Αναφορά εργασίας: “ Κατασκευή μεταφραστή για την εκπαιδευτική γλώσσα προγραμματισμού Cimple”

Μάθημα: “ΜΥΥ-802 Μεταφραστές”

Διδάσκων: Γ. Μανής

Ακαδημαϊκό έτος: 2020-2021

Πανεπιστήμιο Ιωαννίνων

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Μπουλώτης Παναγιώτης ΑΜ:4271

Κεφάλαια

0) Εισαγωγή

Στην αναφορά αυτή θα γίνει η ανάλυση του μεταφραστή που δημιουργήσαμε για την εκπαιδευτική προγραμματιστική γλώσσα Cimple, εφόσον επεξηγήσουμε συνοπτικά την λειτουργία της.

Στόχος της εργασίας είναι να ελέγξουμε πώς ένα πρόγραμμα γλώσσας Cimple (αρχεία με κατάληξη .ci) είναι συντακτικά ορθό και εκτελέσιμο με την μορφή αρχείου assembly για τον επεξεργαστή MIPS και .c στην περίπτωση που δεν υπάρχουν συναρτήσεις ή/και διαδικασίες στο αρχείο .ci. Στην περίπτωση λανθασμένης σύνταξης οφείλουμε να εκτυπώσουμε κατατοπιστικά μηνύματα λάθους στο τερματικό, ώστε να είναι δυνατό να γίνουν οι κατάλληλες μετατροπές από τον προγραμματιστή.

Ο κώδικας της, δηλαδή το αρχείο final_3090_4271, αναπτύχθηκε στη γλώσσα Python με βάση τις διαφάνειες του διδάσκοντα, μέρος των οποίων θα χρησιμοποιηθεί παρακάτω.

Για την καλύτερη κατανόηση του project, οι φάσεις της διαδικασίας μεταγλώττισης έχουν χωριστεί στα αντίστοιχα κεφάλαια, στα οποία θα επεξηγηθεί η θεωρία τους. Στη συνέχεια γίνεται η ανάλυση του κώδικα της φάσης εκείνης, και τέλος παράδειγμα εκτέλεσης του προγράμματος.

Για την εκτέλεση του προγράμματος πρέπει να χρησιμοποιηθεί η παρακάτω μορφή εκτέλεσης στο τερματικό:

- Εφόσον είμαστε στο κατάλογο που βρίσκεται το αρχείο .ci και το αρχείο του project (final_3090_4271.py)
- Αν το λογισμικό από το οποίο εκτελούμε είναι Windows πρέπει να εκτελέσουμε: python final_3090_4271.py file.ci
  όπου file.ci το αρχείο που θέλουμε να μεταγλωττίσουμε.
- Αν το λογισμικό μας είναι Linux η μόνη διαφορά εκτέλεσης είναι πως πρέπει να χρησιμοποιήσουμε την εντολή python3 αντί για την εντολή python.
- Αν το file.ci είναι συντακτικά σωστό, θα εμφανιστεί το μήνυμα
  “Syntax Analysis complete” και θα δημιουργηθούν αρχεία .txt, .int, .asm και .c για τους λόγους που θα εξηγήσουμε παρακάτω.

Το αρχείο final_3090_4271 έχει προγραμματιστεί με τέτοιο τρόπο, ώστε στη περίπτωση που δεν έχει δοθεί το file.ci ή έχουν δοθεί παραπάνω αρχεία ως ορίσματα, να εμφανίσει τα κατάλληλα μηνύματα σφάλματος.

1) Σύντομη περιγραφής γλώσσας Cimple

1.α) Περιγραφή

(Πηγή για τα παρακάτω είναι η αντίστοιχη διαφάνεια του διδάσκοντα για την γλώσσα Cimple)

Η γλώσσα την οποία θέλουμε να μεταγλωττίσουμε στην παρούσα εργασία είναι η Cimple, η οποία είναι ικανή να εκτελέσει απλά περιορισμένα προγράμματα, κυρίως για την επίλυση απλών μαθηματικών προβλημάτων.

Το αλφάβητο που αναγνωρίζει:

- Γράμματα λατινικής αλφαβήτου (a..z, A..Z) και **ακέραια** ψηφία (0..9)
- Σύμβολα για τις αντίστοιχες πράξεις/συγκρίσεις: (‘+’, ‘-’, ‘*’, ‘/’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘<>’)
- Ειδικό σύμβολο για την ανάθεση τιμής ή μεταβλητής σε μεταβλητή ‘:=’
- Ειδικό σύμβολο για τον τερματισμό του προγράμματος ‘.’ (Προσοχή: Δεν υπάρχουν πραγματικοί αριθμοί στη γλώσσα αυτή, μόνο ακέραιοι)
- Ειδικό σύμβολο για τα σχόλια ‘#’
- Υπόλοιπα σύμβολα για λειτουργίες αντίστοιχες σε γλώσσες όπως C (‘;’, ‘:’, ‘,’ , ‘[’, ‘]’, ‘{’, ‘}’, ‘(’, ‘)’) (Προσοχή: οι κλειστές παρενθέσεις ανταποκρίνονται σε λογικές παραστάσεις, ενώ οι ανοικτές σε αριθμητικές)

Παρακάτω είναι οι δεσμευμένες λέξεις της γλώσσας, η ανάλυση των οποίων θα γίνει κατανοητή στα παρακάτω κεφάλαια:

```
program     declare
if          else        while
switchcase  forcase     incase    case     default
not         and         or
function    procedure   call      return   in       inout
input       print
```

(πηγή: διαφάνεια μαθήματος)

Κάποιες βασικές σημειώσεις για την γλώσσα αυτή είναι οι εξής:

- Οι ακέραιες σταθερές πρέπει να ανήκουν στο διάστημα: $[-(2^{32}-1), 2^{32}-1]$
- Οι λέξεις που έχουν παραπάνω από **30** χαρακτήρες, δεν είναι έγκυρες
- Τα σχόλια πρέπει να βρίσκονται μέσα σε 2 σύμβολα #
- Κάθε πρόγραμμα πρέπει να αρχίζει με την λέξη program και ένα αντίστοιχο αναγνωριστικό (id) , το οποίο θα καθορίσει το όνομα των αρχείων που θα δημιουργηθούν.
- Αντίστοιχα οι εξισώσεις και οι διαδικασίες ορίζονται πριν το “main” κομμάτι του κώδικα με το αναγνωριστικό τους να δίνεται μετά την δεσμευμένη λέξη function/procedure.
- Αν πριν το αναγνωριστικό κάποιας παραμέτρου έχει δοθεί το “**in**”, αυτό σημαίνει πως η μετάδοση της γίνεται **με τιμή**, ενώ στην περίπτωση της “**inout**”, **με αναφορά.**

1.β) Παράδειγμα αρχείου.ci

Παρακάτω είναι ένα παράδειγμα αρχείου .ci

```
program absolute
   declare x;
   declare tempx;
   declare y;
   function abs(in x)
   {
      # tempx is a temporary value of x #
      tempx := 0;

      if(x<0){
         tempx := - x;
      }else{
         tempx := x;
      };
      return (tempx);
   }

   # main #
   {
      input(x);
      y := abs( in x);
      print(y);
   }.
```

Παρατηρούμε πως το πρόγραμμα αυτό έχει το όνομα " **absolute** " (λόγω της πρώτης γραμμής). Στην αρχή ορίζουμε πως τα αναγνωριστικά "x", "tempx" και "y" ανταποκρίνονται σε ακέραιες μεταβλητές που θα χρησιμοποιηθούν.
Ορίζουμε την συνάρτηση abs η οποία παίρνει την παράμετρο x με τιμή και μέσω των ελέγχων από τις εντολές if και else, υπολογίζει την απόλυτη τιμή της παραμέτρου, την οποία επιστρέφει μέσω της μεταβλητής tempx.
Στο κύριο μέρος του προγράμματος (που αναγνωρίζουμε πως βρίσκεται μετά το σχόλιο # main#) αρχικά ζητείται από τον χρήστη η τιμή της μεταβλητής x και καλείται η συνάρτηση με παράμετρο την τιμή αυτή, αποθηκεύοντας το αποτέλεσμα στη μεταβλητή y.
Τέλος, η τιμή της y εκτυπώνεται με την χρήση της εντολής print.

2)Λεκτική Ανάλυση

2.α) Θεωρία

Η λεκτική ανάλυση είναι η πρώτη φάση της μεταγλώττισης και σε συνδυασμό με την συντακτική ανάλυση ελέγχουν αν το πρόγραμμα που θέλουμε να δημιουργήσουμε είναι συντακτικά ορθό. Υλοποιείται στη συνάρτηση lex() (υπάρχει με την ίδια ονομασία και στον κώδικα μας) και έχει ως στόχο να αναγνωρίσει **λεκτικές μονάδες**. Οι λεκτικές μονάδες είναι διαφορετικές ανάλογα την γραμματική της γλώσσας, αλλά στην περίπτωση μας μια λεκτική μονάδα μπορεί να είναι:

- ένα αναγνωριστικό (πχ thisisanid)

- ένας αριθμός (πχ 525)

- ένα σύμβολο πράξης ('+', '-', '*', '/', '=')

- σύμβολο σύγκρισης (' > ', ' < ', ' <= ', ' >= ', ' <> ')

- σ. ανάθεσης (:=)

- σ. διαχωρισμού (' ; ', ' , ' , ' : ')

- σ. ομαδοποίησης (' (', ') ' , ' [', '] ', ' { ', ' } ')

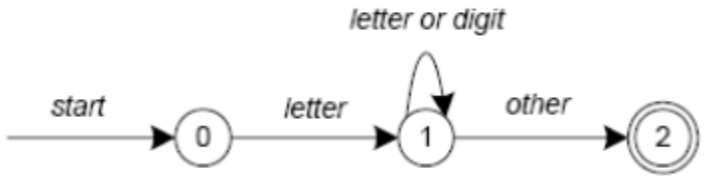
- σ. σχόλιου (#)

- σ. λευκού χαρακτήρα (' ', ' \t ', ' \n ')

- σ. τέλος προγράμματος (.)

Η λογική που ακολουθεί είναι αυτή των αυτομάτων κατάστασης, δηλαδή υπάρχουν καταστάσεις στις οποίες παραμένει ή μεταβαίνει, ανάλογα του ψηφίου/χαρακτήρα/συμβόλου που διαβάζει. Η συνάρτηση lex() με κάθε εκτέλεση της αρχίζει από την αρχική κατάσταση, την κατάσταση 0 και προχωράει σε επόμενες, μέχρι να φτάσει στην τελική ή σε κατάσταση σφάλματος ώστε να σταματήσει την εκτέλεση της.

Υπάρχουν 2 τρόποι υλοποίησης του λεκτικού αναλυτή, με πίνακα ή με σειρά εντολών απόφασης. Για λόγους απλότητας επιλέξαμε την υλοποίηση σειράς εντολών απόφασης. Στο σχήμα παρακάτω (από την αντίστοιχη διαφάνεια του διδάσκοντα) αναλύεται η περίπτωση της λεκτικής μονάδας αναγνωριστικού:




```
letter(letter|digit)*
```

Πιο συγκεκριμένα, από την κατάσταση 0, άμα αναγνωρίσουμε χαρακτήρα, μεταβαίνουμε στην κατάσταση 1 και παραμένουμε σε αυτή όσο αναγνωρίζουμε χαρακτήρα ή αριθμητικό ψηφίο. Οτιδήποτε άλλο, μας μεταβαίνει στην τελική κατάσταση.

Για να το πετύχει αυτό η συνάρτηση lex() επιστρέφει tokens. Σε αυτά υπάρχει το token_type και το token_string. Το token_type είναι δείκτης της ιδιότητας της λεκτικής μονάδας, ενώ το token_string είναι η συμβολοσειρά που διαβάζει η συνάρτηση. Για παράδειγμα, στην ανάγνωση της λεκτικής μονάδας "tempx" η συνάρτηση επιστρέφει token_type = "id" (ως αναγνωριστικό δλδ) και ως token_string = "tempx"

Στη περίπτωση λεκτικής μονάδας ">" το token_type = "greater than" (μεγαλύτερο από) και το token_string = ">".

Το token_type της λεκτικής μονάδας είναι ιδιαίτερα σημαντικό για τη φάση της συντακτικής ανάλυσης, ώστε να αναγνωρίζουμε ποια θα πρέπει να είναι η επόμενη ενέργεια της.

2.β) Ανάλυση αντίστοιχου κώδικα

Αρχικά οφείλουμε να αναφερθούμε στις μεταβλητές που χρησιμοποιούνται στην τωρινή φάση της μεταγλώττισης. Κατά σειρά εμφάνισης στο κώδικα:

- Οι σταθερές min_value και max_value απεικονίζουν τα όρια των αριθμών που μπορούν να εμφανιστούν και χρησιμοποιούνται στην κατάσταση 2 (την digit_state, δηλαδή την κατάσταση αναγνώρισης αριθμών)
- Οι πίνακες letters για την κατάσταση 1 (word_state) και digits για την κατάσταση 2 (digit_state) έχουν την πληροφορία όλων των λατινικών χαρακτήρων και δεκαδικών ψηφίων αντίστοιχα.
- Οι πίνακες num_symbols, rel_symbols για αριθμητικές πράξεις και συγκριτικές αντίστοιχα. Τα στοιχεία του πίνακα num_symbols οδηγούν απευθείας στην κατάσταση 7, την τελική (final_state). Από τα rel_symbols το σύμβολο "<" οδηγεί στην κατάσταση 3 (lesser_than state) ενώ το σύμβολο ">" οδηγεί στην κατάσταση 4 (greater_than_state)
- Η σταθερά dec_symbol οδηγεί στη τελική κατάσταση από την κατάσταση 5, της άνω κάτω τελείας (colon_state)
- Ο πίνακας sep_symbols χρησιμοποιείται για τα διαχωριστικά σύμβολα, από τον οποίο το σύμβολο ":" οδηγεί στην κατάσταση 5.
- Ο πίνακας group_symbols που περιλαμβάνει τα σύμβολα ομαδοποίησης όπως και η σταθερά comma_symbol οδηγούν απευθείας σε τελική κατάσταση.
- Το EOF_symbol που σηματοδοτεί το τέλος του προγράμματος πηγαίνοντας στην κατάσταση 8 EOF_state
- Το cmt_symbol το οποίο μεταβαίνει στη κατάσταση 6, των σχόλιων (comment_state). Στη κατάσταση αυτή, οποιοδήποτε σύμβολο/χαρακτήρας/ψηφίο αγνοείται, πέρα από το EOF_symbol (οδηγεί σε κατάσταση σφάλματος, εκτυπώνοντας την πληροφορία του Error_comments) και την $2^{η}$ εμφάνιση του cmt_symbol που σηματοδοτεί την έξοδο από την κατάσταση αυτή και την επιστροφή στην αρχική κατάσταση.

- Ο πίνακας white_characters περιέχει όλους τους λευκούς χαρακτήρες η εμφάνιση των οποίων ενημερώνει το αυτόματο να παραμείνει στην αρχική κατάσταση.
- Στη συνέχεια υπάρχουν οι καταστάσεις που αναφέρθηκαν πριν με την αντίστοιχη τους τιμή και συμπληρώνονται στον πίνακα lex_states. Στη συνάρτηση του lex() αρχικά γίνεται η ανάθεση της start_state στην τοπική μεταβλητή state ώστε να πραγματοποιείται η λειτουργία του αυτομάτου εφόσον δεν βρισκόμαστε στην τελική κατάσταση.
- Οι υπόλοιπες σταθερές αναφέρονται στα σφάλματα που προκύπτουν κατά την διάρκεια εκτέλεσης της συνάρτησης του λεκτικού και ο πίνακας reserved περιέχει όλες τις δεσμευμένες λέξεις της γλώσσας Cimple. Αν ο λεκτικός αναλυτής αναγνωρίσει κάποια από αυτές, τότε αναθέτει το token_type ίσο με το token_string, ώστε να μπορέσει στη συνέχεια ο συντακτικός αναλυτής να την χειριστεί κατάλληλα.
- Ο καθολικός πίνακας lex_result αποθηκεύει κάθε φορά το αποτέλεσμα της lex() ώστε να μπορέσουν στη συνέχεια οι συναρτήσεις της συντακτικής ανάλυσης να έχουν καθολικά την πληροφορία αυτή με κάθε μετάβαση από μια συνάρτηση σε άλλη.

Στην υποενότητα του κώδικα "File handling" γίνεται ο έλεγχος της ορθής εκτέλεσης στο τερματικό του προγράμματος και στη συνέχεια το άνοιγμα του αρχείου (file) για την ανάγνωση του. Συγκεκριμένα, ελέγχει το πλήθος ορισμάτων και αν το πλήθος είναι ένα, σημαίνει πως έχει εκτελεστεί μόνο το κομμάτι python final_3090_4271 και δεν υπάρχει .ci αρχείο για να μεταγλωττιστεί. Αν το πλήθος όμως είναι μεγαλύτερο του 2 σημαίνει πως έχουν δοθεί παραπάνω από ένα αρχεία. Και στις δύο περιπτώσεις, εκτυπώνονται κατάλληλα μηνύματα, ώστε να κατανοήσει ο χρήστης την λανθασμένη εκτέλεση του.

Στην υποενότητα του κώδικα "Lexical Analysis", πραγματοποιείται η κανονική Λεκτική Ανάλυση. Πριν τον ορισμό της lex(), ορίζεται η συνάρτηση printError(error, line) η οποία εκτυπώνει το μήνυμα που δίνεται στην παράμετρο error μαζί με την γραμμή (line) του αρχείου .ci στο οποίο παρουσιάστηκε το σφάλμα αυτό.

<u>Ανάλυση της συνάρτησης του λεκτικού αναλυτή:</u>

Γίνεται η ανάθεση της τοπικής μεταβλητής state στην αρχική κατάσταση. Αρχικοποιούνται οι τοπικές μεταβλητές:

file_pos: για την θέση της κεφαλής στο αρχείο .ci

prod_word: για την περίπτωση παραγωγής αναγνωριστικού

prod_num: αντίστοιχα για νούμερα

token_type & token_string: τα αποτελέσματα της συνάρτησης που αποθηκεύονται στον πίνακα **result** ως result[0] και result[1] αντίστοιχα.

Όσο η κατάσταση στην οποία βρισκόμαστε δεν είναι τελική, εκτελούμε το αυτόματο κατάστασης. Δίνεται η θέση της τωρινής κεφαλής του αρχείου στην file_pos μέσω της file_tell() και η τοπική μεταβλητή **char** διαβάζει τον πρώτο χαρακτήρα που αναγνωρίζει η κεφαλή. Η μεταβλητή file_pos αυξάνεται κατά ένα κάθε φορά που μεταβαίνουμε σε τελική κατάσταση.

Κάθε φορά που μπαίνουμε σε μια κατάσταση ενημερώνονται κατάλληλα τα token_type και token_string με τον τρόπο που περιγράψαμε παραπάνω, επομένως οι αξιόλογες παρατηρήσεις είναι στις παρακάτω περιπτώσεις :

Αν το char αναγνωρίσει:

- **'\n'** , αυξάνεται κατά ένα η καθολική μεταβλητή **file_line**
- Αναγνωριστικό ή ψηφίο τα προσθέτει στο prod_word ή prod_num αντίστοιχα.

Αν βρισκόμαστε στην αρχική κατάσταση και δεν αναγνωριστεί κάποια περίπτωση των if και elif, μεταβαίνοντας στην συνθήκη else σημαίνει πως παρουσιάστηκε μη έγκυρο σύμβολο και εκτελείται η printError με το **Error_non_valid_symbol** ως error.

Στη κατάσταση 1, ελέγχουμε αν η παραγόμενη λέξη έχει μήκος μεγαλύτερο του 30, το οποίο είναι σφάλμα, οπότε εκτελείται ομοίως το **Error_string_length**.

Στην κατάσταση 2, ελέγχουμε αν εμφανίζεται αλφαβητικός χαρακτήρας αντί για ψηφίο, οπότε εκτελείται ομοίως το **Error_letter_after_digit**. Επιπλέον γίνεται έλεγχος της τιμής του με μετατροπή σε ακέραιο και, αν είναι εκτός ορίων, εκτελείται ομοίως το **Error_out_of_bounds**.

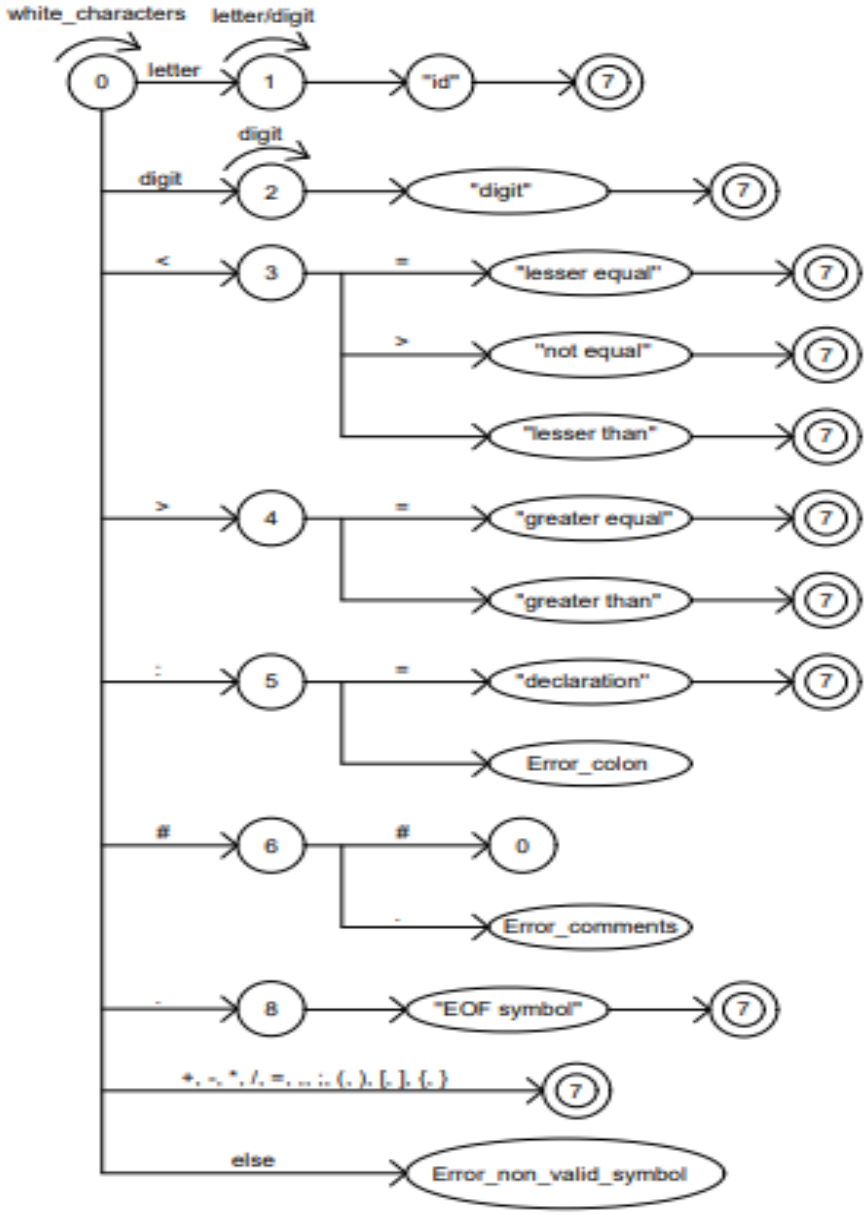
Στην κατάσταση 5, αν δεν εμφανιστεί το σύμβολο "=" μετά το σύμβολο ":" εκτελείται ομοίως το **Error_colon**.

Στην κατάσταση 6, αν εμφανιστεί το σύμβολο "." μετά το σύμβολο "#" εκτελείται ομοίως το **Error_comments**. Στην κατάσταση 8, ελέγχουμε αν αναγνωρίζεται κώδικας μετά του EOF_symbol. Αν ναι, εκτυπώνεται αντίστοιχο Warning μήνυμα στο τερματικό για να ειδοποιήσει τον χρήστη.

Στην κατάσταση 7 (τελική) εκτελείται η file.seek(file_pos) ώστε να αποθηκευτεί η κεφαλή στην τωρινή θέση του αρχείου για την επόμενη εκτέλεση.

Τέλος, γίνεται ο έλεγχος για τις δεσμευμένες λέξεις και αποθηκεύεται το token στο result που επιστρέφει η συνάρτηση.

Σχηματικά το αυτόματο:



2.γ) Παράδειγμα εκτέλεσης:

Για το παρακάτω πρόγραμμα cimple οι επαναληπτικές εκτελέσεις της συνάρτησης lex έχει τα παρακάτω αποτελέσματα:

```
program inpout

   declare x;

   #
    we can write whatever we want here
    there are no limitations
   #
   {

   input(x);
   print(x);

   }.
```

Αριθμός εκτέλεσης	**result[0] (Token_type)**	**result[1] (Token_string)**
1	program	program
2	id	inpout
3	declare	declare
4	id	x
5	question mark	;
6	left bracket	{
7	input	input
8	right parenthesis	(
9	id	x
10	left parenthesis	)
11	question mark	;
12	print	print
13	right parenthesis	(
14	id	x
15	left parenthesis	)
16	question mark	;
17	right bracket	}
18	EOF symbol	.

3)Συντακτική ανάλυση

3α) Θεωρία

Όπως αναφέρθηκε και προηγουμένως, ο συντακτικός αναλυτής χρησιμοποιείται για τον έλεγχο της συντακτικής ορθότητας του κώδικα. Είναι μια συλλογή συναρτήσεων που εκτελούνται με τη σειρά που ορίζει η γραμματική της προγραμματιστικής γλώσσας που θέλουμε να μεταγλωττίσουμε.

Δουλεύει άμεσα με την λεκτική ανάλυση καθώς καλεί την συνάρτηση lex για να αναγνωρίσει αν οι λεκτικές μονάδες που επιστρέφει, είναι σύμφωνες με την πορεία των γραμματικών κανόνων της γλώσσας.

Για καλύτερη κατανόηση θα χρησιμοποιηθούν 2 παραδείγματα της γραμματικής της Cimple, με την αντιστοιχία της σε κώδικα. Πιο συγκεκριμένα, για τις γραμματικές

α)

program : program ID block . :

```
def program():
   token = lex()
   if token_type == "program" :
      token = lex()
      if token_type == "id":
         token = lex()
         block()
         if token_type == "EOF symbol":
            return
         else:
            print("Error: EOF symbol wasn't detected")
      else:
         print("Error: no program name detected")
   print("Error: no 'program' keyword detected")
```

Με την εκτέλεση της συνάρτησης program αρχικά ελέγχουμε αν το token_type, που δόθηκε ως αποτέλεσμα από την lex στη μεταβλητή token, είναι ίσο με τη δεσμευμένη λέξη “program” που σηματοδοτεί την αρχή του προγράμματος. Στη περίπτωση που το πρόγραμμα δεν αρχίζει με την λέξη αυτή, εκτυπώνεται το μήνυμα "Error: no 'program' keyword detected".

Συνεχίζοντας, ξαναεκτελούμε την lex() για να αναγνωρίσουμε αν η επόμενη λεκτική μονάδα, αν ανταποκρίνεται σε αναγνωριστικό (δηλαδή το όνομα του προγράμματος). Ξανακαλούμε την lex και προχωράμε στην εκτέλεση της συνάρτησης block().

Εφόσον επιστρέψουμε από την συνάρτηση αυτή, ελέγχουμε αν η λεκτική μονάδα είναι το ειδικό σύμβολο τέλος προγράμματος. Στην αντίθετη περίπτωση, ο συντακτικός αναλυτής δεν αναγνωρίζει το τέλος του προγράμματος στο αρχείο .ci και ενημερώνει τον χρήστη για την έλλειψη αυτή.

β) elsepart : else statements | ε :

```
def elsepart():
   if token_type == "else":
      token = lex()
      statements()
   return
```

Σε αυτό το παράδειγμα, ομοίως με πριν ελέγχουμε αν το token_type της λεκτικής μονάδας που επιστρέφει η lex( ) (από την συνάρτηση που καλεί την elsepart( ))είναι με την δεσμευμένη λέξη "else". Αν ναι, ξανακαλούμε τον λεκτικό αναλυτή και προχωράμε στην εκτέλεση της συνάρτησης statements( ) η οποία στη συνέχεια θα καλέσει τις υπόλοιπες συναρτήσεις, εξαρτώμενη από το αποτέλεσμα της lex( )
Στην περίπτωση όμως που δεν αναγνωρίζουμε τη συγκεκριμένη μονάδα ή επιστρέφουμε από την συνάρτηση statements, επιστρέφουμε στη συνάρτηση που κάλεσε την elsepart() και συνεχίζεται η συντακτική ανάλυση.

3β) Ανάλυση αντίστοιχου κώδικα

Στη τελική μορφή του μεταφραστή, οι συναρτήσεις υπεύθυνες για την συντακτική ανάλυση, είναι εμπλουτισμένες με συναρτήσεις και γενικά επιπλέον γραμμές για τις επόμενες φάσεις μεταγλώττισης, αλλά προς το παρόν θα αναλύσουμε την αντιστοιχία της γραμματικής σε κώδικα. Για την ενημέρωση σφαλμάτων, χρησιμοποιούμε την συνάρτηση printError().

Η εναρκτήρια συνάρτηση της διαδικασίας συντακτικής ανάλυσης, είναι η syntax() η οποία στη συνέχεια εκτελεί την program() η οποία αναφέρθηκε και προηγουμένως στο παράδειγμα α). Αυτή η συνάρτηση δημιουργήθηκε ώστε μετά την επιτυχημένη εκτέλεση της συνάρτησης program και των επόμενων, να εκτυπώνει το μήνυμα "Syntax Analysis completed successfully" . Επιπλέον στη συνέχεια θα είναι υπεύθυνη για τον χειρισμό αρχείων που θα δημιουργούνται με την ολοκλήρωση της μεταγλώττισης.

Ακολουθώντας την γραμμική της γλώσσας Cimple έχουμε τις συναρτήσεις: (**Σημείωση**: με μπλε συμβολίζονται οι συναρτήσεις, ενώ με πράσινο οι δεσμευμένες λέξεις)

- **program()** : όπως αναλύθηκε προηγουμένως.
- **block()** : declarations subprograms statements
  Αυτή η συνάρτηση καλείται για να ελέγξει αν το πρόγραμμα που μεταγλωττίζουμε έχει μεταβλητές μέσω της συνάρτησης declarations. Στη συνέχεια προχωράει στη συνάρτηση subprograms για τον χειρισμό υποπρογραμμάτων (δηλαδή διαδικασιών ή συναρτήσεων) και τέλος προχωράει στη συνάρτηση statements για τον υπόλοιπο κώδικα του προγράμματος.
  Ενώ για την τωρινή φάση της μεταγλώττισης είναι μια συνάρτηση που καλεί τις αναφερόμενες, είναι ιδιαίτερα σημαντική για τις επόμενες φάσεις, καθώς θα χειριστεί την σωστή λειτουργία των συναρτήσεων και του κύριου προγράμματος. Είναι επίσης σημαντικό να αναφέρουμε πως η συνάρτηση block καλείται αναδρομικά μετά τον ορισμό ενός υποπρογράμματος.

- **declarations()** : (declare varlist ;) ∗
  Χειρισμός των μεταβλητών του προγράμματος που μεταγλωττίζουμε, όσες φορές εμφανίζεται η δεσμευμένη λέξη "declare" ως αποτέλεσμα της lex(). Αν εμφανιστεί, καλεί την varlist() για να ελέγξει τα ονόματα των μεταβλητών. Μετά την ολοκλήρωση της συνάρτησης, όπως και σε επόμενες εντολές, απαιτείται η lex() να αναγνωρίσει το ειδικό σύμβολο του ελληνικού ερωτηματικού. Αν δεν το αναγνωρίσει, εκτυπώνουμε μήνυμα λάθους.

- **varlist()** : ID (, ID)∗ | ε
  Αν η lex() επιστρέψει αναγνωριστικό (id) και όσες φορές εμφανίζεται το ειδικό σύμβολο του κόμματος εκτελούμε τον έλεγχο των ονομάτων μεταβλητών. Στη περίπτωση που εμφανίζεται το κόμμα χωρίς αναγνωριστικό μετά, εκτυπώνουμε μήνυμα λάθους.

- **subprograms** : (subprogram)∗
  Όσες φορές εμφανίζεται η δεσμευμένη λέξη procedure ή function ως αποτέλεσμα της lex() εκτελούμε την subprogram() για τον χειρισμό υποπρογραμμάτων.

- **subprogram()** : function ID (formalparlist) block
  | procedure ID (formalparlist) block
  Αναγνωρίζεται η αντίστοιχη δεσμευμένη λέξη υποπρογράμματος, ελέγχεται η ύπαρξη id και στη συνέχεια μέσα σε παρενθέσεις ο ορισμός της μορφής παραμέτρων που ορίζουν το υποπρόγραμμα μέσω της formalparlist

- **formalparlist()** : formalparitem (, formalparitem)∗ | ε
  Επαναληπτική εκτέλεση της formalparitem για μια ή παραπάνω παραμέτρους διαχωρισμένες από κόμμα.

- **formalparitem()** : in ID | inout ID
  Οι δεσμευμένες λέξεις που αναγνωρίζονται αντιστοιχούν στον τύπο μετάδοσης τιμής της παραμέτρου με το αντίστοιχο id. Στην περίπτωση έλλειψης αναγνωριστικού, εκτυπώνεται μήνυμα σφάλματος.

- **statements()** : statement ;
  | { statement (; statement)∗ }
  Εκτέλεση της statement εφόσον βρίσκεται μέσα σε αγκύλες ('{', '}') ή/και το τέλος κάθε statement ακολουθείται από το ελληνικό ερωτηματικό.

- **statement()** : assign_stat | if_stat | while_stat | switchcase_stat
  | forcase_stat | incase_stat | call_stat | return_stat
  | inputStat | print_stat | ε
  Ανάλογα του τύπου της λεκτικής μονάδας, εκτελείται η συνάρτηση που χειρίζεται την αντίστοιχη δεσμευμένη λέξη ή αναγνωριστικό ή επιστροφή αν δεν εμφανίσει τίποτα από αυτά.

- **assign_stat()** : ID := expression
  Στη περίπτωση που η statement αναγνωρίσει ένα αναγνωριστικό κατανοούμε πως αναφερόμαστε στην ανάθεση τιμής σε μια μεταβλητή μέσω του ειδικού συμβόλου ":=" και στη συνέχεια εκτελούμε την expression για να ελεγχθεί η ανάθεση.

- **if_stat()** : if (condition) statements elsepart
  Αν αναγνωρίσουμε την δεσμευμένη λέξη if μεταβαίνουμε στον χειρισμό συνθηκών στο εσωτερικό των παρενθέσεων, η έλλειψη των οποίων εκτυπώνει μήνυμα σφάλματος. Στη συνέχεια εκτελείται η statements για έλεγχο των επόμενων λεκτικών μονάδων και η elsepart για τον έλεγχο εμφάνισης αντίθετης συνθήκης.

- **elsepart()**: όπως αναλύθηκε προηγουμένως

- **while_stat()** : while (condition) statements
  Λειτουργεί ως επαναληπτική μέθοδος του προγράμματος με έλεγχο των συνθηκών στο εσωτερικό παρενθέσεων και εκτέλεσης της statements στην συνέχεια.

- **switchcase_stat()** : switchcase (case (condition) statements)* default statements
  Η switchcase δουλεύει όπως η if για μια ή παραπάνω συνθήκες που εμφανίζονται μέσα στη παρένθεση μετά την δεσμευμένη λέξη case. Μετά από τον έλεγχο των συνθηκών case, μεταβαίνει στον έλεγχο για την ύπαρξη της δεσμευμένης λέξης default.

- **forcase_stat()** : forcase (case (condition) statements)* default statements
  Ακριβώς ίδια σύνταξη συνάρτησης με την switchcase, αλλά το forcase ως πρώτος έλεγχος. Η forcase δουλεύει σαν επαναληπτική switchcase

- **incase_stat()** : incase (case (condition) statements)* default statements
  Ακριβώς ίδια σύνταξη συνάρτησης με την switchcase, αλλά το incase ως πρώτος έλεγχος. Όμοια λειτουργία με την forcase

- **return_stat()** : return(expression)
  Αναγνωρίζοντας την "return" μεταβαίνει στην εκτέλεση της expression() για τον έλεγχο της αριθμητικής παράστασης που επιστρέφει μια συνάρτηση.

- **call_stat()** : call ID(actualparlist)
  Αναγνωρίζοντας την "call" και στη συνέχεια αναγνωριστικό, μεταβαίνει στην ανάθεση παραμέτρων μέσω της actualparlist για το κάλεσμα διαδικασίας.

- **print_stat()** : print(expression)
  Όμοια σύνταξη με την return_stat, για την εκτύπωση της παράστασης.

- **input_stat()** : input(id)
  Όμοια σύνταξη με την return_stat, για το διάβασμα του αναγνωριστικού από τον χρήστη.

- **actualparlist()** : actualparitem (, actualparitem)∗ | ε
  Ίδια σύνταξη με την formalparlist() αλλά με εκτέλεση της actualparitem.

- **actualparitem()** : : in expression | inout ID
  Όμοια σύνταξη με την formalparitem(), χρησιμοποιείται για τις παραμέτρους που δίνονται σε ένα υποπρόγραμμα όταν το εκτελούμε μετά τον ορισμό του.

- **condition()** : boolterm (or boolterm)∗
  Ελέγχει τις εκτελέσεις της boolterm όσο εμφανίζεται η "or"

- **boolterm()** : boolfactor (and boolfactor)∗
  Ελέγχει τις εκτελέσεις της boolfactor όσο εμφανίζεται η "and"

- **boolfactor()** : not [condition] | [condition] | expression relational_op expression
  Ελέγχει την κατάλληλη σύνταξη στην περίπτωση εμφάνισης των λεκτικών μονάδων λογικής συνθήκης. Αν δεν εμφανίζονται αυτά, ελέγχει την ύπαρξη αριθμητικής παράστασης από την expression με τελεστές σύγκρισης από την relational_op().

- **expression()** : optional_sign term (add_op term)∗
  Αρχικά γίνεται έλεγχος αριθμητικού συμβόλου με την εκτέλεση της optional_sign()και, αν υπάρχει, μεταβαίνει στην term. Εφόσον επιστρέψει, ανιχνεύει επαναληπτικά την ύπαρξη αριθμητικού συμβόλου μέσω της add_op για να δημιουργήσει μια εξίσωση αντίστοιχης πράξης.

- **term()** : factor (mul_op factor)∗
  Εκτελεί την factor και εφόσον επιστρέψει ελέγχει επαναληπτικά ύπαρξη συμβόλου "*", "/" με την mul_op εκτελώντας την factor.

- **factor()** : number | (expression) | ID idtail
  Ελέγχει αν η λεκτική μονάδα είναι αριθμός, αριθμητική παράσταση μέσα σε παρένθεση ή αναγνωριστικό. Το αναγνωριστικό μπορεί να ανταποκρίνεται είτε σε μεταβλητή είτε σε συνάρτηση και για τον διαχωρισμό αυτό, εκτελείται η idtail().

- **idtail()** : (actualparlist) | ε
  Ελέγχει το άνοιγμα παρένθεσης που σημαίνει πως το id που αναγνωρίστηκε από την factor ανταποκρίνεται σε συνάρτηση και εκτελείται η actualparlist για ανάθεση παραμέτρων με το κάλεσμα της συνάρτησης.

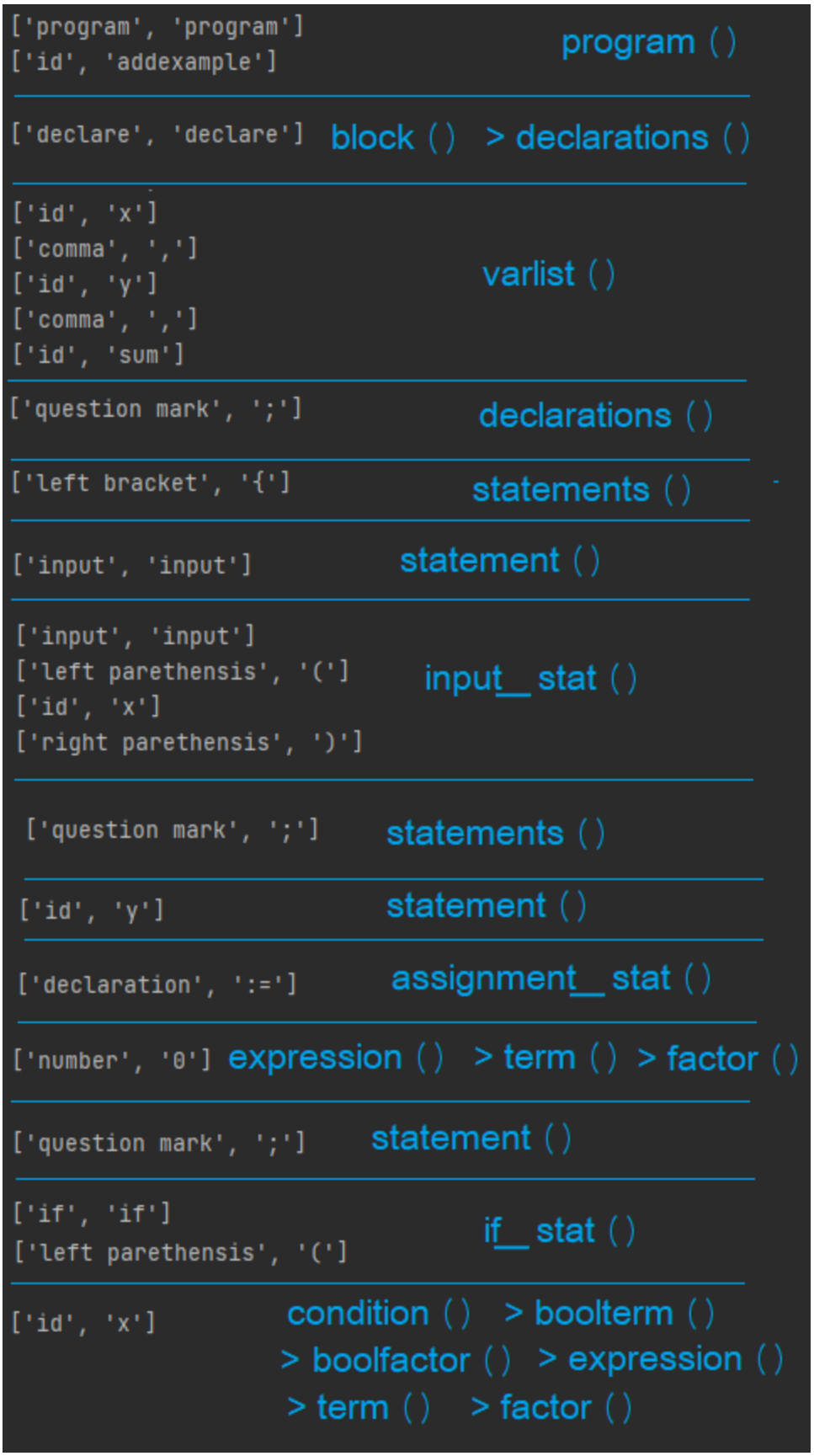
- **optional_sign()** : add_op | ε
  Ελέγχει την ύπαρξη σύμβολων πρόσθεσης, αφαίρεσης με την add_op
- **relational_op()** : = | <= | >= | > | < | <> (αναγνώριση των συμβόλων σύγκρισης)
- **add_op()** : + | - (αναγνώριση των συμβόλων πρόσθεσης, αφαίρεσης)
- **mul_op()** : * | / (αναγνώριση των συμβόλων πολλαπλασιασμού, διαίρεσης)

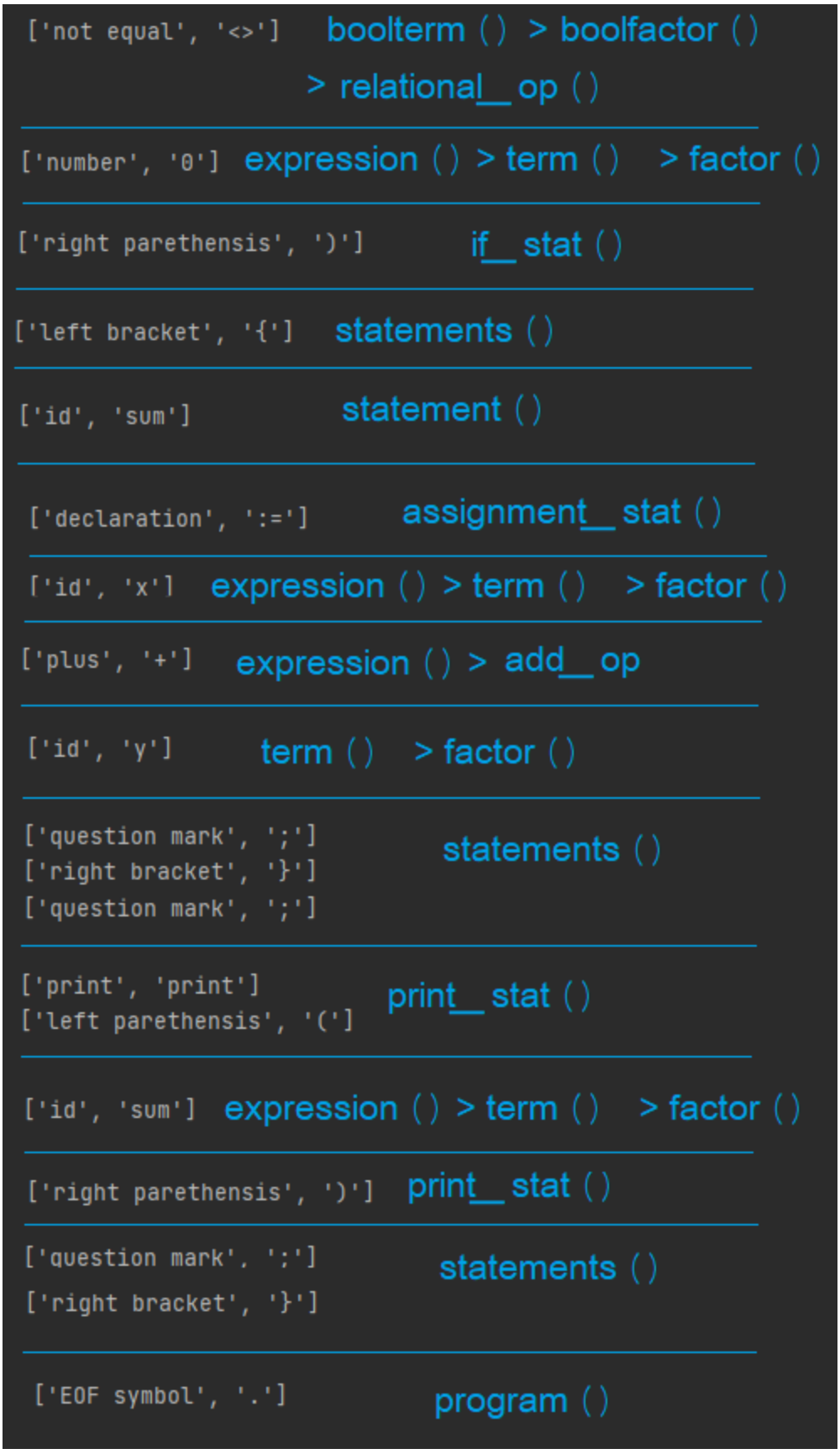
3γ) Παράδειγμα εκτέλεσης

Για την καλύτερη κατανόηση θα ακολουθήσει ένα σύντομο παράδειγμα εκτέλεσης συντακτικής ανάλυσης, με αναφορά την σειρά εκτέλεσης των συναρτήσεων ενός ολοκληρωμένου προγράμματος:

```
program addexample

   declare x,y,sum;

   #main#

   {
      input(x);
      y :=0;
        if (x <>0) {
            sum := x + y;
        };
      print(sum);
   }.
```

Στα αριστερά είναι το αποτέλεσμα της lex μορφής [‘token_type’, ‘token_string’] και στα δεξιά με μπλε χρώμα παρουσιάζεται συνάρτηση της συντακτικής ανάλυσης που βρισκόμαστε την στιγμή εκείνη.


['program', 'program']
['id', 'addexample']
program ()
['declare', 'declare']
block () > declarations ()
['id', 'x']
['comma', ',']
['id', 'y']
['comma', ',']
['id', 'sum']
varlist ()
['question mark', ';']
declarations ()
['left bracket', '{']
statements ()
['input', 'input']
statement ()
['input', 'input']
['left parethensis', '(']
['id', 'x']
['right parethensis', ')']
input_stat ()
['question mark', ';']
statements ()
['id', 'y']
statement ()
['declaration', ':=']
assignment_stat ()
['number', '0']
expression () > term () > factor ()
['question mark', ';']
statement ()
['if', 'if']
['left parethensis', '(']
if_stat ()
['id', 'x']
condition () > boolterm ()
> boolfactor () > expression ()
> term () > factor ()


['not equal', '<>'] boolterm () > boolfactor ()
> relational_ op ()
['number', '0'] expression () > term () > factor ()
['right parethensis', ')'] if_ stat ()
['left bracket', '{'] statements ()
['id', 'sum'] statement ()
['declaration', ':='] assignment_ stat ()
['id', 'x'] expression () > term () > factor ()
['plus', '+'] expression () > add_ op
['id', 'y'] term () > factor ()
['question mark', ';']
['right bracket', '}']
['question mark', ';'] statements ()
['print', 'print']
['left parethensis', '('] print_ stat ()
['id', 'sum'] expression () > term () > factor ()
['right parethensis', ')'] print_ stat ()
['question mark', ';']
['right bracket', '}'] statements ()
['EOF symbol', '.'] program ()

4)Παραγωγή ενδιάμεσου κώδικα

4α) Θεωρία

Σε αυτή τη φάση μεταγλώττισης εφόσον έχουμε εξασφαλίσει πως ο κώδικας μας είναι συντακτικά σωστός, θα κάνουμε τα αρχικά βήματα προς την παραγωγή κώδικα assembly. Για να γίνει όμως αυτό θα πρέπει να μετατρέψουμε τον κώδικα στις κατάλληλες εντολές. Επομένως θα χρησιμοποιήσουμε τις συναρτήσεις του συντακτικού αναλυτή, ώστε να παραγάγουμε τετράδες της μορφής [“εντολή”, “μεταβλητή x”, “μεταβλητή y”, “μεταβλητή z”]

Ορισμένα κατατοπιστικά παραδείγματα τετράδων αυτής της μορφής είναι τα παρακάτω:
α) Για την περίπτωση z := x + y (αντίστοιχα στην assembly add $t0,$t1,$t2 όπου $t0 = x, $t1 = y και $t2 = z) η τετράδα που δημιουργείται είναι: [“+”, “x” , “y” , “z”]
β) Αν έχουμε return x, η αντίστοιχη τετράδα είναι: [“retv”, “x”, “_”, “_”]

Οι τετράδες αυτές θα συμπληρωθούν ανάλογα την δεσμευμένη λέξη που εμφανίζεται στο πρόγραμμα. Για να το καταφέρουμε αυτό είναι απαραίτητες οι παρακάτω βοηθητικές συναρτήσεις:

- **nextquad():** επιστρέφει τον αριθμό της επόμενης τετράδας που θα δημιουργηθεί
- **genquad(op,x,y,z):** δημιουργεί την τετράδα δίνοντας τα κατάλληλα ορίσματα και προστίθεται στην λίστα με όλες τις τετράδες (στην περίπτωση του κώδικα μας αυτή η λίστα ονομάζεται **“quad_list”.**
- **newtemp():** δημιουργεί και επιστρέφει μια νέα προσωρινή μεταβλητή για να αποθηκεύσει μια τιμή που θα αναθέσουμε σε μια μεταβλητή του προγράμματος.
- **emptylist():** δημιουργεί μια κενή λίστα που θα συμπληρωθεί στην συνέχεια.
- **makelist(x):** δημιουργεί μια νέα λίστα με μοναδικό στοιχείο την παράμετρο x.
- **merge(list1, list2):** επιστρέφει την λίστα που δημιουργείται από την συνένωση των δυο λιστών που δέχεται ως παραμέτρους.
- **backpatch(list, z):** Η λίστα list περιέχει δείκτες σε τετράδες στις οποίες το 4^{ο} στοιχείο (μεταβλητή z) είναι ασυμπλήρωτο. Η συνάρτηση αυτή θα ανατρέξει την λίστα με όλες τις τετράδες για να βρει αυτές τις συγκεκριμένες και συμπληρώσει το κενό στοιχείο με τη μεταβλητή z που έχει ως παράμετρο.

Ιδιαίτερη σημασία έχει η έννοια του άλματος. Όπως γνωρίζουμε, στη γλώσσα assembly δεν υπάρχει η έννοια της επανάληψης και συνθήκης με τον ίδιο τρόπο που βλέπουμε σε γλώσσες όπως Python, Java, C. Υπάρχει όμως η εντολή j ή αλλιώς jump, η οποία κάνει μετάβαση στον προορισμό που ορίζουμε με την ταμπέλα της γραμμής στην οποία θα μεταβούμε. Αντίστοιχη εντολή υπάρχει και στην γλώσσα C, η εντολή “goto ‘label’ ”, την οποία χρησιμοποιούμε στην περίπτωση που δημιουργείται το αρχείο .c.

Το αναφερόμενο αυτό αρχείο, δημιουργείται μόνο στην περίπτωσή που το αρχείο .ci δεν περιέχει υποπρογράμματα, για λόγους απλότητας. Επιπλέον σε αυτή τη φάση δημιουργείται ένα αρχείο με κατάληξη .int το οποίο περιέχει όλες τις τετράδες, με νούμερο ετικέτας στην αρχή κάθε γραμμής.

4β) Ανάλυση αντίστοιχου κώδικα

Προτού αναφερθούμε στην διαδικασία παραγωγής τετράδας σε κάθε περίπτωση, αξίζει να αναλύσουμε τις μεταβλητές που χειριζόμαστε στη τωρινή φάση. Για λόγους ευκολίας η αρχικοποίηση τους βρίσκεται στην "υποενότητα" των συναρτήσεων και μεταβλητών για την φάση του ενδιάμεσου κώδικα.

- **quad_num**: Κρατάει το πλήθος των τετράδων που περιέχει η λίστα τους. Χρησιμοποιείται ως η μεταβλητή που επιστρέφει η **nextquad(),** καθώς το πλήθος αυτό είναι και ο αριθμός του δείκτη στην επόμενη τετράδα που θα δημιουργηθεί. Κάθε φορά που καλείται η genquad, αυξάνεται το πλήθος κατά ένα και κατά συνέπεια και το quad_num.
- **function_flag:** Boolean μεταβλητή η οποία αρχικοποιείται στην τιμή False. Μετατρέπεται σε True, αν κατά τη διάρκεια της συντακτικής ανάλυσης εντοπιστεί οποιαδήποτε από τις παρακάτω δεσμευμένες λέξεις: "function" , "procedure", "return", "call". Αν μετά τη συντακτική ανάλυση η τιμή αυτή είναι True, δεν δημιουργούμε το αρχείο .c, καθώς υπάρχουν υποπρογράμματα.
- **quad_list:** Όλες οι τετράδες αποθηκεύονται εδώ μέσα με κάθε κλήση της genquad.
- **T_value, T_value_list:** Όμοια λειτουργία με τα 2 παραπάνω, απλώς σε αυτή τη περίπτωση αφορούν τις προσωρινές μεταβλητές που δημιουργούνται με την κάθε κλήση της **newtemp().**
- **program_name:** Το όνομα του προγράμματος που ανατίθεται στο αναγνωριστικό μετά την δεσμευμένη λέξη program.
- **function_list, procedure_list:** Λίστες με τα ονόματα όλων των συναρτήσεων και διαδικασιών αντίστοιχα. Συμπληρώνονται κάθε φορά που ορίζεται σωστά ένα υποπρόγραμμα και χρησιμοποιούνται για να ελέγξουμε πως το υποπρόγραμμα που θέλουμε να καλέσουμε, έχει όντως οριστεί προηγουμένως.
- **variables:** Ομοίως με τα προηγούμενα, συμπληρώνεται κάθε φορά που γίνεται declare κάποια μεταβλητή στο πρόγραμμα. Αν κάπου στο πρόγραμμα υπάρχει κάποια μεταβλητή η οποία δεν ανήκει στη λίστα αυτή, εκτυπώνουμε μήνυμα σφάλματος.

Αξίζει να σημειωθεί πως η βοηθητική συνάρτηση newtemp() επιστρέφει συμβολοσειρές της μορφής "T_0, T_1, …" όπου τα ψηφία αυτά είναι κάθε φορά η μεταβλητή T_value που έχει μετατραπεί σε συμβολοσειρά με την χρήση της εντολής str().
Σε αυτό το σημείο θα αναφερθούμε στις τροποποιήσεις που κάναμε στις συναρτήσεις της συντακτικής ανάλυσης, οι οποίες προκύπτουν από τις οδηγίες των διαφανειών του μαθήματος.

- Αρχικά, η τετράδα που σηματοδοτεί την **αρχή** ενός προγράμματος / υποπρογράμματος με όνομα "name" είναι η ["begin_block", name, "_", "_"]. Επομένως, στη συνάρτηση block, μετά την εκτέλεση της συνάρτησης subprograms() γίνεται genquad της τετράδας αυτής. Γίνεται σε αυτό το σημείο, καθώς σε αυτό το στάδιο έχουν οριστεί οι συναρτήσεις και διαδικασίες (αν υπάρχουν) άρα αρχίζει και η ανάλυση τους. Σε αντίθετη περίπτωση αρχίζει η ανάλυση του κύριου προγράμματος στη περίπτωση που το name είναι ίσο με το program_name.

- Το **τέλος** του προγράμματος/ υποπρογράμματος αντιστοιχεί στη τετράδα [“end_block”, name, “_”, “_”] η οποία δημιουργείται στην block() μετά την κλήση της statements(), εφόσον έχει τελειώσει δηλαδή η ανάλυση του. Αξίζει να σημειωθεί, πως στη περίπτωση που βρισκόμαστε στο κύριο πρόγραμμα, πριν δημιουργηθεί η “end_block” τετράδα δημιουργείται η τετράδα τετράδα [“halt”, “_”, “_”, “_”] που ανταποκρίνεται στο τερματισμό του προγράμματος.

- Για τις **αριθμητικές πράξεις** τοποθετούμε τις βοηθητικές συναρτήσεις στις συναρτήσεις expression() και term() για την πρόσθεση/αφαίρεση και πολλαπλασιασμό/διαίρεση αντίστοιχα.
  Στην expression(): αρχικά αποθηκεύεται στις τοπικές μεταβλητές T1_place και T2_place το αποτέλεσμα της term() και αναγνωρίζοντας το κατάλληλο σύμβολο γίνεται η δημιουργία της τετράδας [“symbol”, “T1_place”, “T2_place”, “w”] όπου w η τοπική μεταβλητή που δημιουργεί η newtemp(). Στη w αποθηκεύεται το αποτέλεσμα της πράξης και θα συνεχίζεται να αποθηκεύεται για περισσότερες πράξεις και στο τέλος αποθηκεύεται στο E_place, το οποίο επιστρέφουμε.
  Στην term(): ακολουθούμε την ίδια διαδικασία, απλώς αυτή τη φορά έχουμε F1_place, F2_place που δέχονται το αποτέλεσμα της factor() και επιστρέφουμε την μεταβλητή T_place. Στην factor() επιστρέφουμε κάθε φορά την λεκτική μονάδα που αναγνωρίζουμε, ή την παράσταση που επιστρέφει η expression(), αποθηκεύοντας την στην F_place.

- Για τις **λογικές πράξεις** χρησιμοποιούμε τις condition() , boolterm(), boolfactor() .
  Αυτή τη φορά αντί για T1_place έχουμε μεταβλητές της μορφής B_true, B_false που αντιπροσωπεύουν το αποτέλεσμα, αν η λογική συνθήκη είναι αληθής/ψευδή. Και στις 3 αυτές συναρτήσεις, επιστρέφουμε μια λίστα που περιέχει την αντίστοιχη true στην πρώτη θέση και false στη δεύτερη για να μπορούμε να της αντλήσουμε.
  Στην condition: Αποθηκεύουμε στη λίστα Q1 το αποτέλεσμα της boolterm() και στα B τα 2 στοιχεία της. Κάνουμε backpatch την B_false στην επόμενη για την συμπλήρωση των τετράδων οι οποίες δεν έχουν συμπληρωθεί επειδή δεν ίσχυε η συνθήκη (λόγω της ύπαρξης λογικού Ή). Με την επόμενη κλήση της boolterm αποθηκεύουμε το αποτέλεσμα της στην λίστα Q2 , οι true τιμές B_true και Q2_list[0] συγχωνεύονται με την merge στην B_true ενώ στην μεταβλητή B_false δίνεται η τιμή της Q2_lis[1], ώστε να γίνει ο έλεγχος για την επόμενη επανάληψη. Τέλος, αποθηκεύονται οι τελικές τιμές B και επιστρέφονται με την B_list.
  Στην boolterm: Ίδια εκτέλεση με την condition, αλλά αυτή τη φορά έχουμε λογικό ΚΑΙ, οπότε αν δεν ισχύει η συνθήκη, κάνουμε έξοδο, επομένως αυτή τη φορά πρέπει να γίνει backpatch στις true τιμές και συγχώνευση στις false. Αντί για B_list , Q1_list και Q2_list έχουμε στις αντίστοιχες θέσεις Q_list, R1_list και R2_list.
  Στην boolfactor: Με την κάθε εκτέλεση της condition ελέγχουμε την B_list που επιστρέφει. Αν έχει αναγνωριστεί η δεσμευμένη “not” επιστρέφουμε την R_list, με αντεστραμμένες τις τιμές της B. Αν αναγνωρίσουμε τις λογικές παρενθέσεις, τότε την επιστρέφουμε με την ίδια μορφή εφόσον ξαναεκτελέσουμε την condition(). Αλλιώς, έχουμε μια ακολουθία σύγκρισης , οπότε αποθηκεύουμε τα αποτελέσματα της

expression() στα E1_place και E2_place μαζί με την relop (αποτέλεσμα της relational_op()) για να δημιουργήσουμε την κατάλληλη τετράδα. Επιπλέον δημιουργούμε την τετράδα ["jump", "_", "_", "_"] ώστε να συμπληρωθεί κατάλληλα ο προορισμός στη συνέχεια, με την αντίστοιχη backpatch ενέργεια.

- Η actualparitem() είναι υπεύθυνη για την δημιουργία τετράδων **παραμέτρων.** Αν η παράμετρος έχει τύπο μετάδοσης με τιμή, δημιουργούμε ["par", a_place, "CV", "_"] , ενώ με αναφορά ["par", b_place, "REF", "_"] όπου a_place και b_place το αποτέλεσμα της expression().
- Η **κλήση συνάρτησης** γίνεται στην idtail(). Αρχικά δημιουργούμε μια νέα τοπική μεταβλητή την οποία θα χρησιμοποιήσουμε για την δημιουργία της τετράδας ["par", w, "RET", "_"] για την τιμή επιστροφής της συνάρτησης και μετά δημιουργούμε την τετράδα ["call", func_name, "_", "_"] όπου func_name το όνομα της συνάρτησης που δέχεται ως όρισμα η id_tail.
- Εφόσον οι **διαδικασίες** δεν επιστρέφουν κάτι, για την **κλήση** τους, δημιουργούμε μέσα στην call_stat() την τετράδα ["call", name, "_", "_"], όπου name το όνομα της διαδικασίας, το οποίο ελέγχουμε πως βρίσκεται μέσα στην procedure_line, δηλαδή έχει οριστεί η διαδικασία επιτυχώς.
- Η **επιστροφή τιμής** γίνεται από την return_stat(), όπου δημιουργείται η ["retv", E_place, "_", "_"] και E_place το αποτέλεσμα της expression().
- Η **εκχώρηση τιμής** γίνεται από την assignment_stat(), όπου δημιουργείται η [":=", E_place, "_", "id_place"] , E_place το αποτέλεσμα της expression() και id_place το αναγνωριστικό που εντοπίστηκε. Και αυτή τη φορά, αν το id δεν είναι έγκυρη μεταβλητή, εκτυπώνεται μήνυμα σφάλματος.

- Η **δομή while** πραγματοποιείται στην while_stat(). Αρχικά αποθηκεύεται ο αριθμός της επόμενης τετράδας στην B_quad ώστε να επιστρέψουμε σε αυτή για να γίνεται επανάληψη μέσω jump. Εφόσον ελεγχθούν οι συνθήκες και αποθηκευτούν οι τιμές στην B_list, εάν η συνθήκη ισχύει, μεταβαίνουμε με backpatch της true τιμής στα statements, αλλιώς φεύγουμε από την επανάληψη με backpatch της false τιμής.

- Η **δομή if** πραγματοποιείται στην if_stat(). Με όμοιο σκεπτικό με την while, αποθηκεύουμε στη B_list τις τιμές της condition(). Με αυτό τον τρόπο, αν η συνθήκη είναι αληθής, με backpatch μεταβαίνουμε στον κώδικα που ισχύει η true τιμή, αλλιώς μεταβαίνουμε στο else κομμάτι (αν υπάρχει). Επίσης υπάρχει η ifList η οποία, με το backpatch της στο τέλος, εξασφαλίζει πως δε θα εκτελεστούν και οι false συνθήκες, αν ισχύουν οι true.

- Η **δομή switchcase** πραγματοποιείται στην switchcase_stat(). Αρχικά δημιουργείται η exitList, μια κενή λίστα που θα σημειώσει την διεύθυνση στην οποία θα κάνει jump ο κώδικας για να φύγει από την επανάληψη. Για κάθε περίπτωση case, αρχικά αποθηκεύουμε στην cond_list τις true/false τιμές της συνθήκης. Αν ισχύει η συνθήκη μέσω της backpatch μεταβαίνουμε στα statements, αλλιώς συνεχίζουμε τον έλεγχο κάθε case.

Αν δεν ισχύει κάποια true τιμή σε κάποιο case, μεταβαίνουμε στη περίπτωση της default ώστε να εκτελεστεί τότε η statements().

- Η **δομή forcase** πραγματοποιείται στην forcase_stat(). Στη μορφή του κώδικα που παραδόθηκε, έχει παρόμοια σύνταξη με την δομή της while, αλλά αντί για B_quad έχουμε p1quad και αντί για B_list, cond_list. Δεν καλύπτει όμως τον χειρισμό για την μετάβαση στην default συνθήκη.

- Η **δομή incase** πραγματοποιείται στην incase_stat(). Αρχικά δημιουργούμε την προσωρινή μεταβλητή w και αποθηκεύουμε την τιμή 1, δημιουργώντας τετράδα ανάθεσης [":=", 1, "_", w]. Σε κάθε case αποθηκεύουμε τα αποτελέσματα της συνθήκης στην cond_list την τιμή 0 στην w ώστε να μπορέσουμε να κάνουμε μετάβαση στην statements() μέσω backpatch της true τιμής της συνθήκης, αλλιώς κάνουμε έξοδο με backpatch της false τιμής. Κατά την έξοδο της επανάληψης θέτουμε αποθηκεύουμε την τιμή 0 στη προσωρινή μεταβλητή w.

- Η **είσοδος τιμής** γίνεται από την input_stat(), όπου δημιουργείται η ["inp", id_place, "_", "_ "] , με id_place το αναγνωριστικό που εντοπίστηκε. Αν το id δεν είναι έγκυρη μεταβλητή, εκτυπώνεται μήνυμα σφάλματος.

- Η **έξοδος τιμής** γίνεται από την print_stat(), όπου δημιουργείται η ["out", E_place, "_", "_ "] , με E_place το αποτέλεσμα της expression().

4.γ) Παραγωγή αρχείου .int

Εφόσον έχουμε δημιουργήσει τις τετράδες συμπληρώνοντας την quad_list και έχουμε εκτελέσει με επιτυχία την συντακτική ανάλυση, καλούμε την συνάρτηση int_file_create() στην syntax(), η οποία δημιουργεί το αρχείο .int. Το αρχείο αυτό έχει ως όνομα πριν την κατάληξη .int το program_name. Έχει αποθηκευμένο όλες τις τετράδες κατά σειρά δημιουργίας της μορφής για παράδειγμα 7: jump _ _ 3, όπου 7 η θέση της $8^{ης}$ τετράδας στην λίστα quad_list.

4.δ) Παραγωγή αρχείου .c

Παρομοίως με την δημιουργία του προηγούμενου αρχείου, το .c αρχείο, (αν είναι να δημιουργηθεί) χρησιμοποιεί το program_name ως όνομα πριν την κατάληξη .c. Δημιουργείται στην c_file_create(), καλώντας την από την syntax().

Αρχικά, μιας και είναι αρχείο C, πρέπει να σημειώσουμε στις συμβολοσειρές variable_string και temp_string τις μεταβλητές και προσωρινές μεταβλητές του προγράμματος που μεταγλωττίζουμε χωρισμένες από κόμμα και προσθέτοντας το σύμβολο “;” στο τέλος.

Στην πρώτη γραμμή του αρχείου σημειώνουμε (μέσω της c_file.write()) το #include <stdio.h>, καθώς θέλουμε να εκτελέσουμε τις εντολές printf() και scanf() που περιέχει η βιβλιοθήκη αυτή.

Στην επόμενη γραμμή σημειώνουμε το “int main() /n {” ώστε να σηματοδοτήσουμε την εκκίνηση του προγράμματος. Στη συνέχεια σημειώνουμε “int” και variable_string (αν υπάρχει) και

αντίστοιχα για το temp_string, ώστε να μπορέσει το πρόγραμμα να χρησιμοποιήσει τις ακέραιες μεταβλητές.

Για το εσωτερικό κομμάτι της main() ελέγχουμε το πρώτο όρισμα της κάθε τετράδας της quad_list και το αντιστοιχούμε με την κατάλληλη εντολή της C . Σε κάθε γραμμή υπάρχει η ταμπέλα "L_'νούμερο_τετράδας' " για την μετάβαση με jump (δηλαδή goto στην C). Μετά από κάθε εντολή, υπάρχει μέσα σε σχόλια η μορφή της αντίστοιχης τετράδας στην symbol_list.

Η αντιστοιχία των πρώτων ορισμάτων των τετράδων με τις εντολές της C παρουσιάζεται στον πίνακα παρακάτω:

quad_list[i][0]	**αντίστοιχη εντολή C**
:=	quad_list[i][3] = quad_list[i][1];
op = "+", "-", "*", "/"	quad_list[i][3] = quad_list[i][1] op quad_list[i][2];
=	**if** quad_list[i][1] == quad_list[i][2];
<>	**if** quad_list[i][1] **!=** quad_list[i][2];
relop_id = "=", ">", "<", "<>", ">=", "<="	**if** quad_list[i][1] **relop_id** quad_list[i][2];
jump	**goto** L_quad_list[i][3];
retv	**return** quad_list[i][1];
inp	**printf**("%d", quad_list[i][1]);
out	**scanf**("%d", &quad_list[i][1]);
halt	**return 0;**
αλλιώς	σχόλιο την τετράδα

Εφόσον τελειώσει η αντιστοιχία αυτή προστίθεται το σύμβολο "}" και κλείνουμε το αρχείο.

4.ε) Παράδειγμα εκτέλεσης

Θα παρουσιάσουμε δύο απλά παραδείγματα, α) ένα με μια απλή συνάρτηση και τον αντίστοιχο κώδικα .int και β) ένα με παράδειγμα που δημιουργεί το .c αρχείο.

α)

```
program delta

   # this is a simple func for testing the creation of only int file #

   declare a,b,c;

   function func(in a, in b, in c){
      return ( b*b - 4* a* c);
   }

   # main #
   {
   input(a);
   input(b);
   input(c);
   print(func(in a, in b, in c));
   }.
```

Αρχείο delta.int:

```
0: begin_block func _ _
1: * b b T_0
2: * 4 a T_1
3: * T_1 c T_2
4: - T_0 T_2 T_3
5: retv T_3 _ _
6: end_block func _ _
7: begin_block delta _ _
8: inp a _ _
9: inp b _ _
10: inp c _ _
11: par a CV _
12: par b CV _
13: par c CV _
14: par T_4 RET _
15: call func _ _
16: out func _ _
17: halt _ _ _
18: end_block delta _ _
```

β)

```
program multipleifs

   declare x,y,z,w;

   #main#
   {
      input(x);
      input(y);
      if(x>0){
         w := 0;
         z := 0;
      }else{
         w := 1;
         z := 1;
      };
      if(x<y){
         w := 2;
         z := 4;
      }else{
         w := 5;
         z := 5;
      };
      print(x);
      print(y);
      print(z);
      print(w);
   }.
```

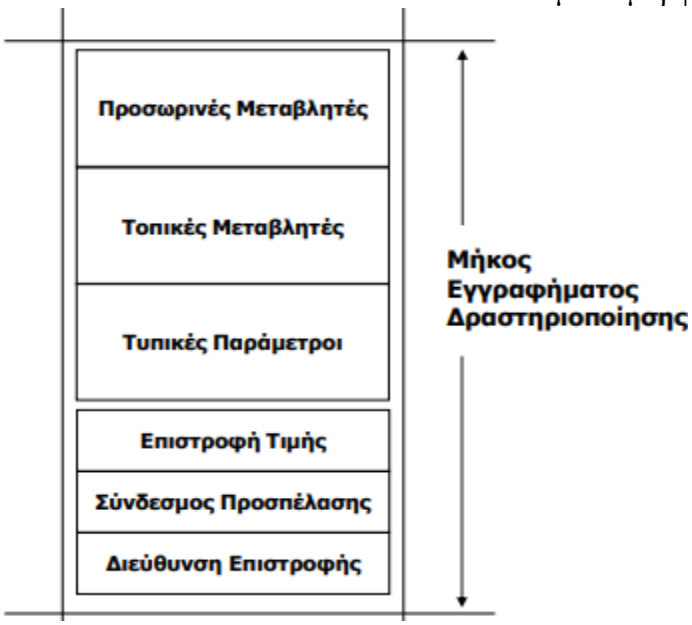
Αρχείο multipleifs.c :

```c
#include <stdio.h>  // for the printf and scanf functions


int main()
{
   int x,y,z,w;
   //(['begin_block', 'multipleifs', '_', '_'])
   L_1: scanf("%d",&x); //(['inp', 'x', '_', '_'])
   L_2: scanf("%d",&y); //(['inp', 'y', '_', '_'])
   L_3: if (x > 0) goto L_5; //(['>', 'x', '0', 5])
   L_4: goto L_8; //(['jump', '_', '_', 8])
   L_5: w = 0; //([':=', '0', '_', 'w'])
   L_6: z = 0; //([':=', '0', '_', 'z'])
   L_7: goto L_10; //(['jump', '_', '_', 10])
   L_8: w = 1; //([':=', '1', '_', 'w'])
   L_9: z = 1; //([':=', '1', '_', 'z'])
   L_10: if (x < y) goto L_12; //(['<', 'x', 'y', 12])
   L_11: goto L_15; //(['jump', '_', '_', 15])
   L_12: w = 2; //([':=', '2', '_', 'w'])
   L_13: z = 4; //([':=', '4', '_', 'z'])
   L_14: goto L_17; //(['jump', '_', '_', 17])
   L_15: w = 5; //([':=', '5', '_', 'w'])
   L_16: z = 5; //([':=', '5', '_', 'z'])
   L_17: printf("%d",x); //(['out', 'x', '_', '_'])
   L_18: printf("%d",y); //(['out', 'y', '_', '_'])
   L_19: printf("%d",z); //(['out', 'z', '_', '_'])
   L_20: printf("%d",w); //(['out', 'w', '_', '_'])
   L_21: return 0; //(['halt', '_', '_', '_'])
   //(['end_block', 'multipleifs', '_', '_'])
}
```

5)Δημιουργία Πίνακα Συμβόλων

5α) Θεωρία

Ο πίνακας συμβόλων αποτελεί την "γέφυρα" μεταξύ του ενδιάμεσου και τελικού κώδικα. Ως εγγραφές του περιέχει όλες τις χρήσιμες πληροφορίες που αφορούν τα υποπρογράμματα, τις μεταβλητές, και τις παραμέτρους. Πριν αναλύσουμε τα περιεχόμενα του, είναι σημαντικό να αναφερθούμε στο **εγγράφημα δραστηριοποίησης.** Δημιουργείται με κάθε συνάρτηση ή διαδικασία όταν καλείται και είναι μια μορφή στοίβας όπως φαίνεται παρακάτω:



(πηγή : διαφάνειες διδάσκοντα αντίστοιχης διάλεξης)
Τα τρία κάτω είναι κοινά για όλες τις οντότητες του πίνακα συμβόλων και για αυτό το λόγο η απόσταση από την αρχή του εγγραφήματος θα αρχικοποιείται στην τιμή 12. Η γλώσσα assembly κάθε φορά που θέλουμε να μεταβούμε στην επόμενη διεύθυνση, μεταβαίνει ανά 4 bytes, επομένως εφόσον έχουμε 3 στοιχεία: 4 * 3 = 12.

Οι οντότητες του πίνακα χωρίζονται σε 3 κατηγορίες, Scope, Entity και Argument,οι οποίες θα εξηγηθούν στην ανάλυση του κώδικα μαζί με την μορφή του πίνακα.

5.β) Ανάλυση αντίστοιχου κώδικα

Για τα στοιχεία (records) του πίνακα συμβόλων χρησιμοποιούμε την μορφή κλάσεων της Python, επειδή θέλουμε να αποθηκεύσουμε διάφορες τιμές και συμβολοσειρές στο εσωτερικό τους. Επομένως:

Record Entities: Όλα τα entities έχουν ως πρώτα δύο ορίσματα στη κάθε __init__ της κλάσης το name (όνομα) και το entity (τύπος) της:

- **Variable:** Οι μεταβλητές, έχουν ως έξτρα όρισμα το offset, δηλαδή την απόσταση από την κορυφή της στοίβας του scope στο οποίο βρισκόμαστε. Κάθε φορά που εμφανίζεται νέο entity αυξάνεται ανά 4.
- **Function/Procedure:** Τα υποπρογράμματα, ενώ είναι δύο διαφορετικές κλάσεις, έχουν την ίδια σύνταξη με την μόνη διαφορά, την συμβολοσειρά αποθηκευμένη στον τύπο τους.

Έχουν ως επιπλέον ορίσματα το start_quad, που συμβολίζει την τετράδα του quad_list στην οποία γίνεται το begin_block, δηλαδή η έναρξη του ορισμού τους. Επίσης έχουν την argument_list, η οποία περιέχει την μορφή παραμέτρων που δέχονται καθώς και το frame_length, το μήκος του εγγραφήματος δραστηριοποίησης, αρχικοποιημένο στην τιμή 12.

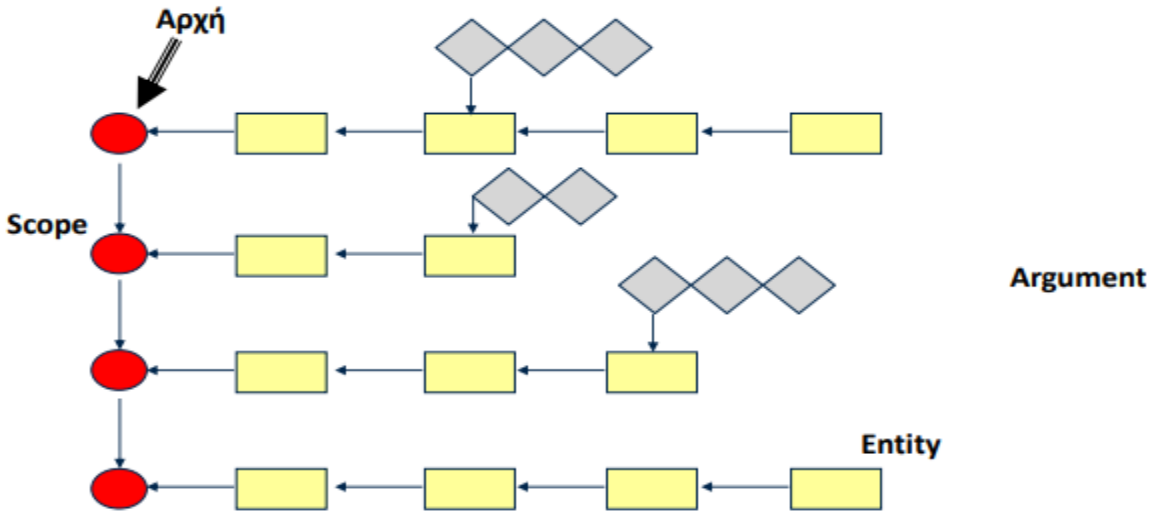
- **Parameter:** Η παράμετρος ενός υποπρογράμματος. Ομοίως με τις μεταβλητές, έχει επιπλέον όρισμα το offset. Επιπρόσθετα περιέχει το parmode, το οποίο είναι ο τρόπος περάσματος (με τιμή – "in" ή με αναφορά – "inout").
- **TempVariable:** Οι προσωρινές μεταβλητές, έχουν ίδια σύνταξη με την κλάση των μεταβλητών.
  (σύμβολο σχήματος: ορθογώνιο παραλληλόγραμμο)

Record Scope: Το περιεχόμενο του πίνακα συμβόλων στο οποίο "κουμπώνουν" τα entities. Για αυτό το λόγο περιέχει ως όρισμα το entity_list, μια λίστα με όλα τα entities του. Κάθε φορά που δημιουργούμε μια συνάρτηση/διαδικασία, δημιουργείται ένα νέο scope αυξάνοντας το nesting_level (δηλαδή το βάθος φωλιάσματος κατά μια μονάδα. Επιπλέον περιέχει το frame_length αρχικοποιημένο στη τιμή 12 για να το μεταφέρει με κατάλληλο τρόπο στα υποπρογράμματα, αν υπάρχουν. Πρέπει να σημειωθεί πως αρκετά συχνά για τον χειρισμό του πίνακα συμβόλων, χρησιμοποιούμε το symbol_list[-1] , δηλαδή το πιο πρόσφατο scope που έχει εισαχθεί στον πίνακα. (σύμβολο σχήματος: κύκλος)

Record Argument: Όπως το entity parameter, περιέχει το όρισμα parmode και εισάγεται στο argument_list των υποπρογραμμάτων. (σύμβολο σχήματος: ρόμβος)

Η μορφή του πίνακα σχηματικά φαίνεται παρακάτω: (το σχήμα είναι από τις διαφάνειες του διδάσκοντα))

Μορφή του Πίνακα Συμβόλων



Το αρχικό scope, με βάθος φωλιάσματος 0 δημιουργείται μέσω της add_scope() στην syntax() και είναι υπεύθυνο για τα entities του κύριου προγράμματος.

Οι βοηθητικές συναρτήσεις τις τωρινής φάσης είναι οι παρακάτω:

- **add_scope(nest_level):** Λαμβάνει ως όρισμα το βάθος φωλιάσματος, δημιουργεί ένα νέο αντικείμενο κλάσης Scope το οποίο επιστρέφει εφόσον ενημερώσει το αντίστοιχο του nesting_level και το προσθέσει στον πίνακα συμβόλων.
- **close_scope():** Διαγράφει το τελευταίο scope που δημιουργήθηκε από τον πίνακα συμβόλων. Καλείται από την block() κάθε φορά που τελειώνει ο ορισμός μιας συνάρτησης, ώστε στη τελική μορφή του πίνακα, να παραμείνει μόνο το scope του κύριου προγράμματος συμπληρωμένο με όλες τις απαραίτητες πληροφορίες.
- **add_entity(ent_type, name, offset, parmode, ent_list):** Δημιουργεί ένα αντικείμενο της entity κλάσης που έχει δοθεί ως ent_type , χρησιμοποιώντας κατάλληλα τα υπόλοιπα ορίσματα.
  Χρησιμοποιείται στα παρακάτω σημεία του κώδικα, όταν:
  - Δημιουργείται μια προσωρινή μεταβλητή, στην newtemp().
  - Αναγνωρίζεται το αναγνωστικό κάποιας μεταβλητής μετά την δεσμευμένη “declare”, στην varlist().
  - Αναγνωρίζεται το αναγνωριστικό του ονόματος ενός υποπρογράμματος, στην subprogram() .
  - Ορίζεται κάποια παράμετρος υποπρογράμματος στην formalparitem().

- **add_argument(parmode, argument_list):** Δημιουργεί ένα αντικείμενο Argument και το επιστρέφει αφού το εισάγει στο argument_list. Καλείται στην formalparitem() όταν αναγνωρίζουμε το “in” ή “inout” που λαμβάνει ως parmode. Επιπλέον με τη χρήση της search_entity(func_name) που θα αναλύσουμε στη συνέχεια, αναγνωρίζει το argument_list του υποπρογράμματος στο οποίο πρέπει να προστεθεί.
- **search_entity(name):** Ανατρέχει όλα τα scope του πίνακα συμβόλων και ελέγχει τα ονόματα των entities που έχει κάθε entity_list. Αν βρεθεί κάποιο entity.name που είναι ίσο με το name που δέχεται ως όρισμα, επιστρέφουμε το αντίστοιχο entity. Επιπρόσθετα γίνεται όμοιος έλεγχος στην λίστα Temp_vars, η οποία είναι μια λίστα με όλα τα entities προσωρινών μεταβλητών. Αυτό ο τελευταίος έλεγχος προστέθηκε, καθώς κατά την διαδικασία αποσφαλμάτωσης, ανακαλύψαμε πως χάνεται η πληροφορία των entities αυτών, αν γίνει close_scope() του scope που τα δημιούργησε. Στη περίπτωση που δεν εντοπίσουμε το entity με το ζητούμενο όνομα, εκτυπώνουμε μήνυμα σφάλματος.

5.γ) Παραγωγή αρχείου .txt

Το αρχείο txt_file δημιουργείται στη συνάρτηση program() χρησιμοποιώντας το program_name και την κατάληξη “.txt”. Χρησιμοποιώντας τις display συναρτήσεις (display_symbol_list(), display_scope(scope), display_entity(entity,ent_type), display_argument(arg)) καταγράφουμε τον πίνακα συμβόλων. Πιο συγκεκριμένα, η **display_symbol_list()** εκτελείται στο block() ώστε να καταγράψει κάθε φορά που έχει οριστεί ένα υποπρόγραμμα τα περιεχόμενα του στο txt_file. Η

ίδια για κάθε scope του πίνακα συμβόλων καλεί την **display_scope(scope)**, η οποία επιστρέφει όλα τα περιεχόμενά του scope. Για το κάθε scope αρχικά καταγράφεται το βάθος φωλιάσματος και στη συνέχεια όλα τα entities του (αν υπάρχουν) με την κλήση της **display_entity(entity,ent_type)** και τέλος το αντίστοιχο frame_length.

Η display_entity() αναλύει κάθε περίπτωση entity και επιστρέφει μια συμβολοσειρά με τα περιεχόμενα τους.

Στην περίπτωση υποπρογραμμάτων, καλείται επαναληπτικά η **display_argument(arg),** η οποία επιστρέφει τη μορφή των παραμέτρων που ορίζουν μια συνάρτηση ή διαδικασία. Κάθε φορά που διαγράφεται κάποιο scope και ανανεώνεται ο πίνακας συμβόλων, καταγράφεται αντίστοιχο μήνυμα στο αρχείο.

5.δ) Παράδειγμα εκτέλεσης

Για το παρακάτω πρόγραμμα:

```
program power
   declare x;
   declare y;
   declare result;
        function second(in x){
            while (x<100){
                x := x * x;
            };
            return (x);
        }

        function third (in x){
            if (x<50){
                x := second(in x) * x;
            };
            return (x);
        }

        function equal(in x, in y){
            if (y = x){
                result := 0;
            }else{
                result := 5;
            };
            return (result);
        }
        # main #
        {
        input(x);
        x := third(in x);
        input(y);
        result := equal (in x, in y);
        print(result);
        }.
```

Ο αντίστοιχος πίνακας συμβόλων είναι ο παρακάτω:

```
Nesting level: 0
Entities:
Var: x / 12
Var: y / 16
Var: result / 20
Function: second | Starting quad : 1 | Arguments: (in) | Frame_length: 24
Scope's framelength: 24
Nesting level: 1
Entities:
Par: x / in / 12
Temp_Var: T_0 / 16
Scope's framelength: 20
--------------------------------------------------------------------------------
Modified Symbol Array:
Nesting level: 0
Entities:
Var: x / 12
Var: y / 16
Var: result / 20
Function: second | Starting quad : 1 | Arguments: (in) | Frame_length: 24
Function: third | Starting quad : 9 | Arguments: (in) | Frame_length: 28
Scope's framelength: 24
Nesting level: 1
Entities:
Par: x / in / 12
Temp_Var: T_1 / 16
Temp_Var: T_2 / 20
Scope's framelength: 24
--------------------------------------------------------------------------------
Modified Symbol Array:
Nesting level: 0
Entities:
Var: x / 12
Var: y / 16
Var: result / 20
Function: second | Starting quad : 1 | Arguments: (in) | Frame_length: 24
Function: third | Starting quad : 9 | Arguments: (in) | Frame_length: 28
Function: equal | Starting quad : 20 | Arguments: (in,in) | Frame_length: 24
Scope's framelength: 24
Nesting level: 1
Entities:
Par: x / in / 12
Par: y / in / 16
Scope's framelength: 20
--------------------------------------------------------------------------------
Modified Symbol Array:
Nesting level: 0
Entities:
Var: x / 12
Var: y / 16
Var: result / 20
Function: second | Starting quad : 1 | Arguments: (in) | Frame_length: 24
Function: third | Starting quad : 9 | Arguments: (in) | Frame_length: 28
Function: equal | Starting quad : 20 | Arguments: (in,in) | Frame_length: 24
Temp_Var: T_3 / 24
Temp_Var: T_4 / 28
Scope's framelength: 32
```

6)Σημασιολογική Ανάλυση

Για το στάδιο αυτό έπρεπε να τροποποιήσουμε κατάλληλα τον κώδικα για να πραγματοποιήσουμε τις ακόλουθες οδηγίες του διδάσκοντα:

```
α) κάθε συνάρτηση έχει μέσα της τουλάχιστον ένα return.
```

Για την αντιμετώπιση αυτής της συνθήκης, χρησιμοποιούμε την καθολική boolean μεταβλητή **return_flag**. Συγκεκριμένα, αρχικοποιείται στην false τιμή και κάθε φορά που αναγνωρίζεται η δεσμευμένη λέξη "return" στην return_stat() την θέτουμε σε τιμή true. Ο έλεγχος της τιμής της γίνεται στην subprogram(), εφόσον έχει ολοκληρωθεί η ανάλυση κάποιας συνάρτησης. Αν η τιμή παραμένει false, εκτυπώνεται το αντίστοιχο μήνυμα σφάλματος. Στην αντίθετη περίπτωση, την θέτουμε ξανά στην τιμή false, ώστε να συνεχιστεί ο έλεγχος και στις επόμενες συναρτήσεις, αν υπάρχουν

```
β) δεν υπάρχει return έξω από συνάρτηση.
γ) μέσα σε μία διαδικασία δεν επιτρέπεται να υπάρχει return.
```

Θα μπορούσαμε να μεταφράσουμε αυτές τις 2 οδηγίες ως εξής: "Αν εντοπιστεί η δεσμευμένη λέξη return κατά την ανάλυση διαδικασίας ή του κυρίου προγράμματος, εκτυπώνουμε μήνυμα σφάλματος". Επομένως για αυτόν τον σκοπό, χρησιμοποιούμε τις boolean μεταβλητές **main_state_flag** και **procedure_state_flag,** καθώς και την μεταβλητή **temp_flag.** Οι boolean μεταβλητές αρχικοποιούνται στην τιμή false, ενώ η temp_flag στην τιμή 0. Η main_state_flag, γίνεται true στο block(), όταν βρισκόμαστε στη συνθήκη name == program_name, ενώ η procedure_state_flag γίνεται true όταν αναγνωριστεί η λέξη procedure. Αν εμφανιστεί η λέξη return στο return_flag() και κάποια από τα παραπάνω flags είναι true, τότε εκτυπώνουμε μήνυμα σφάλματος.
Υπάρχει όμως η περίπτωση μια ή παραπάνω συναρτήσεις να ορίζονται εσωτερικά σε μια διαδικασία και για αυτό το λόγο θα χρησιμοποιήσουμε την μεταβλητή temp_flag. Συγκεκριμένα, αν εμφανιστεί ο ορισμός μιας συνάρτησης και η procedure_state_flag είναι true, αντιλαμβανόμαστε πως η συνάρτηση ορίζεται μέσα σε διαδικασία. Για κάθε τέτοια συνάρτηση αυξάνεται ο αριθμός της temp_flag κατά ένα, ενώ η procedure_state_flag γίνεται false. Εφόσον τελειώσει η ανάλυση της συνάρτησης γίνεται έλεγχος της temp_flag. Αν είναι μεγαλύτερη του 0, σημαίνει πως έχει υπάρξει τουλάχιστον μια συνάρτηση στο εσωτερικό διαδικασίας οπότε αφαιρούμε κατά ένα την τιμή της temp_flag. Στην περίπτωση που η temp_flag είναι 0, σημαίνει πως δεν υπάρχουν άλλες συναρτήσεις, άρα ξαναθέτουμε την procedure_state_flag σε τιμή true για τον υπόλοιπο έλεγχο.

```
δ) κάθε μεταβλητή ή συνάρτηση ή διαδικασία που έχει δηλωθεί να μην έχει δηλωθεί
πάνω από μία φορά στο βάθος φωλιάσματος στο οποίο βρίσκεται.
```

Ο έλεγχος της μεταβλητής έχει ήδη αναλυθεί με την χρήση της καθολικής λίστας variables. Για τα υποπρογράμματα, εκτελούμε τον ίδιο έλεγχο και στις δύο περιπτώσεις στην subprogram(). Συγκεκριμένα ανατρέχουμε την entity_list του τρέχοντος scope. Αν εντοπίσουμε πως κάποιο όνομα ενός entity ισούται με το όνομα του υποπρογράμματος, τότε εντοπίσαμε το λάθος και εκτυπώνουμε κατάλληλο μήνυμα.

ε) κάθε μεταβλητή, συνάρτηση ή διαδικασία που χρησιμοποιείται έχει δηλωθεί και μάλιστα με τον τρόπο που χρησιμοποιείται (σαν μεταβλητή ή σαν συνάρτηση).

Ο έλεγχος αυτός γίνεται καθ' όλη την διάρκεια με την χρήση των λιστών variables, procedures_list, function_list στις συναρτήσεις:

- varlist() ,assignment_stat() και input_stat() για τις μεταβλητές.
- subprogram() και formalparitem() για τα υποπρογράμματα.
- call_stat() συγκεκριμένα για τις διαδικασίες.

στ) οι παράμετροι με τις οποίες καλούνται οι συναρτήσεις είναι ακριβώς αυτές με τις οποίες έχουν δηλωθεί και με τη σωστή σειρά.

Για την παραπάνω οδηγία, δημιουργήθηκε η συνάρτηση **param_check(func_name, par_list).** Δέχεται ως όρισμα, το func_name, το όνομα του υποπρογράμματος και το par_list, τη λίστα με τις παραμέτρους που θέλουμε να ελέγξουμε πως είναι σωστή. Συγκεκριμένα, χρησιμοποιείται η search_entity(func_name) για να εντοπίσουμε την entity που καλούμε. Στην τοπική λίστα func_pars αποθηκεύεται το argument_list του entity-υποπρογράμματος και στη συνέχεια ελέγχεται ως προς το μήκος της σε σύγκριση με την par_list που δέχεται ως όρισμα. Αν το μήκος των δύο λιστών δεν είναι ίσο, τότε εκτυπώνουμε κατάλληλα μηνύματα σφάλματος, αλλιώς ελέγχουμε αν είναι της ίδιας μορφής. Για παράδειγμα οι λίστες ['in', 'in', 'inout'] και ['in', 'in', 'in'] έχουν το ίδιο μήκος αλλά δεν είναι έχουν τα ίδια στοιχεία, επομένως ενημερώνουμε τον χρήστη με το αντίστοιχο μήνυμα σφάλματος για να κάνει διορθώσεις.
Εκτελείται στις συναρτήσεις call_stat() και idtail() για τον έλεγχο των αντίστοιχων διαδικασιών και συναρτήσεων αντίστοιχα.

7) Παραγωγή τελικού κώδικα

7.α) Θεωρία

Στο τελική φάση της μεταγλώττισης, θα δημιουργήσουμε τον τελικό κώδικα, δηλαδή θα μετατρέψουμε τις τετράδες που δημιουργήσαμε στη φάση του ενδιάμεσου κώδικα σε κώδικα assembly, επεξεργαστή MIPS.

Πριν αναλύσουμε την διαδικασία μετατροπής, αξίζει να αναφερθούμε συνοπτικά στις πληροφορίες που χρειαζόμαστε να γνωρίζουμε για την συγκεκριμένη μορφή της γλώσσας assembly.
Η αρχιτεκτονική του MIPS βασίζεται σε καταχωρητές, οι οποίοι έχουν το σύμβολο "$" στην αρχή του ονόματος τους, περιέχουν τιμή και διεύθυνση στην μνήμη. Ανάλογα την χρήση τους, μπορεί να είναι καταχωρητές:

- προσωρινών τιμών ($t0, $t1,…, $t7)
- που διατηρούνται ανάμεσα κλήσεων υποπρογραμμάτων ($s0,$s1,..., $s7)
- ορισμάτων ($a0, $a1, $a2, $a3)
- τιμών ($v0, $v1)
- δείκτη στοίβας σε κάθε εγγράφημα δραστηριοποίησης ($sp)
- frame pointer ($fp) ο οποίος δείχνει στη στοίβα των υποπρογραμμάτων που θα δημιουργηθούν.
- που αποθηκεύει την διεύθυνση επιστροφής ($ra)

Μέσω των εντολών add,sub,div,mul εκτελούμε τις αριθμητικές πράξεις της πρόσθεσης, αφαίρεσης, διαίρεσης και πολλαπλασιασμού αντίστοιχα.

Όσο αφορά την μετακίνηση δεδομένων έχουμε τις παρακάτω εντολές (όπου καταχ. = καταχωρητης) :

- move καταχ. 1, καταχ. 2 : ο καταχωρητής 1 αποκτάει την πληροφορία του 2ου.
- li κατάχ. σταθερά : ο καταχωρητής, αποκτάει την τιμή της σταθεράς.
- lw καταχ. διεύθυνση: ο καταχωρητής, αποκτάει ως τιμή νέας διεύθυνσης την τιμή της του 2ου ορίσματος.
- sw καταχ. διεύθυνση: το αντίστροφο της lw, δίνεται το περιεχόμενο του καταχωρητή στην συγκεκριμένη διεύθυνση της μνήμης.
- Στην περίπτωση των lw και sw αν δίνεται κάποιος καταχωρητής μέσα σε [] ως 2ο όρισμα, σημαίνει πως γίνεται έμμεση αναφορά στην διεύθυνση του καταχωρητή

Ιδιαίτερα σημαντική είναι η λειτουργία των αλμάτων για τις διάφορες εντολές συνθήκης, για την πραγματοποίηση επανάληψης και την κλήση συνάρτησης.

Για τις εντολές συνθήκης (αντίστοιχες if των περισσότερων γλωσσών προγραμματισμού) χρησιμοποιούμε τις εξής:

- beq καταχ. 1 , καταχ. 2, ετικέτα (αν ο 1ος καταχωρητής είναι **ίσος** με τον 2ο ,μετάβαση στην ετικέτα)
- blt καταχ. 1 , καταχ. 2, ετικέτα (αντίστοιχα αν ο 1ος καταχωρητής είναι **μικρότερος**)
- bgt καταχ. 1 , καταχ. 2, ετικέτα (αντίστοιχα αν ο 1ος καταχωρητής είναι **μεγαλύτερος**)
- ble καταχ. 1 , καταχ. 2, ετικέτα (αντίστοιχα αν ο 1ος καταχωρητής είναι **μικρότερος ίσος**)
- bge καταχ. 1 , καταχ. 2, ετικέτα (αντίστοιχα αν ο 1ος καταχωρητής είναι **μεγαλύτερος ίσος**)
- bne καταχ. 1 , καταχ. 2, ετικέτα (αντίστοιχα αν ο 1ος καταχωρητής είναι **διαφορετικός**)

Για την απλή μετάβαση σε ετικέτα: j label

Για την κλήση συνάρτησης: jal label

Για μετάβαση σε διεύθυνση καταχωρητή: jr καταχωρητή

7.β) Ανάλυση αντίστοιχου κώδικα

Για να πραγματοποιήσουμε την αντιστοιχία του ενδιάμεσου κώδικα με τον τελικό θα πρέπει να χρησιμοποιήσουμε ορισμένες βοηθητικές συναρτήσεις :

- **search_level(entity):** Δέχεται ως όρισμα ένα entity και το αναζητάει, ανατρέχοντας των πίνακα συμβόλων. Αν βρεθεί, επιστρέφει το βάθος φωλιάσματος στο οποίο βρίσκεται. Είναι ιδιαίτερα σημαντική για τις υπόλοιπες συναρτήσεις του τελικού κώδικα, καθώς πολλές φορές θα πρέπει να γνωρίζουμε σε ποιο βάθος φωλιάσματος είναι κάποια μεταβλητή ή υποπρόγραμμα.

(Σημείωση: Οι μορφές των επόμενων βοηθητικών συναρτήσεων προκύπτουν από τις οδηγίες που δόθηκαν στις αντίστοιχες διαφάνειες του καθηγητή.)

- **gnvlcode(var):** Η συνάρτηση αυτή μεταφέρει στον καταχωρητή $t0 την διεύθυνση της μη τοπικής μεταβλητής var. Αρχικά αποθηκεύουμε στην current_level το τρέχον επίπεδο φωλιάσματος και στην var_level το βάθος της μεταβλητής που δίνεται ως όρισμα. Αυτό το κάνουμε ώστε να την διαφορά των 2 βαθών για να καταγράψουμε τόσες φορές την lw $t0, -4($t0), εφόσον καταγράψουμε την lw $t0, -4($sp). Στο τέλος καταγράφουμε την εντολή addi $t0, $t0, - < το offset του entity της μεταβλητής var >.

- **loadvr(v, r):** Για την μεταφορά δεδομένων από την μνήμη στον καταχωρητή r ανάλογα τον τύπο του v. Συγκεκριμένα:
  - Αν το v είναι κάποια σταθερά, την φορτώνουμε στην r με την καταγραφή της li r,v. Σε αυτή την περίπτωση, κατά τη διάρκεια ελέγχου του κώδικα, παρατηρήσαμε πως δεν αναγνώριζε τους αρνητικούς αριθμούς. Για αυτό τον λόγο, αρχικά ελέγχουμε και αφαιρούμε το σύμβολο "-" ώστε να μπορέσουμε να περάσουμε στους υπόλοιπους ελέγχους της συνάρτησης. Ταυτόχρονα "ενεργοποιείται" το negative_flag, ώστε να μπορέσουμε αντίστοιχα να δώσουμε σημειώσουμε το σύμβολο στην καταγραφή.
  - Αν το v είναι κάποια καθολική μεταβλητή, δηλαδή ορίζεται στο κύριο πρόγραμμα, στο scope του μηδενικού βάθους φωλιάσματος, καταγράφουμε την εντολή lw r, -offset, όπου το offset, είναι του v.
  - Αν το v είναι μια μεταβλητή εκτός του κύριου προγράμματος, προσωρινή μεταβλητή, ή παράμετρος που περνάει με τιμή, καταγράφουμε τις εξής εντολές, ανάλογα την σχέση του βάθους φωλιάσματος της v και του τωρινού:
    Αν είναι ίσο: lw r, -offset($sp) .
    Αν το τρέχον είναι μεγαλύτερο, δηλαδή έχει δηλωθεί σε κάποιο πρόγονο της: lw r, ($t0) , εφόσον έχουμε καλέσει όμως την gnvlcode(v) ώστε να τοποθετηθεί στην η μεταβλητή $t0 στην διεύθυνση της v.
  - Αν το v είναι μια παράμετρος που περνάει με αναφορά, ομοίως με πριν με έλεγχο του βάθους φωλιάσματος:
    Αν είναι ίσο: lw $t0, -offset($sp) και lw r, ($t0) .
    Αν έχει δηλωθεί σε κάποιο πρόγονο της: gnvlcode(v) , lw $t0, ($t0) και lw r, ($t0).
  - Κατά τη διάρκεια δημιουργίας του κώδικα, εμφανίστηκε η περίπτωση το v να είναι συνάρτηση, αλλά δεν υπήρχαν κατατοπιστικές οδηγίες για την περίπτωση αυτή.

- **storerv(r, v):** Θυμίζει την loadvr, αλλά αυτή την φορά μεταφέρουμε δεδομένα από τον καταχωρητή r στη μεταβλητή v. Ομοίως με πριν, ανάλογα με τον τύπο του v, καταγράφουμε διαφορετικές εντολές. Συγκεκριμένα:
  - Αν το v είναι κάποια καθολική μεταβλητή, καταγράφουμε την εντολή sw r, -offset, όπου το offset, είναι του v.
  - Αν το v είναι μια μεταβλητή εκτός του κύριου προγράμματος, προσωρινή μεταβλητή, ή παράμετρος που περνάει με τιμή, καταγράφουμε τις εξής εντολές, ανάλογα την σχέση του βάθους φωλιάσματος της v και του τωρινού:

Αν είναι ίσο: sw r, -offset($sp) .
Αν έχει δηλωθεί σε κάποιο πρόγονο της: gnvlcode(v) , sw r, ($t0) .

- Αν το v είναι μια παράμετρος που περνάει με αναφορά, ομοίως με πριν με έλεγχο του βάθους φωλιάσματος:
  Αν είναι ίσο: lw $t0, -offset($sp) και sw r, ($t0) .
  Αν το τρέχον είναι μεγαλύτερο: gnvlcode(v) , lw $t0, ($t0) και sw r, ($t0).

Οι συναρτήσεις που επεξεργάζονται το "program_name".asm αρχείο είναι η asm_file_create(start_quad, name, current) καθώς και το final_code(name, current, start_quad, end_quad) η οποία την καλεί κατάλληλα.

asm_file_create: Αρχικά από τα ορίσματα, πρώτο διακρίνουμε το start_quad, το οποίο είναι το νούμερο της θέσης της εναρκτήρια τετράδα του υποπρογράμματος ή του κύριου μέρους στην πρώτη εκτέλεσης της συνάρτησης. Χρησιμοποιώντας την, αποθηκεύουμε στη προσωρινή λίστα quad, κάθε φορά την τρέχουσα τετράδα που θα αναλύσουμε και στη συνέχεια στο current_level αποθηκεύουμε το τρέχον βάθος φωλιάσματος. Ως $2^{ο}$ όρισμα έχουμε το name, το οποίο είναι το όνομα του υποπρογράμματος στην περίπτωση που δεν είμαστε στο κυρίως πρόγραμμα. Αν έχουμε κάποια συνάρτηση ή διαδικασία, με την search_entity και search_level, αποθηκεύουμε προσωρινά όλες τις πληροφορίες που χρειαζόμαστε στις μεταβλητές func και func_level αντίστοιχα. Για να αντιστοιχίσουμε τις τετράδες σε αντίστοιχο κώδικα assembly, κύριος γνώμονας είναι το πρώτο στοιχείο κάθε τετράδας, επομένως αν το $1^{ο}$ στοιχείο (quad[0]) είναι:

- **"begin_block":** Στην περίπτωση του κύριου προγράμματος, σημειώνουμε αρχικά την ταμπέλα Lmain, ώστε να αναγνωρίσουμε, τι ανάλυση κάνουμε και στη συνέχεια προσθέτουμε τις εντολές: add $sp, $sp, < το framelength του scope στο μηδενικό βάθος φωλιάσματος> και move $s0, $sp.
  Στην περίπτωση όμως υποπρογράμματος, δημιουργούμε την ετικέτα L_ <τωρινό start_quad – 1> και sw $ra ($sp)

  (Σημείωση: Για τις παρακάτω περιπτώσεις, πρώτο πράγμα που πρέπει να κάνουμε, είναι η καταγραφή της ετικέτας L_ <start_quad – 1>
- **"jump":** Όπως εξηγήθηκε προηγουμένως, η jump είναι η εντολή άλματος σε ταμπέλα. Επομένως, στη περίπτωση που μεταβαίνουμε στην start_quad που ανταποκρίνεται σε αυτή του κυρίου προγράμματος, καταγράφουμε j L_main ενώ αλλιώς : j L_ <$4^{η}$ παράμετρος της τετράδας>. Για να πραγματοποιήσουμε τον έλεγχο αυτό, χρησιμοποιούμε την καθολική main_start_quad, η οποία αποθηκεύει την εναρκτήρια τετράδα της main, όταν γίνεται η ανάλυση της στην block().

- Αν ανήκει στα σύμβολα **σύγκρισης:** Για αρχή εκτελούμε την συνάρτηση loadvr() δύο φορές. Μία για την $2^{η}$ παράμετρο της τετράδας ως το v και τον καταχωρητή $t1 ως το r και την επόμενη με την $3^{η}$ παράμετρο και τον $t2 αντίστοιχα. Στη συνέχεια, καταγράφουμε αρχικά τις αντίστοιχες συναρτήσεις σύγκρισης που αναλύσαμε προηγουμένως ακολουθούμενες με τα στοιχεία $t1, $t2, L_ <$4^{η}$ παράμετρος της τετράδας>.

- Όμοια διαδικασία ακολουθούμε και για τις **αριθμητικές** πράξεις, , εκτελώντας με τον ίδιο τρόπο την loadvr() και καταγράφοντας τις συναρτήσεις αριθμητικών πράξεων ακολουθούμενες των στοιχείων $t1, $t1, $t2 στις θέσεις των καταχωρητών.

- Το σύμβολο **ανάθεσης ":=":** Εκτελούμε την loadvr με την $2^η$ παράμετρο της τετράδας ως το v και τον καταχωρητή $t1 ως το r, καθώς η ανάθεση γίνεται στον $t1. Μετά την κλήση της, καλούμε και την storerv με τον $t1 ως r και το $4^ο$ στοιχείο της τετράδας ως v.

- **"out":** Για την περίπτωση εξόδου, θα πρέπει να καταγράψουμε την εντολή li $v0, 1 και εφόσον εκτελέσουμε την loadvr με το $2^ο$ στοιχείο της τετράδας και τον "$a0" ως παραμέτρους, καταγράφουμε την εντολή syscall ώστε να καταγράψει στο τερματικό.

- **"inp":** Για την περίπτωση εισόδου, αρχικά καταγράφουμε την εντολή li $v0, 1 και την εντολή syscall . Στη συνέχεια εκτελούμε την storerv με $v0 και $2^ο$ στοιχείο της τετράδας ως παραμέτρους. Επομένως όταν εκτελούμε τον κώδικα, μας ζητείται από το τερματικό να δώσουμε κάποια ακέραια τιμή.

- **"retv":** Για την επιστροφή τιμής, αρχικά εκτελούμε την loadvr με το $2^ο$ στοιχείο της τετράδας και τον "$t1 " ως παραμέτρους. Στη συνέχεια, καταγράφουμε την εντολή move $v0, $t1 για να αποθηκεύσουμε την τιμή επιστροφής του προσωρινού καταχωρητή στον καταχωρητή τιμής $v0.

- **"par":** Η διαδικασία συμπλήρωσης παραμέτρων υποπρογράμματος είναι ανάλογη του τύπου μετάδοσης της και του βάθους φωλιάσματος της συνάρτησης/διαδικασίας και της καλούσας . Επίσης σε αυτό το σημείο, χρειαζόμαστε την καθολική μεταβλητή i_num, η οποία αυξάνεται κάθε φορά που εμφανίζεται μια παράμετρος, και θα χρησιμοποιηθεί στη συνέχεια. Επομένως, ανεξαρτήτως του τύπου του, αν προσθέτουμε στο πρόγραμμα μας παράμετρο για πρώτη φορά (δηλαδή το i_num είναι ίσο με το 0) τότε καταγράφουμε την εντολή addi $fp, $sp framelength, όπου framelength το frame_length του υποπρογράμματος που χρησιμοποιεί την παράμετρο. Στη συνέχεια ελέγχουμε τον τύπο της παραμέτρου μέσω του $3^ου$ ορίσματος της τετράδας:
  - Αν το τρίτο όρισμα είναι **"CV":** εκτελούμε την loadvr με ορίσματα το $2^ο$ στοιχείο της τετράδας και τον $t0 και καταγράφουμε την εντολή
    sw $t0, -(12 + 4 * i_num) ($fp)
  - Αν το τρίτο όρισμα είναι **"REF"** το $2^ο$ στοιχείο της τετράδας είναι μεταβλητή, προσωρινή μεταβλητή ή παράμετρος με τρόπο μετάδοσης με τιμή **και:**
    Το βάθος φωλιάσματος είναι ίσο με το τρέχον καταγράφουμε addi $t0, $sp, -offset όπου offset το offset του $2^ου$ στοιχείου.
    Αλλιώς εκτελούμε την gnvlcode με το $2^ο$ στοιχείο της τετράδας ως όρισμα.
  - Αντίστοιχα αν ο τρόπος μετάδοσης είναι με αναφορά για το ίδιο βάθος καταγράφουμε: lw $t0, -offset
    Αλλιώς: lw $t0, ($t0)

Στο τέλος όλων των "REF" περιπτώσεων, καταγράφουμε την εντολή:
sw $t0, -(12 + 4 * i_num) ($fp)

- Αν το τρίτο όρισμα τετράδας είναι **"RET",** καταγράφουμε τις εντολές:
  addi $t0, $sp, -offset
  sw $t0, -8($fp)

- **"call":** Στη περίπτωση της κλήσης συνάρτησης, η πρώτη μας ενέργεια είναι να βρούμε τα στοιχεία της κληθείσας συνάρτησης/διαδικασίας, αποθηκεύοντας στα στις αντίστοιχες τοπικές μεταβλητές f_entity, f_level, f_framelength και f_start_quad. Αρχικά ελέγχουμε τα δυο βάθη φωλιάσματος, αν το υποπρόγραμμα που την καλεί βρίσκεται στο ίδιο βάθος με την κληθείσα, δηλαδή func_level = f_level τότε καταγράφουμε τις εντολές: lw $t0, -4($sp) και sw $t0, -4($sp). Στην αντίθετη περίπτωση, σημειώνουμε sw $sp, -4($sp). Σε κάθε περίπτωση για να μεταφέρουμε τον δείκτη στοίβας στην κληθείσα σημειώνουμε την εντολή addi $sp, $sp, < το framelength της κληθείσας >. Στη συνέχεια συμπληρώνουμε την εντολή jal L_<start_quad της κληθείσας> για να καλέσουμε το υποπρόγραμμα και για να πάρουμε πίσω τον δείκτη στοίβας την εντολή
  addi $sp, $sp, - < το framelength της κληθείσας >.

- **"end_block"** όταν έχουμε ανάλυση **υποπρογράμματος** ,δηλαδή όταν επιστρέφουμε από μια συνάρτηση**:** Καταγράφουμε τις εντολές lw $ra, ($sp) και jr $ra ώστε να λάβουμε την διεύθυνση επιστροφής από τον $sp στον $ra και να επιστρέψουμε στο σημείο που την κάλεσε με την εντολή jr.

- **"halt":** Όταν εντοπίζουμε το σύμβολο τερματισμού, σημειώνουμε τις παρακάτω εντολές στο τέλος του αρχείου: li $v0, 10 και syscall.

Η συνάρτηση **final_code(name, current, start_quad, end_quad)** εκτελεί επαναληπτικά την asm_file_create() καθώς τοποθετείται στη συνάρτηση block() και γεμίζουμε το αρχείο με τα στοιχεία κάθε υποπρογράμματος, πριν μεταβούμε στην ανάλυση του κύριου μέρους. Για να μην καταγράψουμε τις ίδιες τετράδες παραπάνω από μία φορά, χρησιμοποιούμε την καθολική λίστα start_quad_list, στην οποία τοποθετούνται όλες οι τετράδες που έχουν αναλυθεί. Αν ξανακαλέσουμε την final_code και διαπιστώσουμε πως η θέση που έχουμε αποθηκευμένη στο όρισμα start_quad αντιστοιχεί σε μια τετράδα που έχουμε εξετάσει, προχωράμε στην επανάληψη. Για την ολοκλήρωση της επανάληψης ελέγχουμε αν η start_quad γίνεται ίση με την end_block η οποία αντιστοιχεί στην τετράδα του υποπρογράμματος ή κυρίου μέρους που έχει την συμβολοσειρά "end_block" ως το 1^{ο} στοιχείο της τετράδας.

Τέλος, αξίζει να σημειωθεί πως στη συνάρτηση program() σημειώνεται η πρώτη εντολή του αρχείου .asm , η j Lmain ώστε να μεταβούμε κατευθείαν στο κομμάτι που χειριζόμαστε το κύριο πρόγραμμα.
Σύμφωνα με τους ελέγχους του τελικού κώδικα, στη τωρινή φάση δεν είναι ολοκληρωμένος και χρειαζόμαστε να πραγματοποιήσουμε περισσότερους για να κατανοήσουμε τις αδυναμίες του και να τις διορθώσουμε.

7.γ) Παράδειγμα εκτέλεσης