεκτικός Αναλυτής και έπειτα η συνάρτηση bool_factor().

```
# <boolterm> ::= <boolfactor> (and <boolfactor>)* #
def bool_term():
    global token

    bool_factor()
    while (token[0] == 'and'):
        token = lex()
        bool_factor()
    return
```

- **<boolfactor> ::= not [<condition>] | [<condition>] | <expression> <relational-oper> <expression>:**

Στην κλήση της συνάρτησης `bool_factor()`, αρχικά εφαρμόζεται έλεγχος για την ταυτοποίηση του token που διαβάστηκε με τη δεσμευμένη λέξη “`not`”. Εφόσον ο έλεγχος είναι αληθής καλείται ο Λεκτικός Αναλυτής προκειμένου να συνεχιστεί η ανάγνωση και έπειτα ελέγχεται αν το token που διαβάζεται είναι η αριστερή αγκύλη “[”. Εάν ισχύει, καλείται ξανά ο Λεκτικός Αναλυτής και ακολούθως η συνάρτηση `condition()` για την εισαγωγή των συνθηκών της `bool_factor()`. Σταματάει όταν διαβάσει τη δεξιά αγκύλη ”]”, γεγονός που σηματοδοτεί τη λήξη των συνθηκών, καλεί τον Λεκτικό Αναλυτή και τερματίζει. Αν το token που διαβάστηκε δεν ταυτίζεται με τη δεσμευμένη λέξη “`not`” αλλά με την αριστερή αγκύλη “[”, καλεί τον Λεκτικό Αναλυτή και κατόπιν τη συνάρτηση `condition()`, ώστε να εισαχθούν οι συνθήκες. Έπειτα ελέγχεται αν ακολουθεί δεξιά αγκύλη ”]”, σηματοδοτώντας το “`κλείσιμο`” των συνθηκών και ξανακαλεί τον Λεκτικό Αναλυτή. Τέλος, αν δε διαβάσει ούτε την δεσμευμένη λέξη “`not`” ούτε το άνοιγμα αγκύλης , καλεί τις συναρτήσεις `expression()`, `relational-oper()` και ξανά την `expression()` και τερματίζει. Σε κάθε σημείο των ελέγχων, το πρόγραμμα μπορεί να ανιχνεύσει και να ειδοποιήσει τον χρήστη για τα αντίστοιχα λάθη.

```

# <boolfactor> ::= not [<condition>] | [<condition>] |
#                                <expression> <relational-oper> <expression> #
def bool_factor():
    global token

    if (token[0] == 'not'):
        token = lex()
    if (token[0] == '['):
        token = lex()
        condition()
        if (token[0] == ']'):
            token = lex()
            return
        else:
            print('ERROR: Expected right square bracket "]" after the condition.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected left square bracket "[" after the condition.')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    elif (token[0] == '['):
        token = lex()
        condition()
        if (token[0] == ']'):
            token = lex()
            return
        else:
            print('ERROR: Expected right square bracket "]" after the condition.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        expression()
        relational_oper()
        expression()
return

```

- **<expression> ::= <optional-sign> <term> (<add-oper> <term>)*:**

Κατά την κλήση της συνάρτησης expression(), καλούνται πρώτα οι συναρτήσεις optional_sign() και term(). Όσο το token ταυτίζεται πλήρως με το σύμβολο “+” ή το σύμβολο “-” καλούνται οι συναρτήσεις add_oper() και term(). Όταν το token θα πάψει να ταυτίζεται με κάποιο από τα σύμβολα, τότε η συνάρτηση τερματίζει.

```
# <expression> ::= <optional-sign> <term> ( <add-oper> <term> )* #
def expression():
    global token

    optional_sign()
    term()
    while (token[0] == '+' or token[0] == '-'):
        add_oper()
        term()
    return
```

- **<term> ::= <factor> (<mul-oper> <factor>)*:**

Η συνάρτηση term() μετά την κλήση της, καλεί με τη σειρά της τη συνάρτηση factor(). Έπειτα όσο το token ταυτίζεται πλήρως με το σύμβολο “*” ή το σύμβολο “/” καλούνται οι συναρτήσεις mul_oper() και factor(). Τη στιγμή που το token θα ταυτιστεί με διαφορετικό σύμβολο πραγματοποιείται ο τερματισμός της term().

```
# <term> ::= <factor> ( <mul-oper> <factor> )* #
def term():
    global token

    factor()
    while (token[0] == '*' or token[0] == '/'):
        mul_oper()
        factor()
    return
```

- **<factor> ::= constant | (<expression>) | id <idtail>:**

Όταν καλείται η συνάρτηση factor(), αρχικά εφαρμόζονται πολλαπλοί ταυτόχρονοι έλεγχοι. Αν το token που διαβάζεται είναι ψηφίο, τότε καλείται ο Λεκτικός Αναλυτής προκειμένου να συνεχιστεί η ανάγνωση και τερματίζει. Διαφορετικά, αν το token ταυτίζεται με την αριστερή παρένθεση "(" καλείται ο Λεκτικός Αναλυτής, έπειτα η συνάρτηση expression(). Τη στιγμή που διαβάσει το κλείσιμο της παρένθεσης ")" καλεί ξανά το Λεκτικό Αναλυτή και τερματίζει. Στην περίπτωση που το token ταυτίζεται με το αναγνωριστικό id καλείται ο Λεκτικός Αναλυτής και στη συνέχεια η συνάρτηση id_tail() και τέλος πραγματοποιεί τερματισμό. Στην περίπτωση που δε διαβαστεί κάτι από τα παραπάνω το πρόγραμμα ειδοποιεί τον χρήστη για το συγκεκριμένο λάθος. Αυτό συμβαίνει και σε κάθε σημείο των ελέγχων, αν το πρόγραμμα ανιχνεύσει κάποιο συντακτικό λάθος.

```
# <factor> ::= constant | (<expression>) | id <idtail> #
def factor():
    global token

    if (token[1] == 2):      #digit = integer -> constant
        token = lex()
        return
    elif (token[0] == '('):
        token = lex()
        expression()
        if (token[0] == ')'):
            token = lex()
            return
        else:
            print('ERROR: Expected right parenthesis ")" after the expression.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[1] == 1):    #identifier (ID)
        token = lex()
        id_tail()
        return
    else:
        print('ERROR: Expected constant or expression or variable instead of "%s".' % token[0])
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return
```

- **<idtail> ::= ε | <actualpars>:**

Η συνάρτηση id_tail(), αρχικά ελέγχει αν το token που διαβάστηκε σηματοδοτεί το άνοιγμα παρένθεσης. Αν συμβαίνει αυτό τότε καλείται η συνάρτηση actual_pars() και η id_tail() τερματίζει.

```
# <idtail> ::= ε | <actualpars> #
def id_tail():
    global token

    if (token[0] == '(':    #<actualpars> token -> "(" open parenthesis symbol
        actual_pars()
        return

# <relational-oper> ::= = | <= | >= | > | < | <> #
def relational_oper():
    global token
```

- **<relational-oper> ::= = | <= | >= | > | < | <>:**

Αφού κληθεί η συνάρτηση relational_oper(), πραγματοποιούνται διαδοχικοί έλεγχοι για την ταύτιση του token που διαβάστηκε με κάποιο από τα σύμβολα “=”, “≤”, “≥”, “>”, “<” ή “<>”. Αν ο έλεγχος είναι αληθής τότε καλείται ο Λεκτικός Αναλυτής και τερματίζει. Στην περίπτωση που η αποτίμηση του ελέγχου είναι ψευδής τότε το πρόγραμμα ειδοποιεί το χρήστη με το κατάλληλο μήνυμα λάθους και πραγματοποιεί έξοδο.

```
# <relational-oper> ::= = | <= | >= | > | < | <> #
def relational_oper():
    global token

    if (token[0] == '='):
        token = lex()
    elif (token[0] == '<='):
        token = lex()
    elif (token[0] == '>='):
        token = lex()
    elif (token[0] == '>'):
        token = lex()
    elif (token[0] == '<'):
        token = lex()
    elif (token[0] == '<>'):
        token = lex()
    else:
        print('ERROR: Missing: "=" | "<=" | ">=" | ">" | "<" | "<>" !')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return
```

- **<add-oper> ::= + | - :**

Με την κλήση της συνάρτηση add_oper(), πραγματοποιούνται δύο έλεγχοι. Αν το token που διαβάζεται ταυτίζεται το σύμβολο "+" ή το σύμβολο "-". Σε κάθε μια από τις δύο περιπτώσεις, αν ο έλεγχος είναι αληθής, καλείται ο Λεκτικός Αναλυτής και η συνάρτηση ολοκληρώνεται.

```
# <add-oper> ::= + | - #
def add_oper():
    global token

    if (token[0] == '+'):
        token = lex()
    elif (token[0] == '-'):
        token = lex()
    return
```

- **<mul-oper> ::= * | / :**

Η συνάρτηση mul_oper(), εφαρμόζει δύο έλεγχους. Ο πρώτος ελέγχει αν το token που διαβάζεται είναι το σύμβολο "*" και ο δεύτερος αν είναι το σύμβολο "/". Όταν ο έλεγχος είναι αληθής, σε κάθε περίπτωση, καλείται ο Λεκτικός Αναλυτής και η συνάρτηση τερματίζει.

```
# <mul-oper> ::= * | / #
def mul_oper():
    global token

    if (token[0] == '*'):
        token = lex()
    elif (token[0] == '/'):
        token = lex()
    return
```

- **<optional-sign> ::= ε | <add-oper>:**

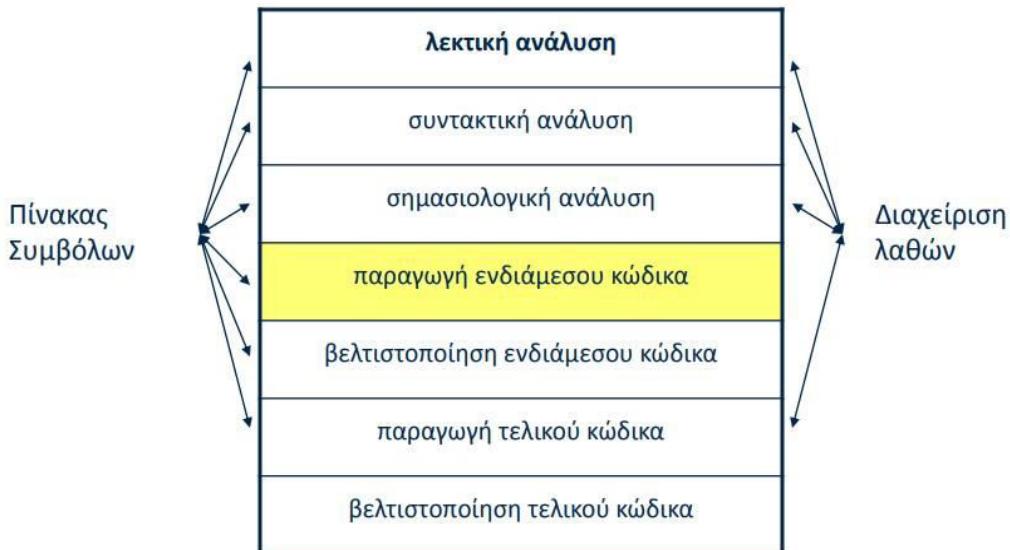
Όταν καλείται η συνάρτηση optional_sign(), ελέγχεται αν το token που διαβάστηκε είναι είτε το σύμβολο "+" είτε το σύμβολο "-". Στην περίπτωση που υπάρξει ταύτιση, καλείται η συνάρτηση add_oper() και τερματίζει.

```
# <optional-sign> ::= ε | <add-oper> #
def optional_sign():
    global token

    if (token[0] == '+' or token[0] == '-'):      #uses the tokens of <add_oper>
        add_oper()
        return

program()
print('-----')
```

Παραγωγή Ενδιάμεσου Κώδικα - Intermediate Code Production:



Από τον Συντακτικό Αναλυτή στον Ενδιάμεσο Κώδικα:



Ενδιάμεσος Κώδικας:

Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες, οι οποίες αποτελούνται από:

- Έναν τελεστή
- Τρία τελούμενα

Παράδειγμα:

+ , a,b,t_1

Τελεστής: +

Τελούμενα: a,b,t_1

Οι τετράδες αυτές είναι αριθμημένες. Πιο συγκεκριμένα κάθε τετράδα έχει μπροστά της ένα μοναδικό ο οποίος τη χαρακτηρίζει. Μετά την ολοκλήρωση μιας τετράδας, εκτελείται η τετράδα με τον αμέσως μεγαλύτερο αριθμό, εκτός από την περίπτωση που η τετράδα που μόλις εκτελέστηκε μας υποδείξει κάτι διαφορετικό.

Παράδειγμα:

100: +,a,b,c

Χαρακτηριστικός Αριθμός: 100

Τελεστές:

Διακρίνουμε τρεις κατηγορίες τελεστών:

- Τελεστές Αριθμητικών Πράξεων
- Τελεστή Εκχώρησης
- Τελεστή Άλματος χωρίς Συνθήκη

Τελεστές Αριθμητικών Πράξεων:

Οι τετράδες είναι της μορφής:

ορ, x, y, z

Ερμηνεία:

Εφαρμόζεται ο τελεστής ορ στα τελούμενα x και y και το αποτέλεσμα εκχωρείται στο τελούμενο z, όπου

- Το ορ μπορεί να είναι ένα εκ των: + , - , * , /
- Τα τελούμενα x,y μπορεί να είναι:
 - ◆ ονόματα μεταβλητών
 - ◆ αριθμητικές σταθερές
- Το τελούμενο z μπορεί να είναι:
 - ◆ όνομα μεταβλητής

Παράδειγμα:

- +,a,b,c αντιστοιχεί στην πράξη: $c=a+b$
- /,a,b,c αντιστοιχεί στην πράξη: $c=a/b$

Τελεστής Εκχώρησης:

Οι τετράδες είναι της μορφής:

`:=, x, _, z`

Ερμηνεία:

Η μορφή αυτή αντιστοιχεί στην εκχώρηση `z:=x`. Η τιμή του `x` εκχωρείται στη μεταβλητή `z`, όπου:

- Το τελούμενο `x` μπορεί να είναι:
 - ◆ Όνομα μεταβλητής
 - ◆ Αριθμητική σταθερά
- Το τελούμενο `z` μπορεί να είναι:
 - ◆ Όνομα μεταβλητής

Παράδειγμα:

Κώδικας Starlet:

`r:=4`

`pi:=3.14`

`area = pi * r * r`

Ισοδύναμος Ενδιάμεσος Κώδικας:

`100: :=,4,_r`

`110: :=,3.14,_pi`

`120: *,pi,r,T_1`

`130: *,T_1,r,area`

Τελεστής Άλματος χωρίς Συνθήκη:

Ο τελεστής άλματος χωρίς συνθήκη μπορεί να έχει δύο μορφές:

- Μεταπήδηση χωρίς συνθήκη

Έχει τη μορφή:

jump, _, _, z

Ερμηνεία:

Γίνεται μεταπήδηση χωρίς όρους στη θέση z.

Παράδειγμα:

100: :=,1,_x

110: jump 130

120: :=,2,_x

130 ...

Όταν φτάσουμε στο 130 η τιμή του x θα είναι 1 και όχι 2.

- Μεταπήδηση με συνθήκη

Έχει τη μορφή:

relop, x, y, z

Ερμηνεία:

Γίνεται μεταπήδηση στη θέση z αν ισχύει η x relop y.

Παράδειγμα:

100: =,a,4,120

110: jump,_,_,140

120: :=,1,_b

130: jump 150

140: :=,2,_b

150: ...

Το b θα έχει την τιμή 1 αν ισχύει η συνθήκη a=4 και 2 αν δεν ισχύει.

Αρχή και Τέλος Ενότητας:

- begin_block, name, _, _

Αρχή του προγράμματος ή του υποπρογράμματος με το όνομα name.

- end_block, name, _, _

Τέλος του προγράμματος ή του υποπρογράμματος με το όνομα name.

- halt , _, _, _

Τερματισμός υποπρογράμματος.

Παράδειγμα:

100: begin_block, add, _, _

110: := , 1 , _, x

120: := , 2 , _, y

130: + , x , y , z

140: halt , _, _, _

150: end_block , add , _, _

Συναρτήσεις-Διαδικασίες:

- Παράμετροι Συνάρτησης

Μορφή: par, x, m, _

Όπου x παράμετρος συνάρτησης και m ο τρόπος μετάδοσης.

Τρόποι Μετάδοσης Παραμέτρων:

- ◆ CV : μετάδοση με τιμή
- ◆ REF: μετάδοση με αναφορά
- ◆ RET: επιστροφή τιμής συνάρτησης

- Κλήση Συνάρτησης

Μορφή: call, name, _, _

Όπου name το όνομα της συνάρτησης.

- Επιστροφή τιμής Συνάρτησης

Μορφή: ret, x, _, _

Όπου x η τιμή που επιστρέφεται.

Παράδειγμα κλήσης συνάρτησης:

x := foo (in a, inout b)

Ισοδύναμος ενδιάμεσος κώδικας:

100: par , a , cv , _

110: par , b , ref , _

120: par , T_1 , ret , _

130: call , foo , _ , _ , _

140: ... τιμή στο x

Παράδειγμα κλήσης διαδικασίας:

call foo (in a, inout b)

Ισοδύναμος ενδιάμεσος κώδικας:

100: par , a , cv , _

110: par , b , ref , _

120: call , foo , _ , _ , _

130 ...

Βοηθητικές Υπορουτίνες:

- **nextquad():**

Η υπορουτίνα nextquad() επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί quad_ID.

```
# Subroutine nextQuad(): returns the quad_ID of the next quad #
def nextquad():
    global quad_ID
    return quad_ID
```

- **genquad(op, x, y, z):**

Η υπορουτίνα genquad() δημιουργεί την επόμενη τετράδα (op, x, y, z) που πρόκειται να παραχθεί. Αρχικά δημιουργεί μια κενή λίστα. Έπειτα με την κλήση της υπορουτίνας nextquad() προσθέτει μέσα σε αυτή τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί. Αφού προσθέσει και τα τελούμενα της τετράδας αυτής αυξάνει τον χαρακτηριστικό αριθμό(quad_ID) και προσθέτει τη λίστα που δημιούργησε στη λίστα με όλες τις τετράδες που έχουν χρησιμοποιηθεί από το πρόγραμμα. Μετά την ολοκλήρωσή της επιστρέφει τη λίστα με την τετράδα που δημιουργήθηκε.

```
# Subroutine genquad(op, x, y, z): creates the next quad (op, x, y, z) #
def genquad(op, v1, v2, v3):
    global quad_ID
    global quad_list

    list = []
    list = [nextquad()]           # adding quad number
    list += [op] + [v1] + [v2] + [v3]   # adding arguments

    quad_ID += 1                 # next_quad = current_quad + 1
    quad_list += [list]          # adding current list to the list with all the quads used by the program

    return list
```

- **newtemp():**

Η υπορουτίνα newtemp() δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή της μορφής :T_1, T_2, T_3 ... Η υπορουτίνα δημιουργεί μια μεταβλητή της μορφής T_x και μια κενή λίστα για τις προσωρινές μεταβλητές. Στη συνέχεια δημιουργεί μια βοηθητική λίστα μέσα στην οποία προσθέτει την μεταβλητή. Έπειτα συνδέει τα στοιχεία της λίστας σε 1 συμβολοσειρά και τα αποθηκεύει σε μια μεταβλητή(tmp_var) και αυξάνει την τιμής της μεταβλητής T_x κατά ένα ώστε σε επαναληπτική κλήση να μπει στη λίστα η επόμενη μεταβλητή. Τέλος προσθέτει στη λίστα των προσωρινών μεταβλητών τη μεταβλητή που δημιουργήθηκε, την οποία μεταβλητή και επιστρέφει.

```
# Subroutine newtemp(): creates & returns a temporary variable (T_x) #
def newtemp():
    global T_x
    global tmp_variables_list

    T_x = 1
    tmp_variables_list = []

    list = ['T_']
    list.append(str(T_x))
    tmp_var = "".join(list)
    T_x += 1

    tmp_variables_list += [tmp_var]

    return tmp_var
```

- **emptylist():**

Η υπορουτίνα emptylist() δημιουργεί μία κενή λίστα ετικετών τετράδων, την οποία και επιστρέφει.

```
# Subroutine emptylist(): creates an empty list of quad labels #
def emptylist():
    label_list = []

    return label_list
```

- **makelist(x):**

Η υπορουτίνα makelist() δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x και την επιστρέφει.

```
# Subroutine makelist(x): creates a list of quad labels including only "x" #
def makelist(x):
    current_list = [x]

    return current_list
```

- **merge(list1, list):**

Η υπορουτίνα mergelist() παίρνει σαν όρισμα δύο λίστες list1 και list2. Δημιουργεί μία νέα κενή λίστα μέσα στην οποία τοποθετεί τη συνένωση των δύο λιστών και την επιστρέφει την καινούρια λίστα.

```
# Subroutine mergelist(l1, l2): creates a list of quad labels merging the two lists #
def mergelist(list_1, list_2):
    list = []
    list += list_1 + list_2

    return list
```

- **backpatch(list,z):**

Η λίστα list αποτελείται από δείκτες σε τετράδες των οποίων το τελευταίο τελούμενο δεν είναι συμπληρωμένο. Παίρνει σαν όρισμα μια λίστα(list) και μια ετικέτα(z). Διατρέχει τη λίστα με τις τετράδες και συμπληρώνει κάθε μια από αυτές με την ετικέτα z.

```
# Subroutine backpatch(list, z): searches for quads with void last variable and fills them with the "z" label
def backpatch(list, z):
    global quad_list

    for i in range(len(list)):
        for j in range(len(quad_list)):
            if (list[i] == quad_list[j][0] and quad_list[j][4] == '_'):
                quad_list[j][4] = z
                #j = len(quad_list)

    return
```

Αρχή και Τέλος block:

- **Program():**

Η συνάρτηση program() αποθηκεύει το token[0], το οποίο είναι το όνομα του προγράμματος σε μια μεταβλητή. Έπειτα καλεί τη συνάρτηση block() με ορίσματα το όνομα του προγράμματος και ένα flag το οποίο δείχνει ότι δε βρισκόμαστε μέσα σε κάποιο block κώδικα.

```
# <program> ::= program id <block> endprogram #
def program():
    global token

    if (token[0] == 'program'):
        token = lex()           #Refill
        if (token[1] == 1):      #Checking to see if after the 1st letter follows another letter or digit
            prog_name = token[0]
            token = lex()
            block(prog_name, 1)
        else:
            print('ERROR: Expected the name of the program instead of "{}".'.format(token[0]))
            print('Line -> {}:{}' .format(line, in_line_position))
            exit(-1)
    if (token[0] == 'endprogram'):
        token = lex()
    else:
        print('ERROR: Expected the keyword "endprogram" instead of "{}".'.format(token[0]))
        print('Line -> {}:{}' .format(line, in_line_position))
        exit(-1)
    else:
        print('ERROR: Expected the keyword "program" instead of "{}".'.format(token[0]))
        print('Line -> {}:{}' .format(line, in_line_position))
        exit(-1)
    return
```

- **Block():**

Η συνάρτηση block() παίρνει σαν όρισμα το όνομα ενός block κώδικα και ένα flag το οποίο δείχνει αν το block αναφέρεται σε όλο το πρόγραμμα ή σε κάποιο υποπρόγραμμα. Αφού κληθούν οι συναρτήσεις declarations() και subprograms(), καλείται η υπορουτίνα genquad() η οποία δημιουργεί μια τετράδα που σηματοδοτεί την έναρξη ενός block. Ελέγχει αν το block αναφέρεται στο καθολικό block. Αν ναι, τότε καλείται η υπορουτίνα genquad(), η οποία δημιουργεί μια τετράδα που σηματοδοτεί τον τερματισμό του προγράμματος. Άλλιώς, καλείται για να δημιουργήσει μια τετράδα που σηματοδοτεί το τέλος του block.

```
# <block> ::= <declare> <subprograms> <statements> #
def block(block_name, mainProgBlockFlag):
    global token

    declarations()
    subprograms()
    genquad('begin_block', block_name, '_', '_')
    statements()
    if (mainProgBlockFlag == 1):
        genquad('halt', '_', '_', '_')
    genquad('end_block', block_name, '_', '_')

    return
```

- **Subprogram():**

Η συνάρτηση block() αποθηκεύει το token[0], που είναι το όνομα της διαδικασίας και αφού κληθεί ο Λεκτικός Αναλυτής, καλείται η συνάρτηση funcbody(), με ορίσματα το όνομα της διαδικασίας και ένα flag το οποίο δηλώνει ότι πρόκειται για συνάρτηση.

```
# <subprogram> ::= function id <funcbody> endfunction #
def subprogram():
    global token

    if (token[0] == 'function'):
        token = lex()
        if (token[1] == 1):      # tokenID
            name = token[0]    # Keep the name of the procedure
            token = lex()
            funcbody(name, 1)  # Because it is a function
            if (token[0] == 'endfunction'):
                token = lex()
            else:
                print('ERROR: Keyword "endfunction" was expected or you missed a semicolon ";"')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
            else:
                print('ERROR: ID number expected.')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Keyword "function" was expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **Funcbody():**

Η συνάρτηση funcbody() καλείται με ορίσματα το όνομα της συνάρτησης και ένα flag. Αφού κληθεί η συνάρτηση formal_pars() καλείται η block() με ορίσματα το όνομα της συνάρτησης και ένα flag το οποίο δηλώνει πως δεν αποτελεί το καθολικό block, αλλά μέρος αυτού.

```
# <funcbody> ::= <formalpars> <block> #
def funcbody(block_name, func):
    global token

    formalpars()          #same syntax principles as <block>
    block(block_name, -1) #not a main program block

    return
```

Εκχώρηση:

- **assignment_stat -> id := expression {P1}**

Αφού κληθεί η συνάρτηση assignment_stat() το token[0], το οποίο στην παρούσα φάση αποτελεί το αναγνωριστικό μια μεταβλητής, εκχωρείται σε μία μεταβλητή. Μετά τους συντακτικούς ελέγχους, καλείται η συνάρτηση expression(), και η τιμή που επιστρέφει εκχωρείται σε μια νέα μεταβλητή. Στη συνέχεια ελέγχει αν πρόκειται για συνάρτηση. Αν ναι, τότε καλείται η υπορουτίνα genquad(), η οποία δημιουργεί μια τετράδα που αντιστοιχεί στην εκχώρηση μιας τιμής επιστροφής συνάρτησης σε μια μεταβλητή και μηδενίζει το flag. Άλλιως, καλείται για να δημιουργήσει την τετράδα που αντιστοιχεί στην εκχώρηση μιας τιμής/έκφρασης σε μια μεταβλητή.

```
# <assignment-stat> ::= id := <expression> #
# INTR_CODE: assignment_stat -> id := expression {P1} #
def assignment_stat():
    global token
    global tmp
    global flag

    id = token[0] #  
  
    token = lex()
    if (token[0] == ':='):
        token = lex()
        E_place = expression() #  
        #{P1}:
        if (flag == 1): #whether it is a function
            genquad(':=', tmp, '_', id)
            flag = 0
        else:
            genquad(':=', E_place, '_', id)
    else:
        print('ERROR: Expected ":" before "{}".format(token[0]))')
        print('Line -> {}:{}' .format(line, in_line_position))
        exit(-1)
    return
```

- **if_stat -> if cond then {P1} statements {P2} else_part endif {P3}:**

Στη δομή if η επιστρεφόμενη τιμή της συνάρτησης condition() αποθηκεύεται σε μια μεταβλητή. Αφού εκτελεστούν οι απαιτούμενοι συντακτικοί έλεγχοι, πριν κληθεί η συνάρτηση statements(), καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες ισχύουν. Στη συνέχεια κατασκευάζεται μια λίστα με την επόμενη τετράδα που πρόκειται να παραχθεί και καλείται η υπορουτίνα genquad() για να εξασφαλίσουμε ότι αν εκτελεστεί ο κώδικας του if δεν θα εκτελεστεί ο κώδικας του else. Έπειτα, πριν κληθεί η συνάρτηση else_part() καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες δεν ισχύουν. Τέλος μετά το διάβασμα της δεσμευμένης λέξης "endif", καλείται η υπορουτίνα backpatch() για να εξασφαλίσουμε ότι αν εκτελεστεί ο κώδικας του if δεν θα εκτελεστεί ο κώδικας του else. Έπειτα δημιουργεί δύο λίστες μέσα στις οποίες τοποθετεί τις επιστρεφόμενες τιμές της συνάρτησης condition() (true και false αντίστοιχα) και τις επιστρέφει.

```
# <if-stat> ::= if (<condition>) then <statements> <elsepart> endif #
# INTR_CODE: if_stat -> if cond then {P1} statements {P2} else_part endif {P3} #
def if_stat():
    global token

    # There is always an if token, otherwise it won't enter the if_stat() function #
    token = lex()
    if (token[0] == '('):
        token = lex()
        cond = condition()
        if (token[0] == ')'):
            token = lex()
            if (token[0] == 'then'):
                token = lex()
                #{P1}:
                backpatch(cond[0], nextquad())           #cond[0] = TRUE
                statements()
                #{P2}:
                is_list = makelist(nextquad())
                genquad('JUMP', '_', '_', '_')
                backpatch(cond[1], nextquad())           #cond[1] = FALSE
                else_part()
                if (token[0] == 'endif'):
                    token = lex()
                    #{P3}:
                    backpatch(is_list, nextquad())
                else:
                    print('ERROR: Expected the keyword "endif" or a questionmark ";".')
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected the keyword "then".')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected right parenthesis ")" instead of "{0}".'.format(token[0]))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected left parenthesis "(" instead of "{0}".'.format(token[0]))
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)

    ifStat_TRUE = cond[0]
    ifStat_FALSE = cond[1]

    return ifStat_TRUE, ifStat_FALSE
```

- **while_stat -> while ({P1} cond) {P2} statements endwhile {P3}:**

Στη δομή while(πριν την κλήση της συνάρτησης condition()) καλείται η υπορουτίνα nextquad() και η τιμή της εκχωρείται σε μια μεταβλητή και έπειτα καλείται η συνάρτηση condition(), της οποίας η τιμή εκχωρείται και αυτή σε μια μεταβλητή. Στη συνέχεια, πριν την κλήση της συνάρτησης statements() καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες ισχύουν. Ακολούθως καλείται η υπορουτίνα genquad() για να μεταβεί στην αρχή της συνθήκης ώστε να πραγματοποιηθεί ξανά έλεγχος. Όταν θα πάψουν να ισχύουν οι συνθήκες τότε καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες δεν ισχύουν. Τέλος, δημιουργεί δύο λίστες μέσα στις οποίες τοποθετεί τις επιστρεφόμενες τιμές της συνάρτησης condition() (true και false αντίστοιχα) και τις επιστρέφει.

```
# <while-stat> ::= while (<condition>) <statements> endwhile #
# INTR_CODE: while_stat -> while ({P1} cond) {P2} statements endwhile {P3}#
def while_stat():
    global token

    #token = lex()    na to dw to proxwraw askopa
    if (token[0] == 'while'):
        token = lex()
        if (token[0] == '('):
            token = lex()
            #{P1}:
            cond_quad = nextquad()
            cond = condition()
            if (token[0] == ')'):
                #{P2}:
                backpatch(cond[0], nextquad())           #cond[0] = TRUE
                token = lex()
                statements()
                if (token[0] == 'endwhile'):
                    token = lex()
                    #{P3}:
                    genquad( JUMP , - , - , cond_quad)
                    backpatch(cond[1], nextquad())         #cond[1] = FALSE
                else:
                    print('ERROR: Expected the keyword "endwhile" instead of "{}".'.format(token[0]))
                    print('Line -> {}:{}' .format(line, in_line_position))
                    exit(-1)
                else:
                    print('ERROR: Expected right parenthesis ")" instead of "{}" in order to close the while-loop.' .format(token[0]))
                    print('Line -> {}:{}' .format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "(" instead of "{}" in order to open the while-loop.' .format(token[0]))
                print('Line -> {}:{}' .format(line, in_line_position))
                exit(-1)

    whileStat_TRUE = cond[0]
    whileStat_FALSE = cond[1]

return whileStat_TRUE, whileStat_FALSE
```

- **do_while_stat -> dowhile {P1} statements enddowhile (cond {P2}):**

Στη συνάρτηση do_while() πριν την κλήση της συνάρτησης statements(), καλείται η υπορουτίνα nextquad() και η τιμή της εκχωρείται σε μια μεταβλητή. Μετά την κλήση της συνάρτησης condition() καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες ισχύουν και μεταβαίνει στην αρχή για να ξανά πραγματοποιήσει τον έλεγχο των συνθηκών. Τη στιγμή που θα σταματήσουν να ισχύουν οι συνθήκες καλείται η υπορουτίνα backpatch() για να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες. Πριν ολοκληρωθεί η διαδικασία δημιουργεί δύο λίστες μέσα στις οποίες τοποθετεί τις επιστρεφόμενες τιμές της συνάρτησης condition() (true και false αντίστοιχα) και τις επιστρέψει.

```
# <do-while-stat> ::= dowhile <statements> enddowhile (<condition>) #
# INTR_CODE: do_while_stat -> dowhile {P1} statements enddowhile (cond {P2}) #
def do_while_stat():
    global token

    if (token[0] == 'dowhile'):
        token = lex()
        #{P1}:
        jumpX = nextquad() #jumpX variable shows where to jump
        statements()
        if (token[0] == 'enddowhile'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                cond = condition()
                #{P2}:
                backpatch(cond[0], jumpX) #cond[0] = TRUE
                backpatch(cond[1], nextquad()) #cond[0] = FALSE
                if (token[0] == ')'):
                    token = lex()
                    return
                else:
                    print('ERROR: Expected right parenthesis ")" instead of "{0}" in order to close the while-loop.'.format(token[0]))
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "(" instead of "{0}" in order to open the while-loop.'.format(token[0]))
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected the keyword "while" instead of "{0}".'.format(token[0]))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)

    dowhileStat_TRUE = cond[0]
    dowhileStat_FALSE = cond[1]

    return dowhileStat_TRUE, dowhileStat_FALSE
```

- **loop_stat -> loop {P1} statements endloop {P2}:**

Με την κλήση της συνάρτησης `loop_stat()` και πριν την κλήση της συνάρτησης `statements()` καλείται η υπορουτίνα `nextquad()` της οποίας την τιμή την εκχωρείται σε μια μεταβλητή. Μετά τη δεύτερη κλήση της συνάρτησης `statements()`(της οποίας η τιμή εκχωρείται και αυτή σε μια μεταβλητή)καλείται η υπορουτίνα `backpatch()` ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες ισχύουν και μεταβαίνει στην αρχή ώστε να ελεγχθούν ξανά οι συνθήκες. Όταν οι συνθήκες δε θα ισχύουν πια καλείται η υπορουτίνα `backpatch()` για να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες. Έπειτα δημιουργούνται δύο λίστες μέσα στις οποίες τοποθετεί τις επιστρεφόμενες τιμές της συνάρτησης `statements()` (true και false αντίστοιχα) και τις επιστρέφει.

```
# <loop-stat> ::= loop <statements> endloop #
# (?) INTR_CODE: loop_stat -> loop {P1} statements endloop {P2} #
def loop_stat():
    global token

    #[P1]:
    jumpX = nextquad()
    token = lex()
    statements()

    if (token[0] == 'loop'):
        token = lex()
        stat = statements()
        if (token[0] == 'endloop'):
            token = lex()
            #[P2]:
            backpatch(stat[0], jumpX)           #stat[0] = TRUE
            backpatch(stat[1], nextquad())      #stat[1] = FALSE
        else:
            print('ERROR: OPEN LOOP - Expected the keyword "endloop".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)

    loopStat_TRUE = stat[0]
    loopStat_FALSE = stat[1]

    return loopStat_TRUE, loopStat_FALSE
```

- **for_case_stat -> {P0} forcase**
(when {P1} (cond) : {P2} statements)
default: statements enddefault
endforcase {P3}:

Στη δομή for case πριν τον έλεγχο για την δεσμευμένη λέξη "forcase" φτιάχνει μια άδεια λίστα(καλώντας την υπορουτίνα emptylist()). Πριν κληθεί η συνάρτηση condition()(της οποίας η τιμή αποθηκεύεται σε μια μεταβλητή)αποθηκεύει τον χαρακτηριστικό αριθμό της επόμενης τετράδας που θα παραχθεί σε μια μεταβλητή. Όταν ισχύουν οι συνθήκες ισχύουν καλείται η υπορουτίνα backpatch() και ακολούθως η συνάρτηση statements(). Έπειτα υλοποιείται μια λίστα με τον χαρακτηριστικό αριθμό της τετράδας που πρόκειται να παραχθεί και πραγματοποιείται μεταπήδηση στην αρχή για επανεξέταση των συνθηκών, με την κλήση των αντίστοιχων υπορουτινών. Στην περίπτωση που δεν ισχύουν συνενώνει την κενή λίστα και την λίστα με τον χαρακτηριστικό αριθμό και την αποθηκεύει στην μεταβλητή που είχε αποθηκεύσει την κενή λίστα. Στη συνέχεια καλείται η υπορουτίνα backpatch() ώστε να συμπληρωθούν οι ασυμπλήρωτες τετράδες. Τέλος αφού διαβάσει τη δεσμευμένη λέξη enddefault καλείται η υπορουτίνα backpatch().

```
# <forcase-stat> ::= forcase
#                               ( when (<condition>) : <statements> )*
#                               default: <statements> enddefault
#                               endforcase #
# INTR_CODE: for_case_stat -> {P0} forcase
#                               ( when {P1} (cond) : {P2} statements )
#                               default: statements enddefault
#                               endforcase {P3} #
def for_case_stat():
    global token

    #{P0}:
    exit_list = emptylist()

    if (token[0] == 'forcase'):
        token = lex()
        while (token[0] == 'when'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                #{P1}:
                cond_quad = nextquad()
                cond = condition()
                if (token[0] == ')'):
                    token = lex()
                    if (token[0] == ':'):
                        token = lex()
                        #{P2}:
                        backpatch(cond[0],nextquad())
                        statements()
                        tmp_list = makelist(nextquad())
                        genquad('JUMP', '_', '_', '_')
                        exit_list = mergelist(exit_list, tmp_list)
                else:
                    print("Error: Expected colon after WHEN")
                    exit(1)
```

```

        backpatch(exit_list, nextquad())
    else:
        print('ERROR: Expected ":" symbol.')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    else:
        print('ERROR: Expected right parenthesis ")".')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    else:
        print('ERROR: Expected left parenthesis "(".')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
if (token[0] == 'default'):
    token = lex()
    if (token[0] == ':'):
        token = lex()
        statements()
        if (token[0] == 'enddefault'):
            token = lex()
            #{P3}:
            backpatch(exit_list, nextquad())
        else:
            print('ERROR: Keyword "enddefault" expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Expected ":" symbol.')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
else:
    print('ERROR: Keyword "default" expected.')
    print('Line -> {0}:{1}'.format(line, in_line_position))

```

- **incase_stat -> incase**
(when {P1} cond : {P2} statements)
endincase {P3}:

Στη δομή incase πριν την κλήση της συνάρτησης condition() καλείται η υπορουτίνα nextquad() και οι επιστρεφόμενες τιμές τους αποθηκεύονται σε μεταβλητές. Πριν κληθεί η συνάρτηση statements() καλείται η backpatch() ώστε να συμπληρωθούν οι τετράδες που έχουν μείνει ασυμπλήρωτες όταν οι συνθήκες ισχύουν. Αφότου εξεταστούν όλες οι συνθήκες εάν έστω και μία ισχύει μεταβαίνει στην αρχή της incase καλώντας την genquad(). Αν καμία από τις συνθήκες δεν ισχύει, καλείται η backpatch() για την συμπλήρωση των ασυμπλήρωτων τετράδων. Στο τέλος δημιουργούνται δύο λίστες μέσα στις οποίες τοποθετεί τις επιστρεφόμενες τιμές της συνάρτησης statements() (true και false αντίστοιχα) και τις επιστρέφει.

```

# <incase-stat> ::= incase
#           ( when (<condition>) : <statements> )*
#           endincase #
# INTR_CODE: incase_stat -> incase
#           ( when {P1} cond : {P2} statements )
#           endincase {P3} #
def in_case_stat():
    global token

    if (token[0] == 'incase'):
        token = lex()
        while (token[0] == 'when'):
            token = lex()
            if (token[0] == '('):
                token = lex()
                #{P1}:
                cond_quad = nextquad()
                cond = condition()
                if (token[0] == ')'):
                    token = lex()
                    if (token[0] == ':'):
                        #{P2}:
                        backpatch(cond[0], nextquad()) #cond[0] = TRUE
                        token = lex()
                        statements()
                    else:
                        print('ERROR: Expected ":" symbol.')
                        print('Line -> {0}:{1}'.format(line, in_line_position))
                        exit(-1)
                else:
                    print('ERROR: Expected right parenthesis ")"')
                    print('Line -> {0}:{1}'.format(line, in_line_position))
                    exit(-1)
            else:
                print('ERROR: Expected left parenthesis "("')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        if (token[0] == 'endincase'):
            token = lex()
            #{P3}:
            genquad('JUMP', '_', '_', cond_quad)
            backpatch(cond[1], nextquad()) #cond[1] = FALSE
        else:
            print('ERROR: Expected the keyword "endincase".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)

    inCaseStat_TRUE = cond[0]
    inCaseStat_FALSE = cond[1]

return inCaseStat_TRUE, inCaseStat_FALSE

```

- **return_stat -> return E {P1}:**

Στη δομή επιστροφής συνάρτησης, αφού καλεστεί ο Λεκτικός Αναλυτής η τιμή της συνάρτησης expression() αποθηκεύεται σε μια μεταβλητή. Ακολούθως καλείται η υπορουτίνα genquad() η οποία θα δημιουργήσει μία τετράδα με τελεστή που υποδηλώνει επιστροφή τιμής και τελεστή την μεταβλητή με την αποτίμηση της expression().

```
# <return-stat> ::= return <expression> #
# INTR_CODE: return_stat -> return E {P1} #
def return_stat():
    global token

    if (token[0] == 'return'):
        token = lex()
        #{P1}:
        E_place = expression()
        genquad('retv', E_place, '_', '_')
    return
```

- **print_stat -> print E {P1}:**

Αφότου κληθεί η συνάρτηση print_stat(), και καλεστεί ο Λεκτικός Αναλυτής η τιμή της συνάρτησης expression() αποθηκεύεται σε μια μεταβλητή. Στη συνέχεια καλείται η υπορουτίνα genquad() η οποία θα δημιουργήσει μία τετράδα με τελεστή που υποδηλώνει εκτύπωση τιμής και τελεστή την μεταβλητή με την αποτίμηση της expression().

```
# <print-stat> ::= print <expression> #
# INTR_CODE: print_stat -> print E {P1} #
def print_stat():
    global token

    if (token[0] == 'print'):
        token = lex()
        #{P1}:
        E_place = expression()
        genquad('out', E_place, '_', '_')    #is out a good name?
    return
```

- <actualpars> ::= (<actualparlist>):

Αφότου εκτελεστούν οι απαραίτητοι συντακτικοί έλεγχοι η συνάρτηση actual_pars() ελέγχει αν πρόκειται για συνάρτηση ή για διαδικασία. Στην πρώτη περίπτωση δημιουργεί μια προσωρινή μεταβλητή της μορφής T_x και την αποθηκεύει σε μια άλλη. Στη συνέχεια καλείται η υπορουτίνα genquad() η οποία παράγει την επόμενη τετράδα που δηλώνει ότι η προσωρινή μεταβλητή είναι παράμετρος συνάρτησης που επιστρέφεται. Ακολούθως ξανά καλείται παράγοντας τετράδα που δηλώνει ότι καλείται η συνάρτηση με το όνομα id_name και κρατάμε την προσωρινή μεταβλητή σε κάποια άλλη *. Άλλιως, καλείται παράγοντας τετράδα που δηλώνει ότι καλείται η διαδικασία με το όνομα id_name.

```
# <actualpars> ::= ( <actualparlist> ) #
def actual_pars(is_func, id_name):
    global token
    global tmp

    if (token[0] == '('):
        token = lex()
        if (token[0] == 'in' or token[0] == 'inout' or token[0] == 'inandout'):
            actual_par_list()
            if (token[0] == ')'):
                token = lex()
                #If it's a function:
                if (is_func == 1):
                    w = newtemp()
                    genquad('par', w, 'RET', '_')
                    genquad('call', id_name, '_', '_')
                    tmp = w
                #If it's a procedure:
                else:
                    genquad('call', id_name, '_', '_')
            else:
                print('ERROR: Expected right parenthesis ")".')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
    return
```

- **<actual_par_item> ::= in <expression> | inout id | inandout id:**

Η συνάρτηση actual_par_item() μετά την κλήση της expression(), η τιμή της οποίας εκχωρείται σε μια μεταβλητή, καλεί την υπορουτίνα genquad(), παράγοντας την επόμενη τετράδα που δηλώνει την μετάδοση της αποτίμησης της expression()(ως παράμετρος συνάρτησης) με τιμή.

Όταν διαβαστεί η δεσμευμένη λέξη "inout" καλείται για την παραγωγή της τετράδας που δηλώνει την μετάδοση της παραμέτρου token[0] με αναφορά. Η ίδια διαδικασία εκτελείται και όταν διαβάσει τη δεσμευμένη λέξη "inandout".

```

<actual_par_item> ::= in <expression> | inout id | inandout id #
def actual_par_item():
    global token

    if (token[0] == 'in'):
        token = lex()
        current_expression = expression()
        genquad('par', current_expression, 'CV', '_')
    elif (token[0] == 'inout'):
        token = lex()
        if (token[1] == 1):
            genquad('par', token[0], 'REF', '_')
            token = lex()
        else:
            print('ERROR: Token ID was expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inandout'):
        token = lex()
        if (token[1] == 1):
            genquad('par', token[0], 'REF', '_')
            token = lex()
        else:
            print('ERROR: Token ID was expected.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    else:
        print('ERROR: Missing one of the reserved words "in | inout | inandout".')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return

```

- Cond -> Boolterm1 {P1} (or {P2} Boolterm2 {P3}):

Κατασκευάζονται δύο λίστες οι οποίες έχουν την αποτίμηση της boolterm(), true και false αντίστοιχα. Για όσο ισχύει η συνθήκη της while καλείται η backpatch() για τη συμπλήρωση όσων τετράδων γίνεται μέσα στον κανόνα και κρατάει την τιμή της δεύτερης κλήσης της boolterm() σε μια μεταβλητή. Τέλος συνενώνει τις λίστες που περιέχουν τις τετράδες για την αληθή αποτίμηση της boolterm() στη λίστα true και βάζει στη λίστα false αυτές που αντιστοιχούν σε ψευδή αποτίμηση και τις επιστρέφει.

```
# <condition> ::= <boolterm> (or <boolterm>)* #
# INTR_CODE: Cond -> Boolterm1 {P1} (or {P2} Boolterm2 {P3}) #
def condition():
    global token

    cond_TRUE = []
    cond_FALSE = []

    bool_term1 = bool_term()
    #{P1}:
    cond_TRUE = bool_term1[0]
    cond_FALSE = bool_term1[1]

    while (token[0] == 'or'):
        token = lex()
        #{P2}:
        backpatch(cond_FALSE, nextquad())
        bool_term2 = bool_term()
        #{P3}:
        cond_TRUE = mergelist(cond_TRUE, bool_term2[0])
        cond_FALSE = bool_term2[1]

    return cond_TRUE, cond_FALSE
```

- **boolterm -> boolfactor1 {P1} (and {P2} boolfactor2 {P3}):**

Η boolterm() υλοποιεί δύο λίστες που περιέχουν αποτίμηση της bool_factor(), true και false αντίστοιχα. Όσο ισχύει η συνθήκη της while η boolterm() καλεί τη backpatch() για να συμπληρωθούν όσες τετράδες γίνεται μέσα στον κανόνα και κρατάει την τιμή της δεύτερης κλήσης της bool_factor() σε μια μεταβλητή. Τέλος συνενώνει τις λίστες που περιέχουν τις τετράδες για την αληθή αποτίμηση της bool_factor() στη λίστα true και βάζει στη λίστα false αυτές που αντιστοιχούν σε ψευδή αποτίμηση και τις επιστρέφει.

```
# <boolterm> ::= <boolfactor> (and <boolfactor>)* #
# INTR_CODE: boolterm -> boolfactor1 {P1} (and {P2} boolfactor2 {P3}) #
def bool_term():
    global token

    boolterm_TRUE = []
    boolterm_FALSE = []

    bool_factor1 = bool_factor()
    #{P1}:
    boolterm_TRUE = bool_factor1[0]
    boolterm_FALSE = bool_factor1[1]

    while (token[0] == 'and'):
        token = lex()
        #{P2}:
        backpatch(boolterm_TRUE, nextquad())
        bool_factor2 = bool_factor()
        #{P3}:
        boolterm_FALSE = mergelist(boolterm_FALSE, bool_factor2[1])
        boolterm_TRUE = bool_factor2[0]

    return boolterm_TRUE, boolterm_FALSE
```

- **boolfactor -> not [condition] {P1} or**
boolfactor -> [condition] {P1} or
boolfactor -> E1_place relop E2_place {P1}:

Στην πρώτη περίπτωση της συνάρτησης bool_factor() τοποθετούνται οι επιστρεφόμενες τιμές της συνάρτησης condition()(true και false) σε δύο λίστες true και false αντίστοιχα. Η ίδια διαδικασία ακολουθείται και για την δεύτερη περίπτωση. Στην τρίτη περίπτωση οι τιμές των συναρτήσεων expression() και relational_oper() αποθηκεύονται σε 3 μεταβλητές αντίστοιχα. Έπειτα τοποθετεί στη λίστα true τον αριθμό της επόμενης μη συμπληρωμένης τετράδας που θα παραχθεί και καλεί τη genquad() για να την παράξει, όταν η αποτίμηση της relop θα είναι αληθής. Όταν η αποτίμηση της relop θα είναι ψευδής θα δημιουργείται μη συμπληρωμένη τετράδα και θα τοποθετείται στην λίστα false. Στο τέλος η συνάρτηση bool_factor() επιστρέφει τις λίστες true και false.

```
# <boolfactor> ::= not [<condition>] | [<condition>] | <expression> <relational-oper>
def bool_factor():
    global token

    boolFactor_TRUE = []
    boolFactor_FALSE = []
    E_place1 = ''
    E_place2 = ''
    relop = ''

    # opposites!
    # INTR_CODE: boolfactor -> not [condition] {P1}
    if (token[0] == 'not'):
        token = lex()
        if (token[0] == '['):
            token = lex()
            cond = condition() # returns 2 lists (TRUE & FALSE)
            if (token[0] == ']'):
                token = lex()
                # {P1}:
                boolFactor_TRUE = cond[1] #cond[1] = FALSE
                boolFactor_FALSE = cond[0] #cond[0] = TRUE
            else:
                print('ERROR: Expected right square bracket "]" after the condition.')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
        else:
            print('ERROR: Expected left square bracket "[" after the condition.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    # NOT opposites!
    # INTR_CODE: boolfactor -> [condition] {P1}
    elif (token[0] != '['):
        token = lex()
        cond = condition()
        if (token[0] == ']'):
            token = lex()
            # {P1}:
            boolFactor_TRUE = cond[0] #cond[0] = TRUE
            boolFactor_FALSE = cond[1] #cond[1] = FALSE
        else:
            print('ERROR: Expected right square bracket "]" after the condition.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    # INTR_CODE: boolfactor -> E1_place relop E2_place {P1}
    else:
        E1_place = expression()
        #print(E1_place)
        relop = relational_oper()
        #print(relop)
        E2_place = expression()
        # {P1}:
        boolFactor_TRUE = makelist(nextquad())
        genquad(relop, E1_place, E2_place, '_') #needs backpatching!
        boolFactor_FALSE = makelist(nextquad())
        genquad('JUMP', '_', '_', '_') #needs backpatching!

    return boolFactor_TRUE, boolFactor_FALSE
```

- expression -> T1 (+- T2 {P1})* {P2}:

Η συνάρτηση expression() τοποθετεί σε μία μεταβλητή την αποτίμηση της term(). Για όσο ισχύουν οι συνθήκες του while και αφότου αποθηκεύσει τις τιμές των add_oper() και term() σε δύο μεταβλητές, δημιουργεί μια προσωρινή μεταβλητή η οποία θα κρατάει το παρόν αποτέλεσμα. Στη συνέχεια παράγεται μια τετράδα που αντιστοιχεί στην πρόσθεση ή αφαίρεση του παρόντος αποτελέσματος στο T2_place, που περιέχει την αποτίμηση της δεύτερης κλήσης της term() και το αποτέλεσμα τοποθετείται στην προσωρινή μεταβλητή που δημιουργήθηκε. Το παρόν αποτέλεσμα κρατείται σε μία μεταβλητή για την περίπτωση που υπάρχουν περισσότερες από δύο μεταβλητές χωρισμένες με "+" ή "-". Όταν θα προστεθούν ή αφαιρεθούν όλες οι μεταβλητές τότε το αποτέλεσμα αποθηκεύεται σε μία νέα μεταβλητή η οποία και επιστρέφεται από τη συνάρτηση.

```
# <expression> ::= <optional-sign> <term> ( <add-oper> <term> )* #
# INTR_CODE: expression -> T1 (+- T2 {P1})* {P2} #
def expression():
    global token

    optional_sign()
    T1_place = term()
    while (token[0] == '+' or token[0] == '-'):
        plus_minus = add_oper()
        T2_place = term()
        #{P1}:
        w = newtemp()
        genquad(plus_minus, T1_place, T2_place, w)
        T1_place = w
    #{P2}:
    E_place = T1_place

    return E_place
```

- term -> factor1 (muloper factor2 {P1})* {P2}:

Αφού κληθεί η συνάρτηση term(), τοποθετείται σε μία μεταβλητή η αποτίμηση της factor(). Εώς ότου σταματήσουν να ισχύουν οι συνθήκες του while, οι τιμές των mul_oper() και factor() κρατούνται σε δύο μεταβλητές και δημιουργείται μια προσωρινή μεταβλητή η οποία θα κρατάει το παρόν αποτέλεσμα. Έπειτα παράγεται μια τετράδα που αντιστοιχεί στον πολλαπλασιασμό ή την διαίρεση του παρόντος αποτελέσματος στο T2_place, που περιέχει την αποτίμηση της δεύτερης κλήσης της factor() και το αποτέλεσμα τοποθετείται στην προσωρινή μεταβλητή που δημιουργήθηκε. Το παρόν αποτέλεσμα κρατείται σε μια μεταβλητή για την περίπτωση που υπάρχουν περισσότερες από δύο μεταβλητές χωρισμένες με "*" ή "/". Όταν θα πολλαπλασιαστούν ή διαιρεθούν όλες οι μεταβλητές τότε το αποτέλεσμα αποθηκεύεται σε μία νέα μεταβλητή η οποία και επιστρέφεται από τη συνάρτηση.

```
# <term> ::= <factor> (<mul-oper> <factor>)* #
# INTR_CODE: factor1 (muloper factor2 {P1})* {P2} #
def term():
    global token

    F1_place = factor()
    while (token[0] == '*' or token[0] == '/'):
        mul_div = mul_oper()
        F2_place = factor()
        #{P1}:
        w = newtemp()
        genquad(mul_div, F1_place, F2_place, w)
        F1_place = w
    #{P2}:
    T_place = F1_place

    return T_place
```

- **<factor> ::= constant | (<expression>) | id <idtail>:**

Στην περίπτωση που το token που διαβάζεται είναι ψηφίο κρατάμε το αλφαριθμητικό της λεκτικής μονάδας σε μία μεταβλητή factorial. Διαφορετικά, αν ακολουθεί αριστερή παρένθεση "(" η αποτίμηση της συνάρτησης expression() τοποθετείται σε μια προσωρινή μεταβλητή και στη συνέχεια εκχωρείται σε μία μεταβλητή factorial. Άλλως, αν το token ταυτίζεται με το αναγνωριστικό id κρατάει το όνομα του αναγνωριστικού σε μία μεταβλητή factorial και καλεί τη συνάρτηση id_tail() με όρισμα αυτή τη μεταβλητή. Μετά την ολοκλήρωσή της επιστρέφεται η τιμή της factorial.

```
# <factor> ::= constant | (<expression>) | id <idtail>
def factor():
    global token

    if (token[1] == 2):          #digit = integer -> constant
        factorial = token[0]     #storing the string of the verbal unit
        token = lex()
        return
    elif (token[0] == '('):
        token = lex()
        E_place = expression()
        if (token[0] == ')'):
            factorial = E_place
            token = lex()
            return
        else:
            print('ERROR: Expected right parenthesis ")" after the expression.')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[1] == 1):          #identifier (ID)
        factorial = token[0]
        token = lex()
        id_tail(factorial)
        return
    else:
        print('ERROR: Expected constant or expression or variable instead of "%s".' % token[0])
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)

    return factorial
```

- **<idtail> ::= ε | <actualpars>:**

Η id_tail() καλείται με όρισμα το όνομα ενός αναγνωριστικού. Φτιάχνει ένα flag με τιμή 1 το οποίο δηλώνει ότι πρόκειται για συνάρτηση. Στη συνέχεια καλείται η συνάρτηση actual_pars() με ορίσματα το flag και το όνομα της συνάρτησης.

```
# <idtail> ::= ε | <actualpars> #
def id_tail(id_name):
    global token
    global flag

    if (token[0] == '('):           #<actualpars> token -> "(" open parenthesis symbol
        flag = 1
        actual_pars(1, id_name)     #1 means it's a function
        return
```

- **<relational-oper> ::= = | <= | >= | > | < | <>:**

Κατά την κλήση της συνάρτησης relational_oper() το token που διαβάστηκε τοποθετείται σε μια μεταβλητή relOp η οποία αποτελεί έναν από τους τελεστές άλματος και στο τέλος της συνάρτησης επιστρέφεται.

```
# <relational-oper> ::= = | <= | >= | > | < | <> #
def relational_oper():
    global token

    if (token[0] == '=' or token[0] == '<=' or token[0] == '>=' or token[0] == '>' or token[0] == '<' or token[0] == '<>'):
        relOp = token[0]
        token = lex()
    else:
        print('ERROR: Missing: "=" | "<=" | ">=" | ">" | "<" | "<>" !')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)
    return relOp
```

- <add-oper> ::= + | - :

Η add_oper() τοποθετεί το token που διάβασε σε μια μεταβλητή additional_op και μετά την ολοκλήρωση της επιστρέφεται. Η μεταβλητή αυτή αναφέρεται και στις δύο πράξεις αφού ξέρουμε ότι η αφαίρεση είναι πρόσθεση με αρνητικό αριθμό.

```
# <add-oper> ::= + | - #
def add_oper():
    global token

    if (token[0] == '+' or token[0] == '-'):
        addition_op = token[0] #addition_op is for both + & - because subtraction is a negative addition
        token = lex()

    return addition_op
```

- <mul-oper> ::= * | / :

Η mul_oper() τοποθετεί το token που διάβασε σε μια μεταβλητή multil_op και μετά την ολοκλήρωση της επιστρέφεται. Η μεταβλητή αυτή αναφέρεται και στις δύο πράξεις.

```
# <mul-oper> ::= * | / #
def mul_oper():
    global token

    if (token[0] == '*' or token[0] == '/'):
        multi_op = token[0]
        token = lex()

    return multi_op
```

- <optional-sign> ::= ε | <add-oper>

Η συνάρτηση optional_sign() εκχωρεί την επιστρεφόμενη τιμή της add_oper() σε μία μεταβλητή addition_substraction.

```
# <optional-sign> ::= ε | <add-oper> #
def optional_sign():
    global token

    if (token[0] == '+' or token[0] == '-'): #uses the tokens of <add_oper>
        addition_subtraction = add_oper()
        return
```

Εγγραφή σε αρχεία:

- **Εγγραφή Ενδιάμεσου Κώδικα σε αρχείο:**

Η συνάρτηση test_file() παίρνει όρισμα ένα αρχείο Starlet, από το οποίο ήδη έχει παραχθεί ο ενδιάμεσος κώδικας. Διατρέχει τη λίστα με τις τετράδες και γράφει τα στοιχεία της κάθε τετράδας στη μορφή:

Χαρακτηριστικός αριθμός: τελεστής τελούμενο1 τελούμενο2 τελούμενο3

Κάθε γραμμή του αρχείου καταλαμβάνεται αποκλειστικά από μια τετράδα. Το αρχείο θα αποθηκεύεται στη μορφή test.int.

```
# Writing all quads on test.int #
def test_file(test):
    for i in range(len(quad_list)):
        quad = quad_list[i]
        test.write(str(quad[0]))
        test.write(": ")
        test.write(str(quad[1]))
        test.write(" ")
        test.write(str(quad[2]))
        test.write(" ")
        test.write(str(quad[3]))
        test.write(" ")
        test.write(str(quad[4]))
        test.write("\n")
```

• Μετατροπή σε Κώδικα C και εγγραφή σε αρχείο:

Υλοποιείται με τη συνάρτηση `c_stuff()` που παίρνει όρισμα ένα αρχείο μέσα στο οποίο θα γραφτεί ο ισοδύναμος κώδικας C. Αν η λίστα με τις μεταβλητές έχει μέγεθος τότε γράφει τη λέξη " int ". Έπειτα διατρέχει τη λίστα με τις μεταβλητές, γράφει στο αρχείο τη μεταβλητή που διαβάζει, ελέγχει αν υπάρχει άλλη μεταβλητή . Αν ναι γράφει στο αρχείο το σύμβολο " , " , διαφορετικά γράφει το σύμβολο " ; ". Αυτή η διαδικασία αντιστοιχεί στη δήλωση μιας μεταβλητής ή στις δηλώσεις πολλών μεταβλητών χωρισμένων με κόμμα. Στη συνέχεια διατρέχει τη λίστα με τις τετράδες. Εκτελεί τους εξής ελεγχους:

- Αν ο τελεστής της τετράδας είναι το " `begin_block` " το οποίο σηματοδοτεί την έναρξη ενός block κώδικα. Γράφει στο αρχείο `L_x`, όπου x η γραμμή που βρίσκεται η τετράδα +1.
- Αν είναι ο τελεστής εκχώρησης " `:=` ", γράφει στο αρχείο το `L_x` και το σύμβολο " `:` ". Έπειτα το τελούμενο 3, το σύμβολο " `=` " το τελούμενο 1 και τέλος το σύμβολο " `;` ". Εστω το παράδειγμα:

`A:=1; -> Starlet`

(`:=`, 1, `,`, `a`) \rightarrow Ενδιάμεσος, όπου τελούμενο `1=1` και τελούμενο `3=a`

`a=1; -> Ισοδύναμος C`

- Αν είναι αριθμητικός τελεστής(4 διαφορετικές περιπτώσεις μέσα στον κώδικα) " `+` ", " `-` ", " `*` " ή " `/` " γράφει στο αρχείο το `L_x` και το σύμβολο " `:` ". Στη συνέχεια το τελούμενο 3 στο οποίο έχει αποθηκευτεί το αποτέλεσμα της πράξης, τα τελούμενα 1 και 2 και ανάμεσα τους τον αντίστοιχο αριθμητικό τελεστή. Τέλος γράφει το κόμμα στο τέλος της πράξης.
- Αν είναι ο τελεστής άλματος 'JUMP' γράφει στο αρχείο το `L_x` και το σύμβολο " `:` ". Ακολούθως την πρόταση `goto L_y`, όπου y είναι το τελούμενο 3 το οποίο υποδεικνύει σε ποια γραμμή κώδικα πρέπει να μεταβεί ο μεταγλωτιστής και στο τέλος το σύμβολο " `;` ".
- Αν είναι τελεστής σύγκρισης(6 διαφορετικές περιπτώσεις στον κώδικα) γράφει στο αρχείο το `L_x` και το σύμβολο " `:` " στο αρχείο το `L_x` και το σύμβολο " `:` ". Ακολουθεί η λέξη " `if` " και μέσα σε παρενθέσεις τα τελούμενα 1 και 2 χωρισμένα με τον τελεστή σύγκρισης. Έπειτα, στην ίδια γραμμή τυπώνει την πρόταση `goto L_y`, όπου y είναι το τελούμενο 3 το οποίο υποδεικνύει σε ποια γραμμή κώδικα πρέπει να μεταβεί ο μεταγλωτιστής και στο τέλος το σύμβολο " `;` ".

- Αν ο τελεστής δηλώνει έξοδο, δηλαδή τύπωμα τιμής ενός τελούμενου τότε γράφει στο αρχείο το L_x και το σύμβολο ”. Εν συνεχεία την πρόταση printf και μέσα σε παρενθέσεις την έκφραση μαζί με το format της τιμής της ως αλφαριθμητικό(σύνταξη print στη C), το οποίο να είναι ίσο με το τελούμενο 1(το οποίο είναι η τιμής της έκφρασης). Τέλος, γράφει σύμβολο ” ; ”.
- Η τελευταία περίπτωση είναι ο τελεστής τερματισμού ” halt ”. Τότε γράφει στο αρχείο σύμβολο ” ; ” και το σύμβολο ” {} ”.

```

def c_stuff(code):
    global tmp_variables_list

    if (len(tmp_variables_list) != 0):
        code.write("int ")
    #Temp_X variables
    for i in range(len(tmp_variables_list)):
        code.write(tmp_variables_list[i])
        if (len(tmp_variables_list) == i + 1):
            code.write("\n\n\t")
        else:
            code.write(",")
    for j in range(len(quad_list)):
        if (quad_list[j][1] == 'begin_block'):
            code.write("L_" + str(j+1) + ": \n\t")
        elif (quad_list[j][1] == "="):
            code.write("L_" + str(j+1) + ": " + quad_list[j][4] + "=" + quad_list[j][2] + "; \n\t")
        elif (quad_list[j][1] == "+"):
            code.write("L_" + str(j+1) + ": " + quad_list[j][4] + "+" + quad_list[j][2] + "+" + quad_list[j][3] + "; \n\t")
        elif (quad_list[j][1] == "-"):
            code.write("L_" + str(j+1) + ": " + quad_list[j][4] + "-" + quad_list[j][2] + "-" + quad_list[j][3] + "; \n\t")
        elif (quad_list[j][1] == "*"):
            code.write("L_" + str(j+1) + ": " + quad_list[j][4] + "*" + quad_list[j][2] + "*" + quad_list[j][3] + "; \n\t")
        elif (quad_list[j][1] == "/"):
            code.write("L_" + str(j+1) + ": " + quad_list[j][4] + "/" + quad_list[j][2] + "/" + quad_list[j][3] + "; \n\t")
        elif (quad_list[j][1] == "JUMP"):
            code.write("L_" + str(j+1) + ": " + "goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == "<"):
            code.write("L_" + str(j+1) + ": " + "if (" + quad_list[j][2] + "<" + quad_list[j][3] + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == ">"):
            code.write("L_" + str(j+1) + ": " + "if (" + quad_list[j][2] + ">" + quad_list[j][3] + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == ">="):
            code.write("L_" + str(j+1) + ": " + "if (" + quad_list[j][2] + ">=" + quad_list[j][3] + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == "<="):
            code.write("L_" + str(j+1) + ": " + "if (" + quad_list[j][2] + "<=" + quad_list[j][3] + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == "<>"):
            code.write("L_" + str(j+1) + ": " + "if (" + str(quad_list[j][2]) + "!=" + str(quad_list[j][3]) + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
        elif (quad_list[j][1] == "="):
            code.write("L_" + str(j+1) + ": " + "if (" + quad_list[j][2] + "==" + quad_list[j][3] + ") goto L_" + str(quad_list[j][4]) + "; \n\t")
    #Printing the result of the expression
    elif (quad_list == "out"):
        code.write("L_" + str(j+1) + ": " + "printf(\"" + quad_list[j][2] + "%d\", " + quad_list[j][2] + "); \n\t")
    elif (quad_list[j][1] == "halt"):
        code.write("L_" + str(j+1) + ": {} \n\t")

```

- **declare():**

Όσων αφορά την εγγραφή κώδικα C η συνάρτηση declare() κάθε φορά που διαβάζει τη δεσμευμένη λέξη "declare" η οποία χρησιμοποιείται για τη δήλωση των μεταβλητών γράφει στο C αρχείο τη λέξη "int " διότι γνωρίζουμε ότι η Starlet υποστηρίζει μόνο ακέραιους. Αντίστοιχα όταν διαβάσει το σύμβολο ";" που βρίσκεται μετά τη δήλωση μίας ή περισσότερων μεταβλητών χωρισμένων με κόμμα γράφει στο C αρχείο ο σύμβολο ";".

```
# <declare> ::= (declare <varlist>;)* #
def declarations():
    global token

    while (token[0] == 'declare'):
        code.write("int ")
        token = lex()
        varlist()
        if (token[0] == ';'):
            code.write(";\n\t")
            token = lex()
            return
        else:
            print('ERROR: Expected ";" after after the variable "{0}".'.format(token[0]))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **my_files():**

Η συνάρτηση my_files() διαχειρίζεται τα αρχεία τα οποία θέλουμε να πάρουμε ως έξοδο μετά την μεταγλώττιση του προγράμματος. Δημιουργεί τα αρχεία test και c_code, τα οποία έχουν δικαίωμα εγγραφής, με την εντολή open της Python. Το αρχείο test θα περιέχει τον ενδιάμεσο κώδικα ενώ το c_code τον ισοδύναμο κώδικα σε C. Έπειτα γράφει στο αρχείο C την πρόταση "int main()" η οποία σηματοδοτεί την έναρξη του κώδικα C και καλείται ο Συντακτικός Αναλυτής με όρισμα το c_file. Στη συνέχεια καλείται η συνάρτηση test_file() με όρισμα το αρχείο test ώστε να γράψει σε αυτό όλες τις τετράδες. Αντίστοιχα καλείται η c_file() για την εγγραφή του ισοδύναμου κώδικα σε C και αφότου ολοκληρωθεί γράφεται το σύμβολο "\n" το οποίο σηματοδοτεί την αλλαγή γραμμής. Όταν ολοκληρωθούν όλες οι εγγραφές τα αρχεία κλείνουν με τη εντολή close της Python.

```
def my_files():
    test = open('test.int', 'w')
    c_code = open('c_code', 'w')

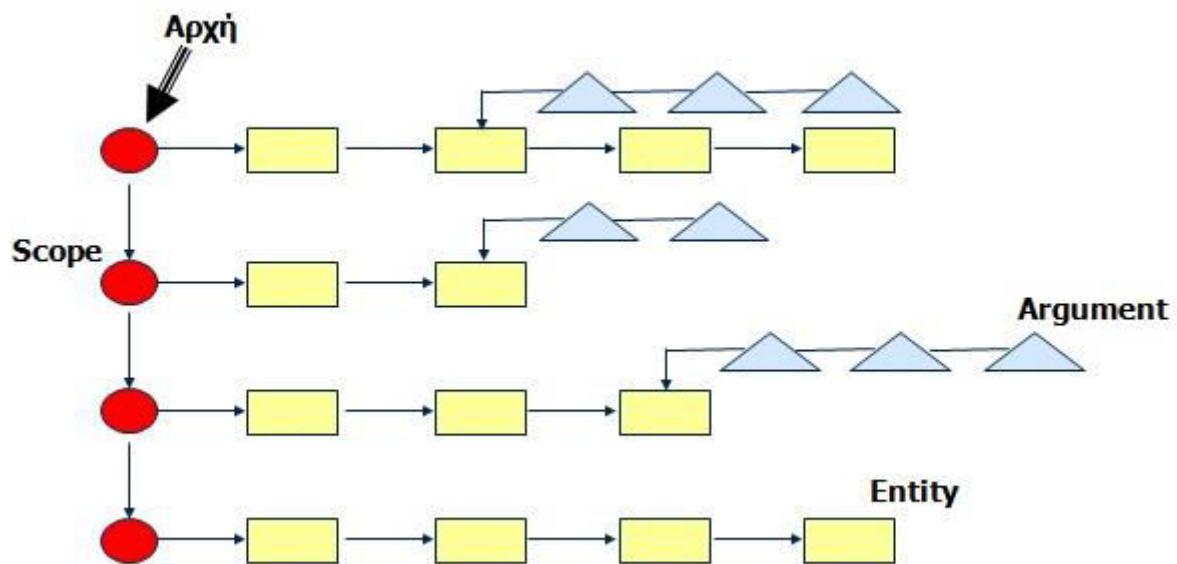
    c_code.write("int main () {\n\t")
    syntax_analyst(c_code)
    test_file(test)
    c_file(c_code)
    c_code.write("\n}")

    c_code.close()
    test.close()
    my_files()
```

Πίνακας Συμβόλων - Symbol Table:



Μορφή του πίνακα:



- **Τρίγωνο-Argument:**

Υλοποιείται με την κλάση Argument που περιέχει τη συνάρτηση `_init_()` η οποία παίρνει σαν όρισμα ένα αντικείμενο το οποίο έχει ένα όνομα, ένα τύπο(γνωρίζουμε ότι είναι πάντα ακέραιος) και μία παραμετροποίηση που δηλώνει αν πρόκειται για μεταβλητή που περνάει με αναφορά ή τιμή ή για επιστροφή τιμής.

```
# --- Triangle: --- #
class Argument():
    def __init__(self):
        self.name = ''                      #The name of the argument identifies it.
        self.type = 'Int'                    #Always integers.
        self.parameterization = ''          #RET | REF | CV
```

- **Τετράγωνο-Entity:**

Υλοποιείται με την κλάση Entity και περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο το οποίο έχει ένα όνομα και ένα τύπο. Ανάλογα με τον τύπο αυτό υλοποιούνται τέσσερις υποκλάσεις:

- **Variables():** Υλοποιείται για αντικείμενα των οποίων ο τύπος είναι μεταβλητή. Περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο του οποίου ο τύπος δηλώνεται ως interger και έχει ένα offset το οποίο δηλώνει την απόσταση από την αρχή της στοίβας και αρχικοποιείται στο 0.
- **Subprogram():** Υλοποιείται για αντικείμενα των οποίων ο τύπος είναι συνάρτηση. Περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο. Δηλώνεται ο τύπος και ετικέτα της πρώτης τετράδας του κώδικα της συνάρτησης και το μήκος εγγραφήματος τα οποία αρχικοποιούνται στο 0. Τέλος, δημιουργείται μια λίστα που περιέχει όλες τις παραμέτρους της συνάρτησης.
- **Parameter():** Υλοποιείται για αντικείμενα των οποίων ο τύπος είναι παράμετρος. Περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο. Δηλώνεται ο η παραμετροποίηση και το offset αρχικοποιείται στο 0.
- **Temp_Var():** Υλοποιείται για αντικείμενα των οποίων ο τύπος είναι προσωρινή μεταβλητή. Περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο του οποίου ο τύπος δηλώνεται ως interger και το offset που αρχικοποιείται στο 0.

```

# --- Rectangle: --- #
class Entity():
    def __init__(self):
        self.name = ''
        self.type = ''
        self.variable = self.Variable()
        self.subprogram = self.Subprogram()
        self.parameter = self.Parameter()
        self.tempVar = self.Temp_Var()

    class Variable:
        def __init__(self):
            self.type = 'Int'
            self.offset = 0          #Distance from the beginning of the stack.

    class Subprogram:
        def __init__(self):
            self.type = ''
            self.startingQuad = 0      #Calling nextquad().
            self.frameLength = 0
            self.args_list = arguments #List of arguments.

    class Parameter:
        def __init__(self):
            self.parameterization = ''
            self.offset = 0

    class Temp_Var:
        def __init__(self):
            self.type = 'Int'
            self.offset = 0

```

- **Κύκλος- Scope:**

Υλοποιείται με την κλάση Scope και περιέχει τη συνάρτηση `_init_()` που παίρνει όρισμα ένα αντικείμενο το οποίο έχει ένα όνομα. Κατασκευάζει μια λίστα από Entities και μια από Scopes και αρχικοποιεί το βάθος φωλιάσματος στο 0.

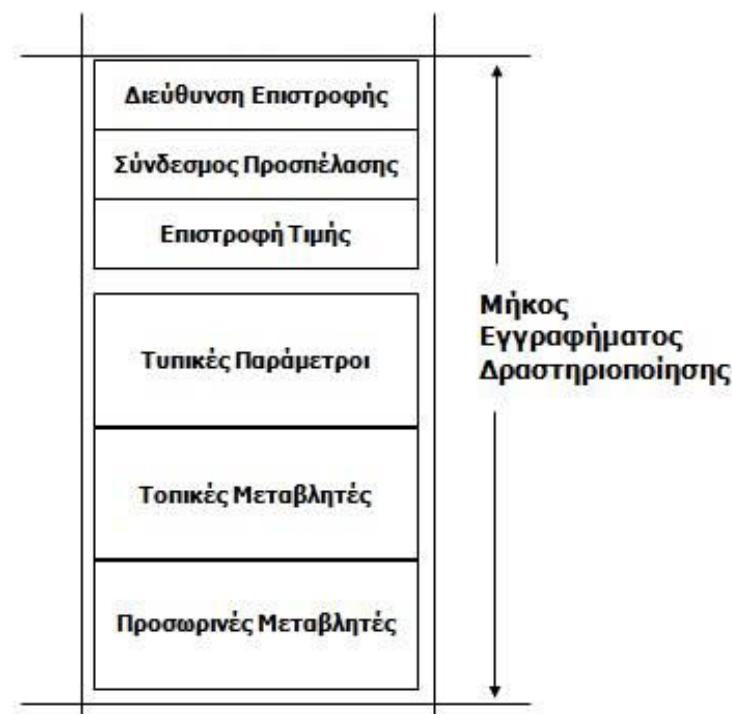
```

# --- Circle: --- #
class Scope():
    def __init__(self):
        self.name = ''
        self.entity_list = entities      #List of entities,
        self.nesting_depth = 0
        self.scope_enclosure = scopes   #Including ALL scopes

```

Εγγράφημα Δραστηριοποίησης:

- Δημιουργείται για κάθε συνάρτηση από αυτήν που την καλεί.
- Όταν αρχίζει η εκτέλεση της συνάρτησης ο δείκτης στοίβας μεταφέρεται στην αρχή του εγγραφήματος δραστηριοποίησης.
- Περιέχει πληροφορίες που χρησιμεύουν για την εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί.
- Όταν τερματίζεται η συνάρτηση ο χώρος που καταλαμβάνει το εγγράφημα δραστηριοποίησης επιστρέφεται στο σύστημα.



- ❖ Διεύθυνση επιστροφής: η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης.
- ❖ Σύνδεσμος Προσπέλασης: δείχνει στο εγγράφημα δραστηριοποίησης που πρέπει να αναζητηθούν μεταβλητές οι οποίες δεν είναι τοπικές αλλά η συνάρτηση έχει δικαίωμα να χρησιμοποιήσει.
- ❖ Επιστροφή τιμής: η διεύθυνση στην οποία θα γραφεί το αποτέλεσμα της συνάρτησης όταν αυτό υπολογιστεί.

- ❖ Χώρος αποθήκευσης παραμέτρων συνάρτησης:
 - αποθηκεύεται η τιμή, αν πρόκειται για πέρασμα με τιμή.
 - αποθηκεύεται η διεύθυνση, αν πρόκειται για πέρασμα με αναφορά.
- ❖ Χώρος αποθήκευσης τοπικών μεταβλητών.
- ❖ Χώρος αποθήκευσης προσωρινών μεταβλητών.

Ενέργειες στον Πίνακα Συμβόλων:

- **Προσθήκη νέου Argument:**

Πριν την κλήση της συνάρτησης new_argument() κατασκευάζεται μία κενή λίστα από arguments και δημιουργείται ένα βαθύ αντίγραφο αυτής. Αφότου κληθεί με όρισμα ένα argument, το τοποθετεί μέσα στη λίστα που δημιουργήθηκε νωρίτερα.

```
arguments = []
arguments = copy.deepcopy(arguments)

# Adding current object to the list: #
def new_argument(object):
    global arguments, entities

    arguments.append(object)          #Adding the object (obj) to my list.
    entities[-1].subprogram.args_list.append(object)
```

- **Προσθήκη νέου Entity:**

Πριν την κλήση της συνάρτησης new_entity () κατασκευάζεται μία κενή λίστα από entities και δημιουργείται ένα βαθύ αντίγραφο αυτής. Αφότου κληθεί με όρισμα ένα entity, το τοποθετεί μέσα στη λίστα που δημιουργήθηκε νωρίτερα.

```
entities = []
entities = copy.deepcopy(entities)

def new_entity(object):
    global arguments, entities, scopes

    scopes[-1].entity_list.append(object)
    entities.append(object)

    arguments = []
```

- **Προσθήκη νέου Scope:**

Πριν την κλήση της συνάρτησης new_scope () κατασκευάζεται μία κενή λίστα από scopes και δημιουργείται ένα βαθύ αντίγραφο αυτής. Αρχικοποιείται μια μεταβλητή η οποία αναφέρεται στο πρώτο scope και τοποθετείται στη λίστα με τα scopes. Ορίζεται μεταβλητή που δείχνει στις θέσεις της λίστας scopes και στη συνέχεια καλείται η συνάρτηση new_scope() με όρισμα το όνομα της νέας συνάρτησης που διαβάζεται. Το όνομα του πρώτου scope ορίζεται ως το όνομα της συνάρτησης. Έπειτα αν το scope δεν είναι το πρώτο, τότε το επόμενο scope βρίσκεται μέσα στην εμβέλεια του πρώτου και μπαίνει μέσα στη λίστα με τα scopes. Εν συνεχείᾳ η τιμή του πρώτου scope δείχνει στο επόμενο και η λίστα των entities αδειάζει με σκοπό να γεμίσει ξανά με νέα στοιχεία. Τέλος, ελέγχει αν υπάρχει το πρώτο scope στη λίστα με όλα τα scopes. Αν δεν υπάρχει ορίζουμε το βάθος φωλιάσματος 0, αλλιώς όσο είναι το τρέχον βάθος +1 .

```

scopes = []
scopes = copy.deepcopy(scopes)
first_scope = Scope()           #Initialization of the 1st scope!
scopes.append(first_scope)

first_point = 1 # ? ? ? ?

# Creation of a new scope:
def new_scope(name):
    global first_scope, entities, scopes, first_point

    first_scope.name = name
    next_scope = Scope()

    if (not first_point):
        next_scope.scope_enclosure.append(first_scope)      #next_scope is included inside the first_scope!

    first_scope = next_scope
    first_scope.entity_list = []
    entities = []                                         #Emptying the entities list in order to renew them

    # next_scope = next (downwards) nesting_depth + 1:
    if (first_scope.scope_enclosure == []):
        first_scope.nesting_depth = 0                      #We are at the beginning
    else:
        first_scope.nesting_depth = first_scope.scope_enclosure[-1].nesting_depth + 1   #One deeper
    first_point = 0      #We're back to square one.

```

- **Διαγραφή Scope:**

Η συνάρτηση Scope καλείται για την διαγραφή ενός scope. Ελέγχει αν η λίστα με τα scopes είναι άδεια. Αν δεν είναι τότε διαγράφεται το τελευταίο scope.

```

def delete_scope():
    global scopes

    if (scopes != []):
        del scopes[-1] #DELETE last scope

```

- **calculate_offset():**

Η συνάρτηση calculate_offset() χρησιμοποιείται για τον υπολογισμό της απόστασης από την αρχή της στοίβας σε bytes. Στην αρχή αρχικοποιείται ένας counter στο 0. Αν η λίστα με τα scopes των entities δεν είναι κενή, τότε διατρέχει τη λίστα και κάθε φορά που ο τύπος που διαβάζει είναι μεταβλητή ή συνάρτηση ή παράμετρο ή προσωρινή μεταβλητή, τότε αυξάνει κατά 1 μονάδα τον counter. Το offset έχει αρχική τιμή 12. Έπειτα η τιμή του counter πολλαπλασιάζεται επί 4 ώστε να βρούμε την τιμή του σε bytes και προστίθεται στο offset. Μετά την ολοκλήρωση της διαδικασίας η τιμή του offset επιστρέφεται.

```
# Calculating our distance in bytes: #
def calculate_offset():
    global scopes

    counter = 0
    if (scopes[-1].entity_list is not []):
        for i in (scopes[-1].entity_list):
            if (i.type == 'VAR' or i.type == 'TEMP' or i.type == 'PARAM' or i.type == 'SUBPR'):
                counter += 1
    #(int) = 4bytes --> START: 3 * 4 = 12
    offset = 12 + (counter * 4)

    return offset
```

- **calculate_startingQuad():**

Η συνάρτηση calculate_startingQuad() κατασκευάζει την πρώτη τετράδα κατά την κλήση μια συνάρτησης. Διατρέχει τη λίστα με τα scopes των entities και όταν ο τύπος που διαβάζει είναι συνάρτηση('SUBPR') και το όνομά είναι το όνομα της συνάρτησης και συμπληρώνει το πεδίο της ετικέτας της πρώτης τετράδας του κώδικα της συνάρτησης(βρίσκεται στην κλάση Subprogram της κλάσης Entity) καλώντας την nextquad().

```
# Calculating the starting quad of the function: #
def calculate_startingQuad():
    global scopes

    for i in scopes[-1].scope_enclosure[-2].entity_list:
        if (i.type == 'SUBPR' and i.name == scopes[-1].scope_enclosure[-1].name):
            i.subprogram.startingQuad = nextquad()
```

- **calculate_frameLength():**

Κατά την κλήση της συνάρτησης calculate_frameLength() υπολογίζεται το μήκος εγγραφήματος δραστηριοποίησης μιας συνάρτησης. Γίνεται προσπέλαση της λίστας με τα scopes των entities. Όταν ο τύπος που διαβάζει είναι συνάρτηση('SUBPR') και το όνομά είναι το όνομα της συνάρτησης, συμπληρώνεται το πεδίο του μήκους εγγραφήματος(βρίσκεται στην κλάση Subprogram της κλάσης Entity) καλώντας τη συνάρτηση calculate_offset().

```
# Calculating the framelen of the function: #
def calculate_frameLength():
    global scopes

    for i in scopes[-1].scope_enclosure[-2].entity_list:
        if (i.type == 'SUBPR' and i.name == scopes[-1].scope_enclosure[-1].name):
            i.subprogram.framelen = calculate_offset()
```

- **add_parameters():**

Με τη συνάρτηση add_parameters() δημιουργούνται entities των παραμέτρων των συναρτήσεων. Πριν ξεκινήσει η προσπέλαση της λίστας των arguments υπολογίζει το offset της παραμέτρου καλώντας τη συνάρτηση calculate_offset(). Για κάθε στοιχείο της λίστας arguments κατασκευάζεται ένα entity καλώντας την Entity, το όνομά του είναι το όνομα του argument που διαβάστηκε καθώς και η παραμετροποίηση του. Αφού πρόκειται για παράμετρο συνάρτησης ο τύπος του θα είναι παράμετρος('PARAM') και το offset θα είναι αυτό που υπολογίστηκε πριν την προσπέλαση της λίστας. Τέλος με αυτό το entity καλείται η new_entity() για τη προσθήκη αυτού στη λίστα με τα entities. Η διαδικασία εκτελείται επαναληπτικά εώς ότου αδειάσει η λίστα με τα arguments.

```
# Creating entities of various parameters of functions [func a(in a, inout b)] #
def add_parameters():
    global arguments

    offset = calculate_offset()

    for i in arguments:
        ent = Entity()
        ent.name = i.name
        ent.type = 'PARAM'
        ent.parameter.parameterization = i.parameterization
        ent.parameter.offset = offset
        new_entity(ent)
```

- **print_symbols_matrix():**

Διατρέχει τη λίστα με τα scopes. Για κάθε scope τυπώνει:

- Το όνομα του scope και το βάθος φωλιάσματος στη μορφή:
Scope: Name: Όνομα Nesting Depth: Βάθος φωλιάσματος
- Τη λέξη Entities και ξεκινά την προσπέλαση της λίστας με τα entities. Για κάθε entity ελέγχει τον τύπο:

◆ Αν είναι μεταβλητή τυπώνει τα περιεχόμενα της(όνομα, τύπο, τύπο μεταβλητής και την απόσταση από την αρχή της στοίβας) στη μορφή:

Entity: Name: Type: Variable_Type: Offset:

◆ Αν είναι προσωρινή μεταβλητή τυπώνει τα περιεχόμενα της(όνομα, τύπο, τύπο μεταβλητής και την απόσταση από την αρχή της στοίβας) στη μορφή:

Entity: Name: Type: Temp_Type: Offset:

◆ Αν είναι υποπρόγραμμα και ο τύπος του είναι συνάρτηση τυπώνει τα περιεχόμενα του(όνομα, τύπο, τύπο συνάρτησης, χαρακτηριστικό αριθμό πρώτης τετράδας) στη μορφή:

Entity: Name: Type: Function_Type: Starting_Quad:

Έπειτα τυπώνει τη λέξη Arguments και ξεκινά την προσπέλαση της λίστας με τα arguments. Για κάθε argument τυπώνει τα περιεχόμενα του(όνομα, τύπο και παραμετροποίηση) στη μορφή:

Argument: Name: Type: Parameter_Mode:

◆ Αν είναι παράμετρος τυπώνει τα περιεχόμενα της(όνομα, τύπο, παραμετροποίηση και την απόσταση από την αρχή της στοίβας) στη μορφή:

Entity: Name: Type: Mode: Offset:

```

# Printing the matrix: Scopes, Entities, Arguments #
def print_symbols_matrix():
    global scopes, entities, arguments

    print("-----")
    for s in scopes:
        print("Scope: " + s.name + " Nesting_Depth: " + str(s.nesting_depth))
        print("\tEntities:")
        for e in s.entity_list:
            if (e.type == 'VAR'):
                print("\t\tEntity: " + e.name + "\t Type: " + e.type + "\t Variable_Type: " + e.variable.type + "\t Offset: " + str(e.variable.offset))
            elif (e.type == 'TEMP'):
                print("\t\tEntity: " + e.name + "\t Type: " + e.type + "\t Temp_Type: " + e.tempVar.type + "\t Offset: " + str(e.tempVar.offset))
            elif (e.type == 'SUBPR'):
                if (e.subprogram.type == 'Function'):
                    print("\t\tEntity: " + e.name + "\t Type: " + e.type + "\t Function_Type: " + e.subprogram.type + "\t Starting_Quad: " + str(e.subprogram.startingQuad))
                    print("\t\tArguments:")
                    for a in e.subprogram.args_list:
                        print("\t\t\tArgument: " + a.name + "\t Type: " + a.type + "\t Parameter_Mode: " + a.parameterization)
                elif (e.type == 'PARAM'):
                    print("\t\tEntity: " + e.name + "\t Type: " + e.type + "\t Mode: " + e.parameter.parameterization + "\t Offset: " + str(e.parameter.offset))
            print("-----")

```

- **<block> ::= <declare> <subprograms> <statements>**:

Δημιουργεί ένα νέο scope καλώντας την new_scope() με όρισμα το όνομα του block. Αν βρίσκεται μέσα σε block κώδικα τότε δημιουργούνται νέα entities με την κλήση της συνάρτησης add_parameters(). Αν το token που διαβάζεται είναι η δεσμεύμενη λέξη "declare" καλείται η συνάρτηση declarations() και ακολουθούν οι subprogram() και η genquad(). Ελέγχει αν βρίσκεται μέσα σε block κώδικα. Αν ναι τότε υπολογίζεται ετικέτα της πρώτης τετράδας με την κλήση της calculate_startingQuad(). Αφότου κληθεί η statements() ελέγχεται αν υπάρχει flag που υποδεικνεύει το exit εκτός do while. Αν ισχύει τότε εμφανίζει μήνυμα λάθους και τερματίζει. Το flag που σηματοδοτεί τη λήξη του block ή τον τερματισμό του προγράμματος παίρνει την τιμή 1. Μετά την κλήση του λεκτικού αναλυτή αν πρόκειται για τερματισμό του προγράμματος καλείται η genquad(), αλλιώς υπολογίζει το μήκος του εγγραφήματος και ακολούθως καλείται η genquad(). Τέλος τυπώνεται ο πίνακας συμβόλων και διαγράφεται το τελευταίο scope.

```

# <block> ::= <declare> <subprograms> <statements> #
def block(block_name, mainProgBlockFlag):
    global token

    #token = lex()
    new_scope(block_name)

    if (mainProgBlockFlag != 1):      #Creating entities
        add_parameters()

    if (token[0] == 'declare'):
        declarations()
    subprograms()
    genquad('begin_block', block_name, '_', '_')

    if (mainProgBlockFlag != 1):      #calculating the starting quad of the next quad
        calculate_startingquad()
    statements()
    if (dowhileFlag != 1 and exitFlag == 1):
        print('ERROR: Keyword "exit" can only be applied inside "do-while" loops.')
        print('Line -> {0}:{1}'.format(line, in_line_position))
        exit(-1)

    closedBlockFlag = 1
    token = lex()
    if (mainProgBlockFlag == 1):
        genquad('halt', '_', '_', '_')
    else:
        calculate_frameLength()      #After closing the block, we are calculating the frameLength
    genquad('end_block', block_name, '_', '_')

    print("Print Symbols-Matrix: ")
    print_symbols_matrix()
    delete_scope()
    print("Last scope is deleted.")

    return

```

- **<varlist> ::= ε | id (, id)* :**

Για το πρώτο id δημιουργείται ένα entity με την κλήση Entity(). Ο τύπος του είναι μεταβλητή('VAR') και το όνομά του είναι το token[0] που έχει διαβαστεί. Ακολούθως το offset της υπολογίζεται με την κλήση calculate_offset() και καλείται η new_entity με όρισμα το νέο entity ώστε να μπει στη λίστα με τα entities. Αν υπάρχουν πολλά id χωρισμένα με κόμμα η παραπάνω διαδικασία εκτελείται για κάθε ένα από αυτά.

```

# <varlist> ::= ε | id ( , id )* #
def varlist():
    global token

    if (token[1] == 1):      #identifier (a.k.a. tokenID)
        code.write(token[0])

        #Creating an entity:
        ent = Entity()
        ent.type = 'VAR'
        ent.name = token[0]
        ent.variable.offset = calculate_offset()
        new_entity(ent)

        token = lex()
        while (token[0] == ','):
            code.write(token[0])
            token = lex()
            if (token[1] == 1):
                code.write(token[0])

                #Creating an entity (again):
                ent = Entity()
                ent.type = 'VAR'
                ent.name = token[0]
                ent.variable.offset = calculate_offset()
                new_entity(ent)

                token = lex()
            else:
                print('ERROR: Variable was expected before "{0}".'.format(token[0]))
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)

    return

```

- <subprogram ::= function id <funcbody> endfunction:

Δημιουργείται ένα entity με την κλήση Entity(). Ο τύπος του είναι υποπρόγραμμα('SUBPR') και το όνομά του είναι το token[0] που διαβάστηκε. Στη συνέχεια ορίζεται ο τύπος του υποπρογράμματος ως συνάρτηση και τέλος καλείται η new_entity() με όρισμα το entity που δημιουργήθηκε, η οποία το τοποθετεί στη λίστα με τα entities.

```
# <subprogram ::= function id <funcbody> endfunction #
def subprogram():
    global token

    if (token[0] == 'function'):
        token = lex()
        if (token[1] == 1):      # tokenID
            name = token[0]    # Keep the name of the procedure

            #Creating an entity:
            ent = Entity()
            ent.type = 'SUBPR'
            ent.name = token[0]
            ent.subprogram.type = 'Function'
            new_entity(ent)

            token = lex()
            funcbody(name, 1)  # Because it is a function
            if (token[0] == 'endfunction'):
                token = lex()
            else:
                print('ERROR: Keyword "endfunction" was expected or you missed a semicolon ";"')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
            else:
                print('ERROR: ID number expected.')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
            else:
                print('ERROR: Keyword "function" was expected.')
                print('Line -> {0}:{1}'.format(line, in_line_position))
                exit(-1)
    return
```

- <funcbody> ::= <formalpars> <block>:

Μετά τις κλήσεις των formalpars() και block() πραγματοποιείται έλεγχος για την ύπαρξη της δεσμευμένης λέξης "return" μέσα στις συναρτήσεις. Αν δεν υπάρχει, δηλαδή δε συναντήσει το flag που υποδεικνύει την επιστροφή τιμής εμφανίζει το αντίστοιχο μήνυμα λάθους και τερματίζει, αλλιώς μηδενίζει το flag.

```
# <funcbody> ::= <formalpars> <block> #
def funcbody(block_name, func):
    global token, minOneReturnFlag

    formalpars()           #same syntax principles as <block>
    block(block_name, -1)   #not a main program block

    #Every function includes at least one return
    if (func == 1):
        if (minOneReturnFlag != 1):
            print('ERROR: Function {0} is missing the keyword "return".'.format(block_name))
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
        else:
            minOneReturnFlag = 0

    return
```

- **<formalparitem> ::= in id | inout id | inandout id:**

Όταν διαβαστεί η δεσμευμένη λέξη "in" τότε δημιουργείται ένα νέο argument με την κλήση της Argument(). Το όνομά του είναι το token[0] και το πέρασμά γίνεται με τιμή. Μπαίνει στη λίστα με τα arguments με την κλήση της new_argument(). Η ίδια διαδικασία ακολουθείται και όταν διαβάσει τις δεσμευμένες λέξεις "inout" και "inandout" με τη μόνη διαφορά στην παραμετροποίηση η οποία είναι επιστροφή τιμής και πέρασμα με αναφορά αντίστοιχα.

```
# <formalparitem> ::= in id | inout id | inandout id #
def formalparitem():
    global token

    if (token[0] == 'in'):
        token = lex()
        if (token[1] == 1):      #identifier ID
            #creation of an argument:
            a = Argument()
            a.name = token[0]
            a.parameterization = 'CV'
            new_argument(a)

            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "in".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inout'):
        token = lex()
        if (token[1] == 1):
            #creation of an argument (again):
            a = Argument()
            a.name = token[0]
            a.parameterization = 'PAR'
            new_argument(a)

            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "inout".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    elif (token[0] == 'inandout'):
        token = lex()
        if (token[1] == 1):
            #Creation of an argument (again):
            a = Argument()
            a.name = token[0]
            a.parameterization = 'REF'
            new_argument(a)

            token = lex()
            return
        else:
            print('ERROR: Variable name expected after the keyword "inandout".')
            print('Line -> {0}:{1}'.format(line, in_line_position))
            exit(-1)
    return
```

- **<return-stat> ::= return <expression>:**

Στη return_stat() ορίζεται ένας σημαφόρος(flag) ο οποίος έχει την τιμή 1 όταν υπάρχει η δεσμευμένη λέξη "return" μέσα σε μία συνάρτηση.

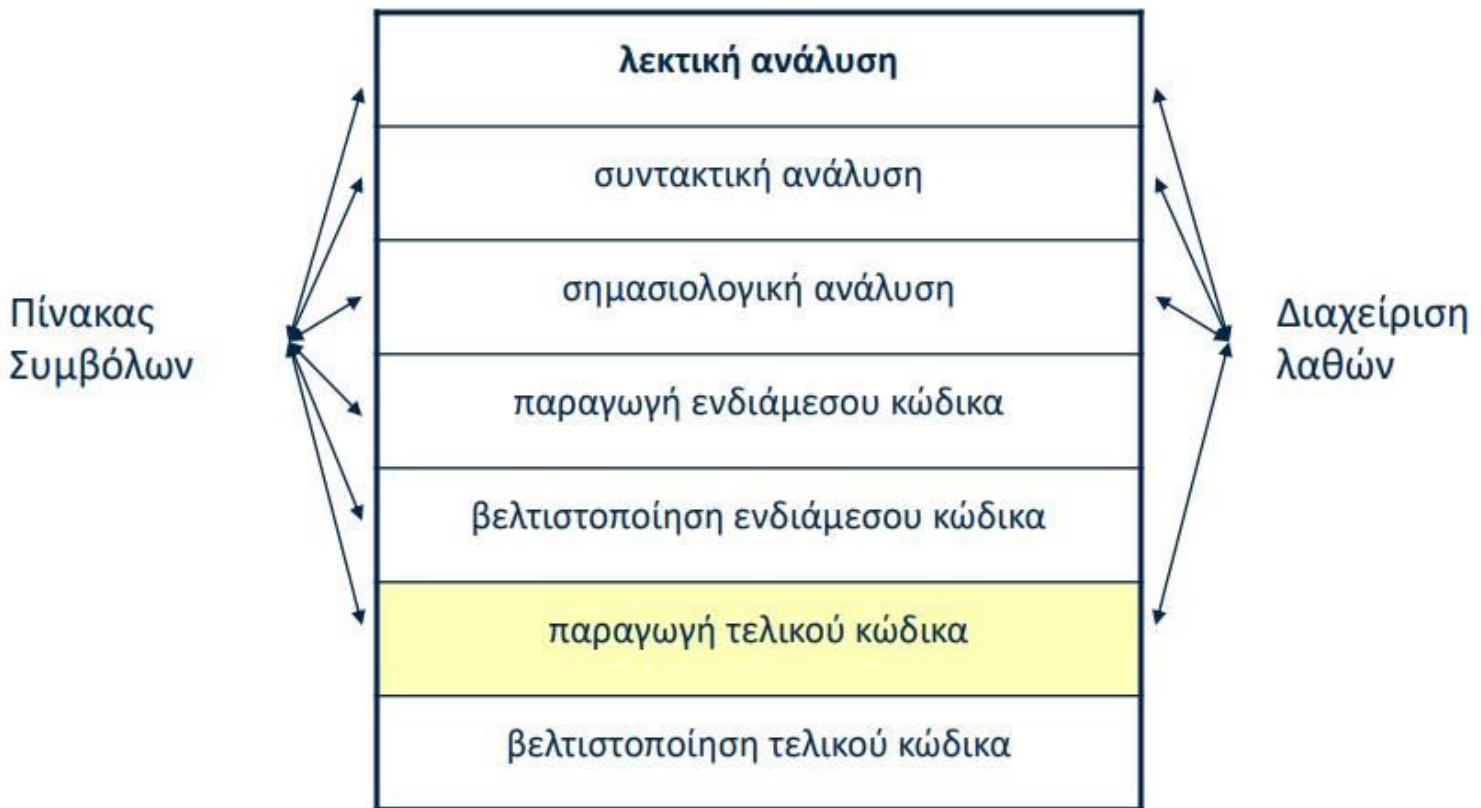
```
# <return-stat> ::= return <expression> #
# INTR_CODE: return_stat -> return E {P1} #
def return_stat():
    global token, minOneReturnFlag

    minOneReturnFlag = 1

    #token = lex()
    if (token[0] == 'return'):
        token = lex()
        #[P1]:
        E_place = expression()
        genquad('retv', E_place, '_', '_')
    return
```

Παραγωγή Τελικού Κώδικα - Final Code Production:

- Οι Φάσεις της Μεταγλώττισης:



- Παραγωγή Τελικού Κώδικα:



- Από κάθε μία εντολή ενδιάμεσου κώδικα παράγουμε τις αντίστοιχες εντολές του τελικού κώδικα.
- Κύριες ενέργειες στη φάση αυτή:
 - οι μεταβλητές απεικονίζονται στην μνήμη (στοίβα).
 - το πέρασμα παραμέτρων και η κλήση συναρτήσεων.
- Θα δημιουργήσουμε κώδικα για τον επεξεργαστή MIPS.

Η αρχιτεκτονική MIPS:

- Καταχωρητές που θα μας φανούν χρήσιμοι:
 - ◆ καταχωρητές προσωρινών τιμών: \$t0...\$t7.
 - ◆ καταχωρητές οι τιμές των οποίων διατηρούνται ανάμεσα σε κλήσεις συναρτήσεων: \$s0...\$s7.
 - καταχωρητές ορισμάτων: \$a0...\$a3.
 - καταχωρητές τιμών: \$v0,\$v1.
 - stack pointer \$sp.
 - frame pointer \$fp.
 - return address \$ra.
- Εντολές που θα μας φανούν χρήσιμες για αριθμητικές πράξεις:
 - ◆ add \$t0,\$t1,\$t2 t0=t1+t2
 - ◆ sub \$t0,\$t1,\$t2 t0=t1-t2
 - ◆ mul \$t0,\$t1,\$t2 t0=t1*t2
 - ◆ div \$t0,\$t1,\$t2 t0=t1/t2
- Εντολές που θα μας φανούν χρήσιμες για μετακίνηση δεδομένων:
 - ◆ move \$t0,\$t1 t0=t1 για μεταφορά ανάμεσα σε καταχωρητές.
 - ◆ li \$t0, value t0=value για σταθερά σε καταχωρητή.
 - ◆ lw \$t1,mem t1=[mem] για περιεχόμενο μνήμης σε καταχωρητή.
 - ◆ sw \$t1,mem [mem]=t1 για περιεχόμενο καταχωρητή σε μνήμη.
 - ◆ lw \$t1,(\$t0) t1=[t0] για έμμεση αναφορά με καταχωρητή.

- ◆ sw \$t1,-4(\$sp) t1=[\$sp-4] για έμμεση αναφορά με βάση τον \$sp.
- Εντολές που θα μας φανούν χρήσιμες για άλματα:
 - ◆ b label branch to label
 - ◆ beq \$t1,\$t2,label jump to label if \$t1==\$t2.
 - ◆ blt \$t1,\$t2,label jump to label if \$t1<\$t2.
 - ◆ bgt \$t1,\$t2,label jump to label if \$t1>\$t2.
 - ◆ ble \$t1,\$t2,label jump to label if \$t1<=\$t2.
 - ◆ bge \$t1,\$t2,label jump to label if \$t1>=\$t2.
 - ◆ bne \$t1,\$t2,label jump to label if \$t1<>\$t2.
- Εντολές που θα μας φανούν χρήσιμες στην κλήση συναρτήσεων:
 - ◆ j label jump to label.
 - ◆ jal label κλήση συνάρτησης.
 - ◆ jr \$ra άλμα στη διεύθυνση που έχει ο καταχωρητής στο παράδειγμα είναι ο \$ra που έχει την διεύθυνση επιστροφής συνάρτησης.

Βοηθητικές Συναρτήσεις:

- **Η συνάρτηση `gnvicode()`:**
 - ◆ Μεταφέρει στον \$t0 την διεύθυνση μιας μη τοπικής μεταβλητής.
 - ◆ Από τον πίνακα συμβόλων βρίσκει πόσα επίπεδα επάνω βρίσκεται η μη τοπική μεταβλητή και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει:

lw \$t0,-4(\$sp)	στοίβα του γονέα
όσες φορές χρειαστεί:	
lw \$t0,-4(\$t0)	στοίβα του προγόνου που έχει
τη μεταβλητή	
add \$t0,\$t0,-offset	διεύθυνση της μη τοπικής μεταβλητής

- **Η συνάρτηση loadvr():**

- Πραγματοποιεί μεταφορά δεδομένων στον καταχωρητή r.
- Η μεταφορά μπορεί να γίνει από τη μνήμη (στοίβα).
- Ή να εκχωρηθεί στο r μία σταθερά.
- Η σύνταξη της είναι `loadvr(v,r)`.
- Διακρίνουμε τις εξής περιπτώσεις:
 - ▶ Αν η v είναι σταθερά:
`li $tr,v`
 - ▶ Αν η v είναι καθολική μεταβλητή, δηλαδή ανήκει στο κυρίως πρόγραμμα:
`lw $tr,-offset($s0)`
 - ▶ Αν η v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή:
`lw $tr,-offset($sp)`
 - ▶ Αν η v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον:
`lw $t0,-offset($sp)`
`lw $tr,($t0)`
 - ▶ Αν η v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον:
`gnlvcode()`
`lw $tr,($t0)`
 - ▶ Αν η v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:
`gnlvcode()`
`lw $t0,($t0)`
`lw $tr,($t0)`

- **Η συνάρτηση storerv():**

- Πραγματοποιείται μεταφορά δεδομένων από τον καταχωρητή r στη μνήμη (μεταβλητή v).
- Η σύνταξη της είναι storerv(r,v).
- Διακρίνουμε τις εξής περιπτώσεις:
 - ▶ Αν η v είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα:
`sw $tr,-offset($s0)`
 - ▶ Αν η v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή:
`sw $tr,-offset($sp)`
 - ▶ Αν η είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον:
`lw $t0,-offset($sp)`
`sw $tr,($t0)`
 - ▶ Αν η v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον:
`gnlvcode(v)`
`sw $tr,($t0)`
 - ▶ Αν η είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:
`gnlvcode(v)`
`lw $t0,($t0)`
`sw $tr,($t0)`

Για την διεξαγωγή της τελευταίας φάσης του project χρησιμοποιήσαμε μόνο την λογική των βοηθητικών συναρτήσεων χωρίς να τις υλοποιήσουμε.

- **Εντολές Αλμάτων:**

- ▶ jump, “_”, “_”, label
- ▶ j label
- ▶ relop(?),x,y,z
- ▶ loadvr(x,1)
- ▶ loadvr(y,2)
- ▶ branch(?) , \$t1,\$t2,z όπου branch(?) : beq,bne,bgt,blt,bge,ble

- **Εκχώρηση:**

- ▶ :=, x, “_”, z
loadvr(x,1)
storerv(1,z)

- **Εντολές Αριθμητικών Πράξεων:**

- ▶ op x,y,z
loadvr(x,1)
loadvr(y,2)
op \$t1,\$t2 op: add,sub,mul,div
storerv(1,z)

- **Εντολές Εισόδου-Εξόδου:**

- ▶ out “_”, “_”, x
li \$v0,1
li \$a0, x
syscall
- ▶ in “_”, “_”, x
li \$v0,5
syscall

Το αποτέλεσμα γράφεται στον \$v0.

- **Επιστροφή Τιμής Συνάρτησης:**

- `retv _, _, X`
`loadvr(x,1)`
`lw $t0,-8($sp)`
`sw $t1,($t0)`

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

- Εναλλακτικά μπορούμε να γράψουμε το αποτέλεσμα στον \$v0, και μετά πρέπει να φροντίσουμε να το πάρουμε από εκεί:
`loadvr(x,1)`
`move $v0,$t1`

- **Παράμετροι Συνάρτησης:**

- Πριν από την πρώτη παράμετρο, τοποθετούμε τον \$fp να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί:
`add $fp,$sp,framelength`
- `par,x,CV, _`
`loadvr(x,0)`
`sw $t0, -(12+4i)($fp)`
όπου i ο αύξων αριθμός της παραμέτρου
- `par,x,REF, _`
 - ▶ Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:
`add $t0,$sp,-offset`
`sw $t0,-(12+4i)($fp)`
 - ▶ Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά:
`lw $t0,-offset($sp)`
`sw $t0,-(12+4i)($fp)`
 - ▶ Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:

```
gnlvcode(x)
sw $t0,-(12+4i)($fp)
```

- ▶ Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά:

```
gnlvcode(x)
lw $t0,($t0)
sw $t0,-(12+4i)($fp)
```

- **par,x,RET, _**

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

```
add $t0,$sp,-offset
sw $t0,-8($fp)
```

- **Κλήση Συνάρτησης:**

- **call, _, _, f**

Αρχικά γεμίζουμε το 2ο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης με την διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα της, ώστε η κληθείσα να γνωρίζει που να κοιτάξει αν χρειαστεί να προσπελάσει μία μεταβλητή την οποία έχει δικαίωμα να προσπελάσει, αλλά δεν της ανήκει.

- ▶ Αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο γονέα:

```
lw $t0,-4($sp)
sw $t0,-4($fp)
```
- ▶ Αν καλούσα και κληθείσα έχουν διαφορετικό βάθος φωλιάσματος, τότε η καλούσα είναι ο γονέας της κληθείσας:

```
sw $sp,-4($fp)
```
- Στη συνέχεια μεταφέρουμε τον δείκτη στοίβας στην κληθείσα:

```
add $sp,$sp,framelength
```
- Καλούμε τη συνάρτηση:

```
jal f
```
- Και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα:

```
add $sp,$sp,-framelength
```

- μέσα στην κληθείσα
 - στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της την οποία έχει τοποθετήσει στον \$ra η jal:

sw \$ra,(\$sp)
 - στην τέλος κάθε συνάρτησης κάνουμε το αντίστροφο, παίρνουμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της συνάρτησης και την βάζουμε πάλι στον \$ra. Μέσω του \$ra επιστρέφουμε στην καλούσα:

lw \$ra,(\$sp)

jr \$ra
- **Αρχή Προγράμματος και Κυρίως Πρόγραμμα:**
 - Το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται, οπότε στην αρχή του προγράμματος χρειάζεται ένα άλμα που να οδηγεί στην πρώτη ετικέτα του κυρίως προγράμματος:

j Lmain
 - Στη συνέχεια πρέπει να κατεβάσουμε τον \$sp κατά framelength της main:

add \$sp,\$sp,framelength
 - Και να σημειώσουμε στον \$s0 το εγγράφημα δραστηριοποίησης της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές:

move \$s0,\$sp

- ▶ **Το σύμβολο: "+":**
+ \$t1 \$t1 \$t2
- ▶ **Το σύμβολο: "-":**
- \$t1 \$t1 \$t2
- ▶ **Το σύμβολο: "**":**
* \$t1 \$t1 \$t2
- ▶ **Το σύμβολο: "/":**
/ \$t1 \$t1 \$t2

Οι καταχωρητές \$t1, \$t1 αντιστοιχούν στις μεταβλητές που προσθέτονται/αφαιρούνται/διαιρούνται/πολλαπλασιάζονται. Το αποτέλεσμα της πράξης αποθηκεύεται στον καταχωρητή \$t2.

- **Τελεστής Εξόδου:**

Τότε γράφει στο αρχείο την μεταφορά της τιμής 1 στον καταχωρητή τιμών \$v0. Έπειτα την μεταφορά του τελούμενου 3 στον καταχωρητή ορισμάτων και τέλος τη λέξη "syscall".

```
def final_touches(assembly_file):
    global quad_list
    # print ("LENGTH_OF_QUAD", str(len(quad_list)))
    for i in range(len(quad_list)):
        if (quad_list[i][1] == 'jump'):
            assembly_file.write(' j' + ' ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '='):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('beq, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '>'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('bgt, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '<'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('blt, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '<>'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('bne, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '>='):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('bge, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '<='):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('ble, $t1, $t2, ' + str(quad_list[i][4]) + '\n')
        if (quad_list[i][1] == '=='):
            # loadvr(quad_list[i][2], 1)
            # storerv(1, quad_list[i][4])
            # continue...
            pass
```

```

        if (quad_list[i][1] == '+'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('+, $t1, $t1, $t2' + '\n')
            # storerv(1, quad_list[i][4])
        if (quad_list[i][1] == '-'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('-', $t1, $t1, $t2' + '\n')
            # storerv(1, quad_list[i][4])
        if (quad_list[i][1] == '*'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('*', $t1, $t1, $t2' + '\n')
            # storerv(1, quad_list[i][4])
        if (quad_list[i][1] == '/'):
            # loadvr(quad_list[i][2], 1)
            # loadvr(quad_list[i][3], 2)
            assembly_file.write('/', $t1, $t1, $t2' + '\n')
            # storerv(1, quad_list[i][4])
        if (quad_list[i][1] == 'out'):
            assembly_file.write('li $v0, 1' + '\n')
            assembly_file.write('li $a0, ' + quad_list[i][4] + '\n')
            assembly_file.write('syscall' + '\n')

```

- **my_files():**

Στη συνάρτηση my_files() προστέθηκε ένα ακόμα αρχείο assembly_file που θα περιέχει τυπωμένο τον τελικό κώδικα. Το αρχείο δημιουργείται με την εντολή open της Python και έχει δικαίωμα εγγραφής και ανάγνωσης. Στη συνέχεια καλείται η final_touches() με όρισμα το αρχείο αυτό ώστε να γράψει εκεί τον τελικό κώδικα. Τέλος το αρχείο κλείνει με την εντολή close.

```

def my_files():
    global quad_list

    # Opening files for writing and reading (w+ rights)
    int_file = open('int_file.int', 'w+')
    c_code = open('c_code.c', 'w+')
    assembly_file = open('assembly_file.txt', 'w+')

    # Final Code text file:
    # print("HELLO WORLD!", len(quad_list))

    # IT CAN WRITE IN HERE:
    # assembly_file.write('Hello')

    c_code.write("int main () {\n\t")

    syntax_analyst(c_code)
    int_stuff(int_file)
    c_stuff(c_code)
    final_touches(assembly_file)
    c_code.write("\n}")

    # Close everything:
    c_code.close()
    int_file.close()
    assembly_file.close()
my_files()
    #print("HELLO WORLD!", len(quad_list))

```

- **print_quad_list():**

Διατρέχει τη λίστα με τις τετράδες και τυπώνει όλα τα στοιχεία των τετράδων αυτών χωρισμένα με κενά.

```
def print_quad_list():
    for i in range(len(quad_list)):
        print(str(quad_list[i][0]) + " " + quad_list[i][1] + " " + quad_list[i][2] + " " + quad_list[i][3] + " " + quad_list[i][4])
```