# Project: Μεταφραστές

Καλπαζίδης Αλέξανδρος : 2985
Μακρής Γρηγόριος : 3022

Καθηγητής : Μανής Γεώργιος

## 1. Εισαγωγή

Το παρών έγγραφο παρέχει πληροφορίες που αφορούν την ανάλυση της εργασία στο μάθημα των Μεταφραστών του 8ου Εξαμήνου σπουδών .

### 1.1 Στόχος

Στόχος αυτής της εργασίας είναι η δημιουργία ενός 'Μεταφραστή' (compiler) για μια νέα γλώσσα ονόματι Starlet της οποίας έχουμε την γραμματική και για την οποία θα παράγουμε κώδικα για τον επεξεργαστή MIPS.

### 1.2 Δομή Εγγράφου

Η δομή του εγγράφου που ακολουθείται παρακάτω βασίζεται στις επιμέρους φάσεις υλοποίησης και είναι η εξής:

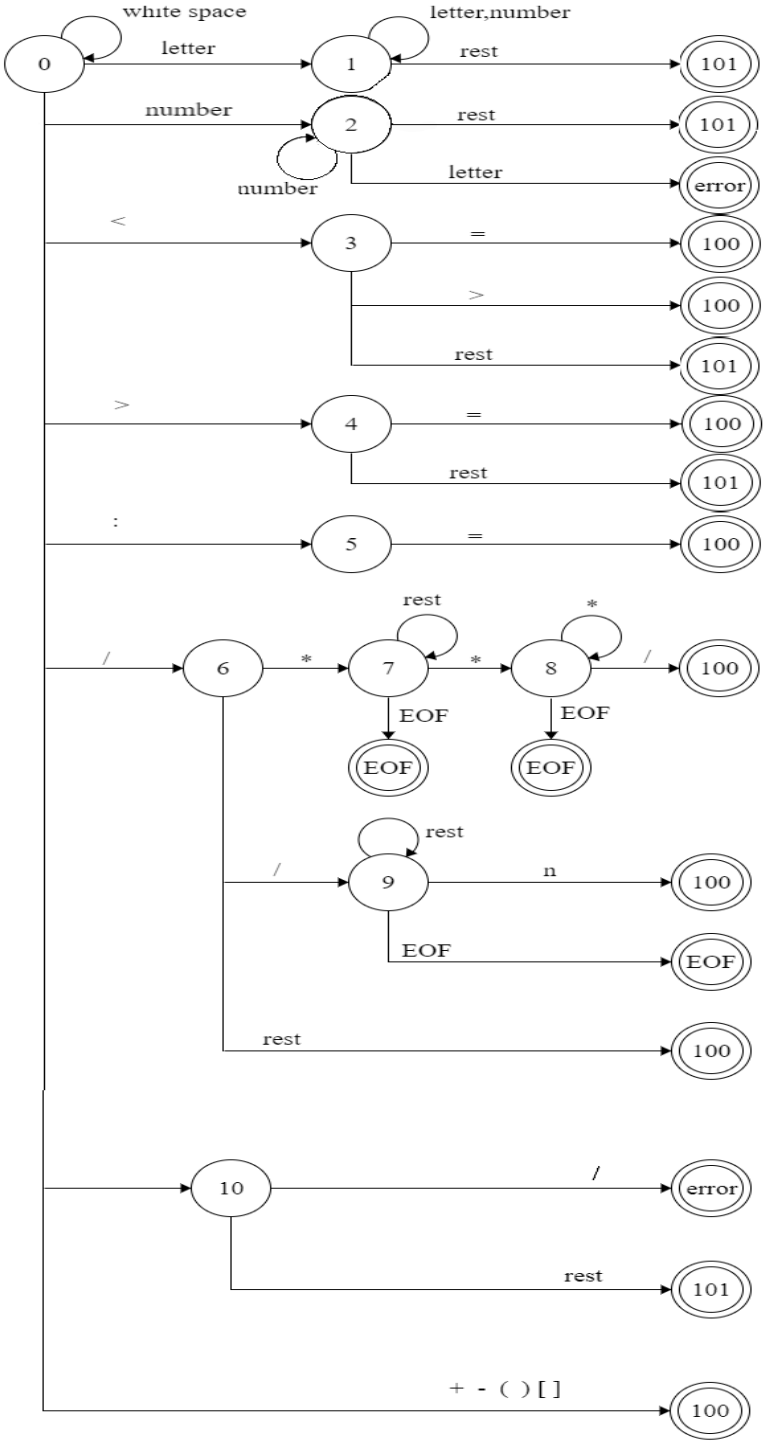
1. Λεκτική Ανάλυση
2. Συντακτική Ανάλυση
3. Σημασιολογική Ανάλυση και Πίνακας Συμβόλων
4. Παραγωγή Ενδιάμεσου Κώδικα
5. Παραγωγή Τελικού Κώδικα

## 2. Ανάλυση Καταστάσεων Υλοποίησης

### 1. Λεκτική Ανάλυση

Η πρώτη διεργασία που πρέπει να γίνει για την μετάφραση ενός κώδικα είναι η λεγόμενη Λεκτική Ανάλυση. Σε αυτή τη φάση στόχος μας είναι, έχοντας ως είσοδο τον κώδικα στη γλώσσα που θέλουμε να μεταφράσουμε (Starlet), να διαβάζουμε χαρακτήρα-χαρακτήρα και με βάση ένα αυτόματο καταστάσεων που έχουμε δημιουργήσει εμείς για της ανάγκες της γλώσσας μας να παράγουμε λεκτικές μονάδες.

Κάθε φόρα που αναγνωρίζεται μια λεκτική μονάδα από τον Λεκτικό αναλυτή, αυτή επιστρέφεται στον Συντακτικό αναλυτή που τον είχε καλέσει (βλ. λεκτικό ) και αυτός με την σειρά του διαχειρίζεται την μονάδα κατά περίπτωση. Παρακάτω ακολουθεί το αυτόματο καταστάσεων για την γλώσσα μας.




Εικόνα 1.

Υπάρχουν κάποιες περιπτώσεις στις οποίες για να αναγνωρίσουμε μια λεκτική μονάδα πρέπει να καταναλώσουμε έναν χαρακτήρα από την επόμενη (βλ. λεκτική μονάδα). Σε αυτές τις περιπτώσεις αφού αναγνωρίσουμε την λεκτική μας μονάδα, ο χαρακτήρα που καταναλώθηκε θα πρέπει να επιστραφεί ώστε να ενσωματωθεί στην επόμενη. Συνεπώς προκύπτει η ανάγκη να αναγνωρίζουμε και άρα να διαχωρίζουμε την κατάσταση στην οποία έχει καταναλωθεί χαρακτήρας ώστε μετά να τον επιστρέψουμε. Για να γίνει το παραπάνω χρησιμοποιούμε όπως φαίνεται και στο αυτόματο μας τη κατάσταση 101 η οποία υποδηλώνει ακριβώς αυτή τη περίπτωση που έχουμε καταναλώσει χαρακτήρα από επόμενη μονάδα. Αντίστοιχα η κατάσταση 100 υποδηλώνει ότι η λεκτική μας μονάδα αναγνωρίστηκε δίχως την κατανάλωση χαρακτήρα από επόμενη μονάδα. Για κάθε χαρακτήρα που διαβάζουμε ενεργούμε με βάση το αυτόματο (βλ. Εικόνα 1) και μεταβαίνουμε στην αντίστοιχη κατάσταση, ενώ ταυτόχρονα κρατάμε σε μία προσωρινή μεταβλητή (`tmp_token`) το μέχρι στιγμής αλφαριθμητικό που έχει δημιουργηθεί από τους αναγνωσμένους χαρακτήρες. Η διαδικασία αυτή επαναλαμβάνεται έως ότου φτάσουμε σε μια από τις τελικές καταστάσεις.

Φτάνοντας σε μία από τις καταστάσεις 100 ή 101 έχουμε διασφαλίσει πως έχουμε επιτυχώς αναγνωρίσει μια ολόκληρη λεκτική μονάδα. Για κάθε λεκτική μονάδα που αναγνωρίζουμε κρατάμε σε μια μεταβλητή (`tmp_tokenId`) ένα αναγνωριστικό το οποίο υποδηλώνει το είδος της (π.χ. το αναγνωριστικό ‘`assignmenttk`’ υποδηλώνει ότι η λεκτική μονάδα αυτή είναι ένας τελεστής ανάθεσης). Αφότου έχει αναγνωριστεί επιτυχώς η μονάδας μας ελέγχουμε αν η τελική κατάσταση μέσω της οποίας αναγνωρίστηκε ήταν η κατάσταση 101 και αν ναι, τότε φροντίζουμε να τροποποιήσουμε το αλφαριθμητικό μας που είναι αποθηκευμένο στην μεταβλητή `tmp_token` αφαιρώντας τον τελευταίο του χαρακτήρα, καθώς αυτός αποτελεί κομμάτι της επόμενης λεκτικής μονάδας. Επίσης φροντίζουμε να επανατοποθετήσουμε τον δείκτη ανάγνωσης του αρχείο μας μία θέση πιο πίσω από ότι βρίσκεται ώστε να μπορέσει να ξαναδιαβαστεί ο χαρακτήρα που λαθεμένα καταναλώθηκε από την προηγούμενη λεκτική μονάδα ενώ αφορούσε την επόμενη. Όταν πλέον έχει τελειώσει και η όποια τυχόν τροποποίηση του αλφαριθμητικού της λεκτικής μας μονάδας ελέγχουμε σε περίπτωση που είναι αριθμός ότι βρίσκεται μέσα στα επιτρεπτά όρια που ορίζει η γλώσσα μας (από -32767 έως 32767) και στην περίπτωση που είναι οτιδήποτε άλλο πέραν από αριθμό και τελεστή (δηλαδή αν είναι αλφαριθμητικό τύπου ‘`idtk`’), ότι το μήκος του δεν ξεπερνά τους 30 χαρακτήρες. Στην περίπτωση αυτή πέραν του μήκους ελέγχουμε και αν το αλφαριθμητικό αυτό είναι κάποια από τις δεσμευμένες λέξεις της γλώσσας μας και αν ναι τροποποιούμε στο αναγνωριστικό του που βρίσκεται στην μεταβλητή `tmp_tokenId` και βάζουμε το κατάλληλο για την δεσμευμένη λέξη αυτή (π.χ. για την λέξη ‘`while`’ αναθέτουμε το αναγνωριστικό ‘`whiletk`’). Τέλος εφόσον ελέγξουμε ότι η λεκτική μας μονάδα δεν είναι ‘σχόλια’ επιστρέφουμε στον Συντακτικό Αναλυτή που μας κάλεσε την λεκτική μονάδα που αναγνωρίσαμε καθώς και το αναγνωριστικό της.

## 2. Συντακτική Ανάλυση

Ο βασικός ρόλος που έχει ο Συντακτικός Αναλυτής είναι να διαπιστώσει εάν το πηγαίο πρόγραμμα ανήκει ή όχι στην γλώσσα με βάση την γραμματική της καθώς και να δημιουργήσει το κατάλληλο «περιβάλλον» μέσα από το οποίο αργότερα θα κληθούν οι σημαντικές ρουτίνες. Υπάρχουν αρκετοί τρόποι για να κατασκευαστεί ένας συντακτικός αναλυτής. Για τις ανάγκες της εργασίας θα προτιμήσουμε τη συντακτική ανάλυση με αναδρομική κατάβαση η οποία βασίζεται σε γραμματική LL(1). Η γραμματική αυτή αναγνωρίζει από αριστερά στα δεξιά, την αριστερότερη δυνατή παραγωγή και όταν βρίσκεται σε δίλλημα ποιόν κανόνα να ακολουθήσει της αρκεί να κοιτάξει το αμέσως επόμενο σύμβολο στην συμβολοσειρά εισόδου.

***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>] |
                              <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>
```

Εικόνα 2.

Παραπάνω φαίνεται η γραμματικής της γλώσσας μας πάνω στην οποία βασιζόμαστε.
Με βάση αυτή για κάθε έναν από τους κανόνες της, φτιάχνουμε και ένα αντίστοιχο υποπρόγραμμα. Η απόφαση του αν θα κληθεί ή όχι κάποιο υποπρόγραμμα, καθώς και αν εν τέλει κληθεί, το ποίο θα είναι αυτό, καθορίζεται από την λεκτική μονάδα και το αναγνωριστικό της που γυρνάει ως ζεύγος τιμών από την κλήση του Λεκτικού αναλυτή `lex()`. Συνεπώς υπεύθυνος για το πότε και που θα κληθεί ο Λεκτικός αναλυτής είναι ο Συντακτικός αναλυτής. 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 παράγεται το αντίστοιχο μήνυμα λάθους. Όταν αναγνωριστεί και η τελευταία λέξη του πηγαίου προγράμματος, τότε η συντακτική ανάλυση έχει στεφτεί με επιτυχία.

# 3. Σημασιολογική Ανάλυση και Πίνακας Συμβόλων

Κάθε γλώσσα απαρτίζεται από ένα αλφάβητο και διάφορους κανόνες-ιδιότητες που την διέπουν, όπως για παράδειγμα η γραμματική και συντακτικό. Υπάρχουν όμως περιπτώσεις που παρότι τόσο οι κανόνες της γραμματικής όσο και του συντακτικού ακολουθούνται πιστά μπορεί η τελική πρόταση που προκύπτει είτε να μην βγάζει νόημα είτε να δίνει διαφορετικό από αυτό που κάποιος θα ανέμενε. Συνεπώς υπάρχουν και κάποιοι κανόνες που τόσο η γραμματική όσο και το συντακτικό δεν μπορούν να «πιάσουν». Ο ρόλος λοιπόν της σημασιολογικής ανάλυσης είναι η διασφαλίσει ότι τηρούνται οι ιδιότητες της γλώσσας μας. Οι ιδιότητες, που σχετίζονται με το περιεχόμενο των δομικών στοιχείων που αναγνωρίζονται και δε μπορούν να καθορισθούν μόνο με τη συντακτική ανάλυση του προγράμματος, συγκροτούν τη στατική σημασία αυτού. Αντίθετα, η δυναμική σημασία ενός προγράμματος εκφράζεται από ιδιότητες, που καθορίζονται πλήρως μόνο κατά την εκτέλεσή του. Οι πιο συνηθισμένες περιπτώσεις στατικής σημασίας είναι οι δηλώσεις και οι έλεγχοι τύπων δεδομένων. Στην δική μας περίπτωση δεν θα μας απασχολήσει ο έλεγχος του τύπου δεδομένων καθώς η γλώσσα μας υποστηρίζει μόνο έναν τύπου (`int`). Αυτό που όμως πρέπει να ελέγξουμε είναι τα εξής:

- Κάθε συνάρτηση θα πρέπει να έχει τουλάχιστον ένα `return`.
- Δεν πρέπει να υπάρχει `return` έξω από συνάρτηση.
- Θα πρέπει να υπάρχει `exit` μόνο μέσα σε βρόχους `loop-endloop`.

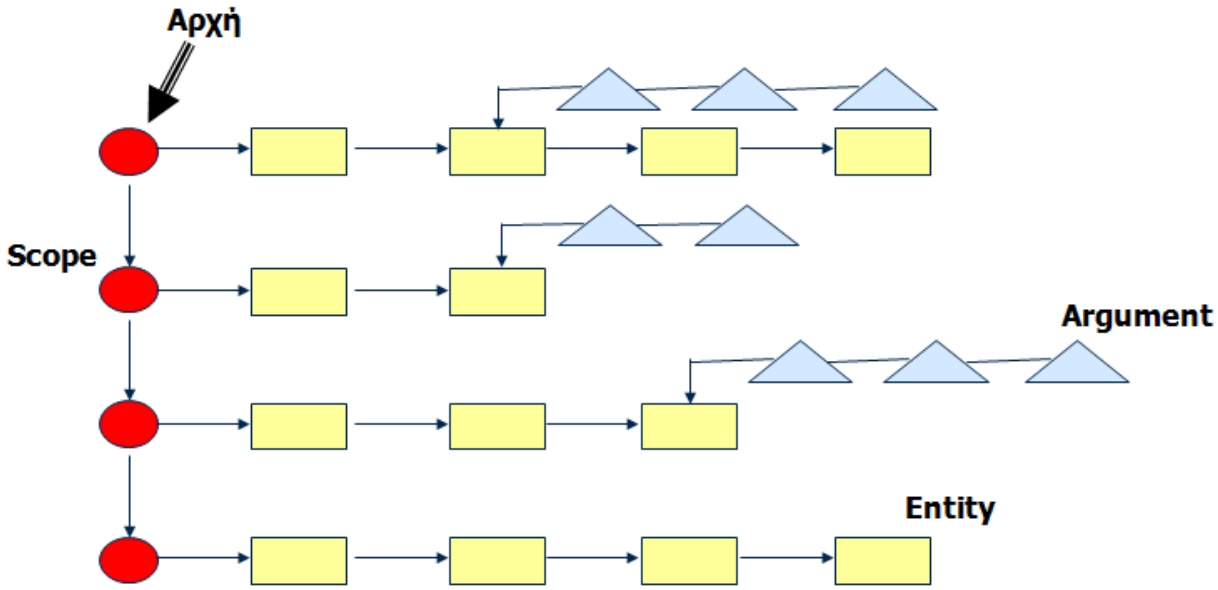
Τόσο για την πρώτη όσο και για την δεύτερη περίπτωση που σχετίζονται με `return`, αυτό που χρησιμοποιούμε για να ελέγξουμε τις δυο αυτές απαιτήσεις της γλώσσας μας είναι μια `global` λίστα ονόματι `funcLevel`, στην οποία κρατάμε πόσα `return` υπάρχουν μέσα στην κάθε συνάρτηση (π.χ. τα πόσα return έχει η i-οστή σε βάθος συνάρτηση βρίσκεται στην `funcLevel[i]` ). Συνεπώς αν υπάρχει return αλλά η λίστα μας είναι άδεια σημαίνει ότι το `return` αυτό βρίσκεται έξω από συνάρτηση και παράγεται το κατάλληλο μήνυμα λάθους. Αντίστοιχα αν φτάσουμε στο τέλος μιας συνάρτησης **i** και η τιμής της λίστα για το στοιχείο αυτό είναι μηδέν (`funcLevel[i] = 0`) σημαίνει ότι μέσα στο σώμα αυτής της συνάρτησης δεν υπήρχε return και παράγεται το κατάλληλο μήνυμα λάθους. Με την ίδια λογική για το `exit` αξιοποιώντας μια `global` λίστα ονόματι `loopList` την οποία ενημερώνουμε κάθε φορά που υπάρχει ένα νέο `loop` , αν βρούμε `exit` και η λίστα μας είναι άδεια τότε παράγουμε και το αντίστοιχο μήνυμα λάθους, καθώς αυτό μας υποδηλώνει πως το exit βρίσκεται εκτός `loop`.

(*)Πέραν όμως από τις παραπάνω απαιτήσεις υπάρχουν και άλλες στις οποίες όμως θα χρειαστεί να κάνουμε χρήση του Πίνακα συμβόλων. Οι απαιτήσεις αυτές είναι η εξής:

- Κάθε μεταβλητή ή συνάρτηση που έχει δηλωθεί δεν πρέπει να έχει δηλωθεί πάνω από μία φορά στο ίδιο βάθος φωλιάσματος στο οποίο βρίσκεται.
- Κάθε μεταβλητή ή συνάρτηση που χρησιμοποιείται πρέπει να έχει δηλωθεί και μάλιστα με τρόπο που χρησιμοποιείται (σαν μεταβλητή ή σαν συνάρτηση)

- Οι παράμετροι με τις οποίες καλούνται οι συναρτήσεις περνάνε με το ίδιο τρόπο (π.χ. πέρασμα με αναφορά ή με τιμή) με αυτές με τις οποίες έχουν δηλωθεί και ακλουθούν την ίδια σειρά.

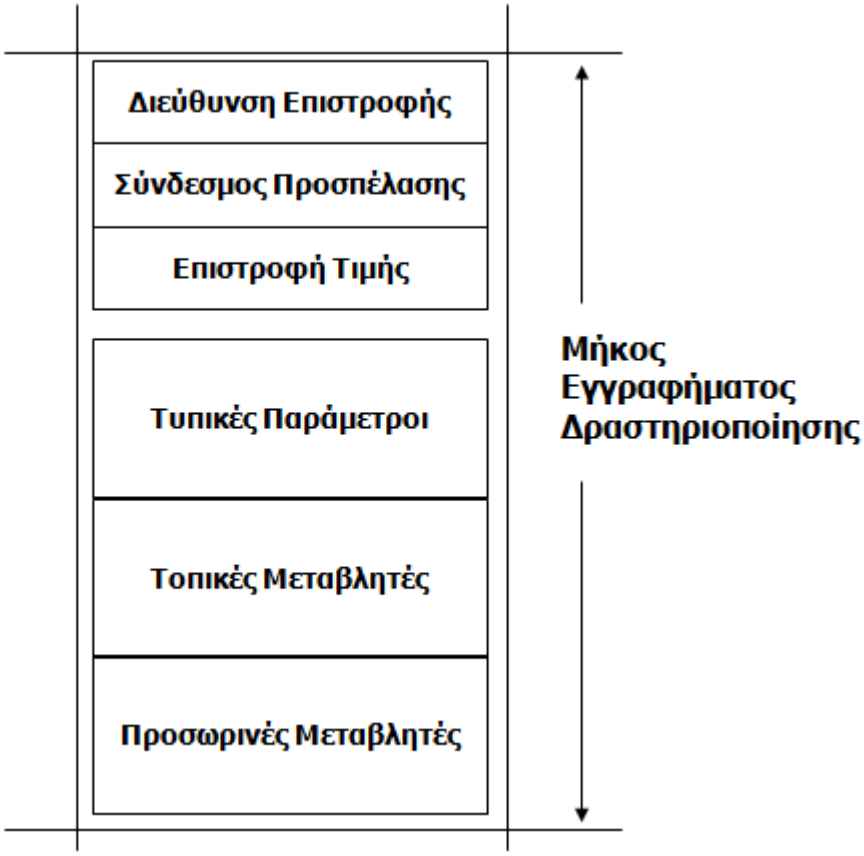
Ο πίνακας συμβόλων καταγράφει πληροφορίες, που σχετίζονται με ονόματα, είτε αυτά είναι ονόματα συναρτήσεων, είτε ονόματα τύπων δεδομένων, μεταβλητών ή και σταθερών. Χρησιμοποιείται σχεδόν σε όλες τις φάσεις της μεταγλώττισης. Πιο συγκεκριμένα, κατά τη λεξική, τη συντακτική και τη σημασιολογική ανάλυση καταχωρούνται σε αυτόν ονόματα, η εκμετάλλευση των οποίων γίνεται κατά τη βελτιστοποίηση και τη σύνθεση του τελικού προγράμματος. Η δομή που θα χρησιμοποιήσουμε εμείς είναι η εξής:




Εικόνα 3.

Το κάθε Scope ουσιαστικά υποδηλώνει ένα Εγγράφημα Δραστηριοποίησης το οποίο δημιουργείται για κάθε νέα κλήση συνάρτησης και υπεύθυνος για την δημιουργία του είναι η συνάρτηση που την καλεί. Στο κάθε Scope υπάρχουν εγγραφές, τα λεγόμενα Entity, (βλ. Εικόνα 3) τα οποία μπορεί είναι : μεταβλητές, συναρτήσεις, σταθερές, παράμετροι, προσωρινές μεταβλητές. Όλες έχουν κάποια κοινά χαρακτηριστικά όπως για παράδειγμα όνομα, τιμή απόκλισης (offset) και τύπο όμως η κάθε μία από αυτές έχει και κάποια πιο ειδικά χαρακτηρίστηκα. Για παράδειγμα μια εγγραφή τύπου συνάρτησης έχει και μία λίστα από Arguments (βλ. Εικόνα 3) τα οποία υποδηλώνουν τους παραμέτρους της συνάρτησης και το βασικό τους πεδίο σαν δομή είναι το τρόπος περάσματος (parMode).

Το Εγγράφημα δραστηριοποίησης Περιέχει πληροφορίες που χρησιμεύουν για την εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί. Όλη αυτή η πληροφορία για να δημιουργηθεί σωστά το Εγγράφημα περιέχεται στον Πινάκα Συμβόλων και τα δομικά τους στοιχεία που περιγράψαμε παραπάνω. Η δομή του Εγγραφήματος Δραστηριοποίησης είναι η εξής:




Εικόνα 4.

- **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.

Όπως αναφέραμε και παραπάνω αυτό που θα μας βοηθήσει στο να συμπληρώσουμε το εγγράφημα μας θα είναι ο Πινάκας συμβόλων μας. Συνεπώς για να μπορούμε να προσθέτουμε, αφαιρούμε και αναζητούμε στοιχεία στον πίνακα μας έχουμε δημιουργήσει και τις αντίστοιχες συναρτήσεις

- **addScope()** : Δημιουργεί και προσθέτει ένα νέο RecordScope στη global λίστα μας scope που είναι και η βασική δομή του Πίνακα μας. Καλείται κάθε φορά που ξεκινάμε τη μετάφραση μια νέας συνάρτησης.

- **deleteScope()** : Διαγράφει το τελευταίο στοιχείο της global λίστας μας scope και ενημερώνει μια global μεταβλητή ονόματι offset την οποία χρησιμοποιούμε για να αναθέτουμε την κατάλληλη κάθε φορά τιμή απόκλισης σε νέες εγγραφές (Entities). Καλείται στο τέλος κάθε συνάρτησης.

- **addEntity()**: Δημιουργεί και προσθέτει ένα νέο Entity στο τέλος της λίστα του τελευταίου recordScope και είναι ενημερώνει κατάλληλα την global μεταβλητή offset.

- **addArgs()**:Δημιουργεί και προσθέτει ένα νέο Argument στο τέλος της λίστα ενός Entity τύπου function.

- **returnArgs()**: Επιστρέφει μία λίστα όλων των Arguments ενός Entity τύπου function.

- **findEntity()**: Αναζητά στον Πίνακα συμβόλων και επιστρέφει σε αυτόν που την κάλεσε, το πρώτο Entity που βρίσκει με το όνομα που του πέρασαν σαν παράμετρο. Η αναζήτηση ξεκινάει από το τελευταίο recordScope (δηλαδή αυτό με το μεγαλύτερο βάθος φωλιάσματος).

- **findEntity_Type()** : Κάνει ότι ακριβώς και η findEntity() όμως πλέον κριτήριο δεν είναι μόνο το όνομα αλλά και ο τύπος ου Entity που αναζητάμε.

Συνεπώς για να μπορέσουμε να καλύψουμε και τις υπόλοιπες τρείς απαιτήσεις που αναφέραμε παραπάνω (*) κάνουμε χρήσει τον συναρτήσεων που μόλις αναφέραμε για τον Πίνακα μας.

# 4. Παραγωγή Ενδιάμεσου Κώδικα

Ο ενδιάμεσος κώδικας χρησιμοποιείται για μια διαφορετική αναπαράσταση του πηγαίου κώδικα μας ώστε να μπορέσουμε να κάνουμε εν τέλει πιο εύκολα τόσο μια βελτιστοποίηση στον κώδικα μας όσο και για να παράγουμε τον τελικό μας κώδικα. Η μορφή ενδιάμεσου κώδικα που χρησιμοποιούμε για τις ανάγκες της εργασίας είναι ένα σύνολο από τετράδες.

- ένας τελεστής
- τρία τελούμενα

π.χ. + a, b, t_1
* t_1,2,t_2
:= t_2,_,c

Οι τετράδες είναι αριθμημένες. Κάθε τετράδα έχει μπροστά της έναν μοναδικό αριθμό που τη χαρακτηρίζει. Μόλις τελειώσει η εκτέλεση μίας τετράδας εκτελείται η τετράδα που έχει τον αμέσως μεγαλύτερο αριθμό, εκτός εάν η τετράδα που μόλις εκτελέστηκε υποδείξει κάτι διαφορετικό.

π.χ. 100: +, a, b, c
110: +, d, e, f

Για την αποθήκευση αυτής της τετράδας χρησιμοποιούμε μια δομής ονόματι `Quad` με πεδία `op , x , y , z` για τον έναν τελεστή και τα τρία τελούμενα αντίστοιχα. Για την αποθήκευση όλων των τετράδων του κώδικα που θέλουμε να μεταφράσουμε έχουμε μία global λίστα από `Quads` ονόματι `quadList` την οποία ενημερώνουμε όποτε παράγεται ένα νέο `Quad` μέσω των αντίστοιχων βοηθητικών ρουτινών που αναφέρουμε παρακάτω.

Βοηθητικές Υπορουτίνες:

- **`nextQuad()`:** Επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί.
- **`genQuad()`:** Δημιουργεί την επόμενη τετράδα και την προσθέτει στη λίστα με τα `Quads`.
- **`newTemp()`:** Δημιουργεί και επιστρέφει μια νέα προσωρινή μεταβλητή.
- **`emptyList()`:** Δημιουργεί μια κενή λίστα ετικετών τετράδων.
- **`makeList()`:** Δημιουργεί μια λίστα ετικετών που περιέχει μόνο την τετράδα που της περνάμε σαν όρισμα.
- **`merge()`:** Δημιουργεί μια λίστα ετικετών τετράδων από την συνένωση των δύο άλλων που παίρνει ως ορίσματα.
- **`backpatch()`:** Ενημερώνει το τελευταίο πεδίο κάθε τετράδα που βρίσκεται μέσα στην λίστα που της περνάμε σαν πρώτο όρισμα με μια ετικέτα που της περνάμε ως δεύτερο όρισμα.

Αφότου έχουμε ενημερώσει επιτυχώς την quadList με όλες μας τις τετράδες (Quads) το μόνο που μένει είναι η εγγραφή του αρχείου .int που θα περιέχει τον συνολικό ενδιάμεσο κώδικα συμπληρωμένο και με τα κατάλληλα labels ώστε να γίνονται σωστά τα άλματα. Η διαδικασία αυτή γίνεται με την συνάρτηση produce() η οποία διατρέχει την quadList και γράφει σε ένα νέο αρχείο και υπολογίζει τα κατάλληλα label για κάθε εντολή ανάλογα με την θέση του κάθε Quad στην λίστα.

## 5. Παραγωγή Τελικού Κώδικα

Η παραγωγή του τελικού κώδικα είναι και η τελευταία φάση της μεταγλώττισης ενός προγράμματος και γίνεται παράγοντας για κάθε μία από τις εντολές του ενδιάμεσου κώδικα τις αντίστοιχες εντολές του τελικού για πού στην περίπτωση μας είναι κώδικα για τον επεξεργαστή MIPS. Οι κύριες ενέργεια στη φάση αυτή είναι τόσο η απεικόνιση των μεταβλητών στην μνήμη όσο και το πέρασμα παραμέτρων και η κλήση συναρτήσεων.

Η κύρια ιδέα της μεταγλώττισης κάθε εντολής ενδιάμεσου σε τελικό είναι η εξής:
Αυτό που χρειάζεται για να συμπληρωθούν όλα τα πεδία των εντολών τελικού κώδικα για την κάθε εντολή ενδιάμεσου με την οποία έχει αντιστοιχηθεί είναι η θέση κάθε μεταβλητής στη μνήμη ώστε να μπορεί να προσπελαστεί, με άλλα λόγια η διεύθυνση της. Για να υπολογιστεί η διεύθυνση χρειάζονται κυρίως δύο πράγματα. Μία αρχική διεύθυνση η οποία είναι το σημείο αναφοράς μας και για κάθε μια συνάρτηση είναι διαφορετική και ισούται με την διεύθυνση που δείχνει στην αρχή του Εγγραφήματος δραστηριοποιήσεις κάθε συνάρτησης. Η διεύθυνση αυτή είναι αποθηκευμένη στον καταχωρητή $sp. Και από μία σχετική απόκλιση offset από αυτή την διεύθυνση που μόλις αναφέραμε. Για να μπορέσουμε όμως να γνωρίζουμε αυτά τα δύο χαρακτηριστικά θα πρέπει να ξέρουμε πόσο είναι το μέγεθος του εγγραφήματος δραστηριοποιήσεις, πράγμα που γίνεται γνωστό στο τέλος κάθε συνάρτηση. Άρα πριν από το τέλος κάθε συνάρτησης υπολογίζουμε το μήκος του εγγραφήματος (frameLength) και καλούμε μια δική μας ρουτίνα ονόματι translate().
Η ρουτίνα αυτή κάθε φορά που καλείται (δηλαδή μια φορά για κάθε συνάρτηση) διατρέχει την quadList και έχοντας πλέον όλα όσα χρειάζεται μεταφράζει κάθε εντολή ενδιάμεσου κώδικα (κάθε τετράδα Quad) που αντιστοιχεί στη συνάρτηση της. Για να το κάνει αυτό βέβαια χρησιμοποίει και βοηθητικές υπορουτίνες οι οποίες είναι υπεύθυνες για την αναζήτηση, την φόρτωση και την εγγραφή των μεταβλητών που μας ενδιαφέρουν από και προς τη μνήμη.

Τέλος για κάθε εντολή που μεταφράζουμε την προσθέτουμε σε μία λίστα ονόματι asmList και καλούμε την συνάρτηση produce() η οποία εν τέλει είναι και αυτή που γράφει στο αρχείο μας .asm τον τελικό κώδικα που είναι έτοιμος προς εκτέλεση.