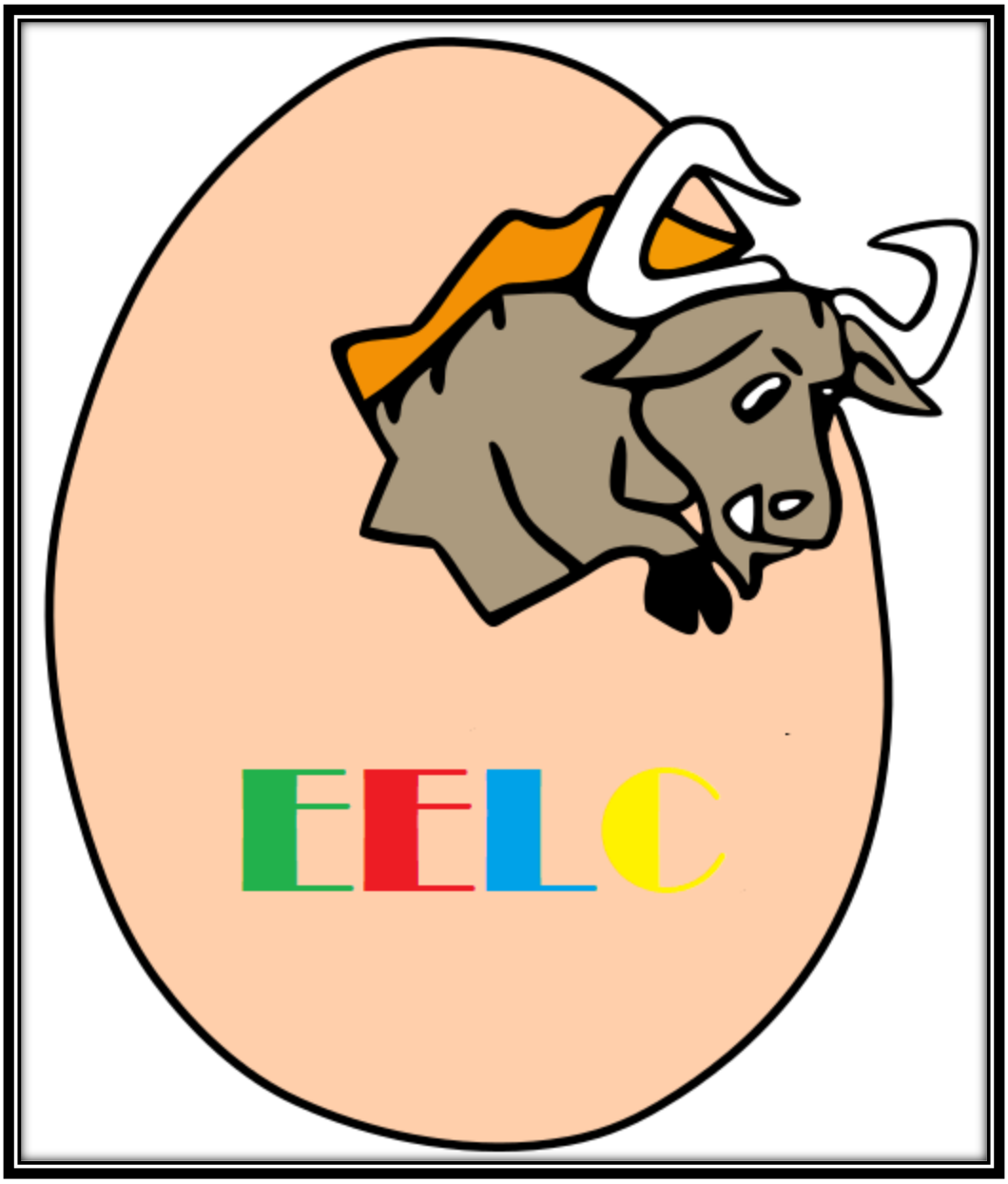


ΜΕΤΑΦΡΑΣΤΕΣ

«ΔΗΜΙΟΥΡΓΙΑ ΜΕΤΑΦΡΑΣΤΗ ΓΙΑ ΤΗΝ ΓΛΩΣΣΑ EEL»

ΔΕΛΗΓΙΑΝΝΗΣ ΝΙΚΟΣ (2681)

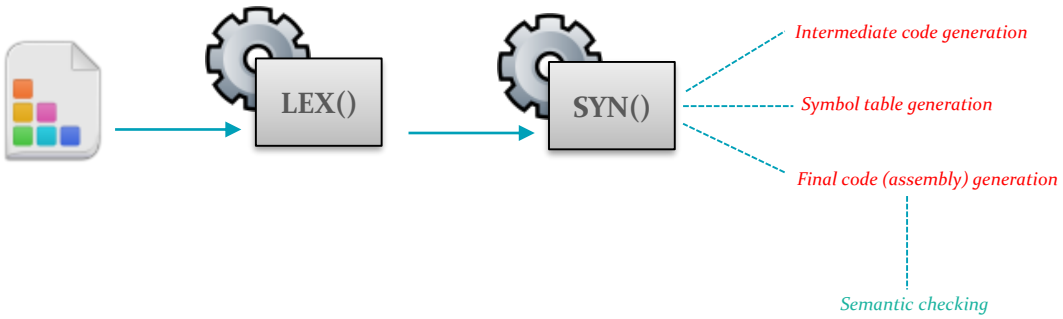
ΠΕΡΙΕΧΟΜΕΝΑ:

[1] ΑΝΑΛΥΣΗ ΑΠΑΙΤΗΣΕΩΝ – ΤΡΟΠΟΣ ΕΡΓΑΣΙΑΣ

Η παρούσα εργασία αφορά την «δημιουργία» μιας δικής μας γλώσσας, της γλώσσας EEL ("Early Experimental Language"). Για να είμαστε λοιπόν σε θέση να φτιάξουμε μια γλώσσα χρειαζόμαστε δύο (2) οντότητες. Αρχικά, πρέπει να συντάξουμε προσεκτικά την γραμματική της γλώσσας που επιθυμούμε να κατασκευάσουμε. Η γραμματική της γλώσσας (βλ Ενότητα 1.α) μας δώθηκε και μάλιστα πρόκειται για γραμματική τύπου LL(1). Αναλυτικότερα, ο ορισμός LL(1) σημαίνει το εξής, **L**eft to Right (ανάγνωση εισόδου απο αριστερά προς τα δεξιά), **L**eftmost Derivation (παραγωγή λέξης απο αριστερά προς τα δεξιά) και ο αριθμός ένα (1) συμβολίζει το πλήθος τον συμβόλων εισόδου που χρειαζόμαστε για να πάρουμε μια απόφαση (κατα το parsing) σχετικά με το ποιόν κανόνα να ακολουθήσουμε. Το δεύτερο κομμάτι – οντότητα που χρειαζόμαστε είναι ο μεταφραστής (compiler) ο οποίος πρέπει να υλοποιήσουμε πιστά, βάσει πάντοτε της γραμματικής που έχουμε. Ο μεταφραστής μας (EELC – Early Experimental Language Compiler) που υλοποιήθηκε σε γλώσσα **`Python`** (έκδοση 2) ακολουθεί την ακόλουθη στρατηγική:

a) Λεκτική ανάλυση πηγαίου κώδικα.
b) Συντακτική ανάλυση πηγαίου κώδικα.
c) Παραγωγή ενδιάμεσου κώδικα
d) Δημιουργία του πίνακα συμβόλων
e) Παραγωγή τελικού κώδικα – Σημασιολογικός έλεγχος

Σημειώνεται πως η παρούσα έκδοση του μεταφραστή παράγει τελικό κώδικα για τον επεξεργαστή MIPS.



Εικόνα 1.1: « Εικονική Αναπαράσταση Λειτουργίας του Μεταφραστή μας »

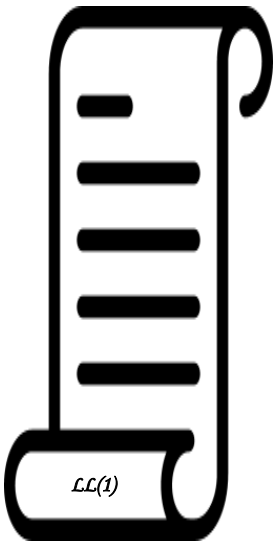
Έχοντας πλέον καταλάβει απολύτως το έργο το οποίο καλούμαστε να υλοποιήσουμε προχωρούμε βήμα βήμα στην παραγωγή του. Η εργασία ολοκληρώθηκε σε τρείς (3) συνολικά φάσεις:

1. Λεκτική και Συντακτική ανάλυση.
2. Παραγωγή ενδιάμεσου κώδικα.
3. Δημιουργία Πίνακα Συμβόλων και Παραγωγή Τελικού κώδικα

[1.α] Η ΓΡΑΜΜΑΤΙΚΗ ΤΗΣ ΓΛΩΣΣΑΣ EEL

Όπως αναφέρθηκε στην προηγούμενη ενότητα, η γραμματική της γλώσσας μας είναι μια γραμματική τύπου **LL(1)**. Αποτελείται συνολικά απο τριάντα-εννιά (39) κανόνες, οι οποίοι είναι οι ακόλοθοι (παρουσιάζονται σε μορφή **BNF** – Backus Naur Form):

```
<PROGRAM>          ::= program id <BLOCK> endprogram
<BLOCK>            ::= <DECLARATIONS> <SUBPROGRAMS> <STATEMENTS>
<DECLARATIONS>     ::= ε | declare <VARLIST> enddeclare
<VARLIST>          ::= ε | id (,id)*
<SUBPROGRAMS>      ::= (<PROCORFUNC>)*
<PROCORFUNC>       ::= procedure id <PROCORFUNCBODY> endprocedure |
                       function  id <PROCORFUNCBODY> endfunction
<PROCORFUNCBODY>   ::= <FORMALPARS><BLOCK>
<FORMALPARS>       ::= (<FORMALPARLIST>)
<FORMALPARLIST>    ::= ε | <FORMALPARITEM> (, <FORMALPARITEM> )*
<FORMALPARITEM>    ::= in id | inout id
<STATEMENTS>       ::= <STATEMENT> (; <STATEMENT>)*
<STATEMENT>        ::= ε | <ASSIGNMENT-STAT> | <IF-STAT>    | <WHILE-STAT>
                         | <REPEAT-STAT>     | <EXIT-STAT> | <SWITCH-STAT>
                         | <FORCASE-STAT>    | <CALL-STAT> | <RETURN-STAT>
                         | <INPUT-STAT>      | <PRINT-STAT>
<ASSIGNMENT-STAT>::= id := <EXPRESSION>
<IF-STAT>          ::= if <CONDITION> then <STATEMENTS> <ELSEPART> endif
<ELSEPART>         ::= ε | else <STATEMENTS>
<REPEAT-STAT>      ::= repeat <STATEMENTS> endrepeat
<EXIT-STAT>        ::= exit
<WHILE-STAT>       ::= while <CONDITION> <STATEMENTS> endwhile
<SWITCH-STAT>      ::= switch <EXPRESSION> (case <EXPRESSION>:<STATEMENTS)+ endswitch
<FORCASE-STAT>     ::= forcase (when <CONDITION>:STATEMENTS>)+ endforcase
<CALL-STAT>        ::= call id <ACTUALPARS>
<RETURN-STAT>      ::= return <EXPRESSION>
<CONDITION>        ::= <BOOLTERM> (or <BOOLTERM>)*
<BOOLTERM>         ::= <BOOLFACTOR> (and <BOOLFACTOR>)*
<BOOLFACTOR>       ::= not[<CONDITION>] | [<CONDITION>] | <EXPRESSION><RELATIONAL-OPER><EXPRESSION>
                                        | TRUE          | FALSE
<EXPRESSION>       ::= <OPTIONAL-SIGN><TERM>(<ADD-OPER><TERM>)*
<TERM>             ::= <FACTOR>(<MUL-OPER><FACTOR>)*
<FACTOR>           ::= costant | (<EXPRESSION>) | id <IDTAIL>
<IDTAIL>           ::= ε | <ACTUALPARS>
<RELATIONAL-OPER>::= = | <= | >= | > | < | <>
<ADD-OPER>         ::= + | -
<MUL-OPER>         ::= * | /
<OPTIONAL-SIGN>    ::= ε | <ADD-OPER>
```



Εικόνα 1.2: « Η γραμματική της EEL »

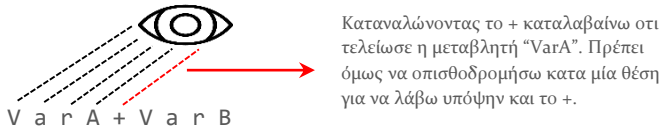
[2] ΣΤΑΔΙΟ Α: ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Στο πρώτο αυτό στάδιο, η δουλεία μας είναι να ελέγξουμε εάν το αλφάβητο που χρησιμοποίησε ο χρήστης για την παραγωγή κώδικα σε EEL είναι όντως αυτό που έχουμε ορίσει εμείς ρητά. Αυτόν τον ρόλο, έχει ο λεκτικός αναλυτής όπου μετατρέπει μια ακολουθία απο χαρακτήρες σε μια ακολουθία απο λεκτικές μονάδες (**tokens**). Στόν πηγαίο κώδικα του μεταφραστή μας, δύο (2) είναι οι συναρτήσεις που αφορούν την λεκτική ανάλυση:

- Η συνάρτηση **`lex()`**
- Η συνάρτηση **`backtrack()`**

Η συνάρτηση **`lex()`** διαβάζει απο το αρχείο πηγαίου κώδικα σε γλώσσα EEL και επιστρέφει κάθε φορά μία λεκτική μονάδα. Επίσης σε κάθε περιπτωσή και όπου αυτό κρίνεται απαραίτητο, επιστρέφει και μήνυμα λάθους (π.χ. διάβασμα χαρακτήρα που δεν ανήκει στο αλφαβητό μας). Ο μεταφραστής μας αυτό που κάνει αρχικά είναι να περνάει ολόκληρο τον κώδικα μία φορά απο λεκτική ανάλυση και έπειτα, ξεκινάει την συντακτική η οποία φυσικά χρειάζεται την προηγούμενη σε κάθε βήμα της, σε κάθε κανόνα της. Επίσης η συνάρτηση **`lex()`** είναι υπεύθυνη για το μέτρημα των γραμμών του κώδικα (escape χαρακτήρες **`'\n'`**), η οποία δεν είναι πάντα απόλυτα ακριβής. Για την ακρίβεια υπάρχει μια μικρή απόκλιση μερικών γραμμών ($\pm$) σε κάποιες περιπτώσεις.

Οπως είπαμε αρκετές φορές στο προηγόυμενο κεφάλαιο, η γραμματική της γλώσσας μας είναι τύπου **`LL(1)`**. Γνωρίζοντας έτσι πως χρειαζόμαστε αρκετές φορές να διαβάσουμε ένα συμβολο για να πάρουμε μια απόφαση για το ποιόν κανόνα να ακολουθήσουμε, αλλα και πολλές φορές για λόγους επιβεβαίωσης της εισόδου που διαβάζει ο **`lex()`** καταναλαβαίνουμε πως σε αρκετές περιπτώσεις καταναλώνεται ένας επιπλέον χαρακτήρας ο οποίος δεν λαμβάνεται υπόψην την συγκεκριμένη στιγμή.

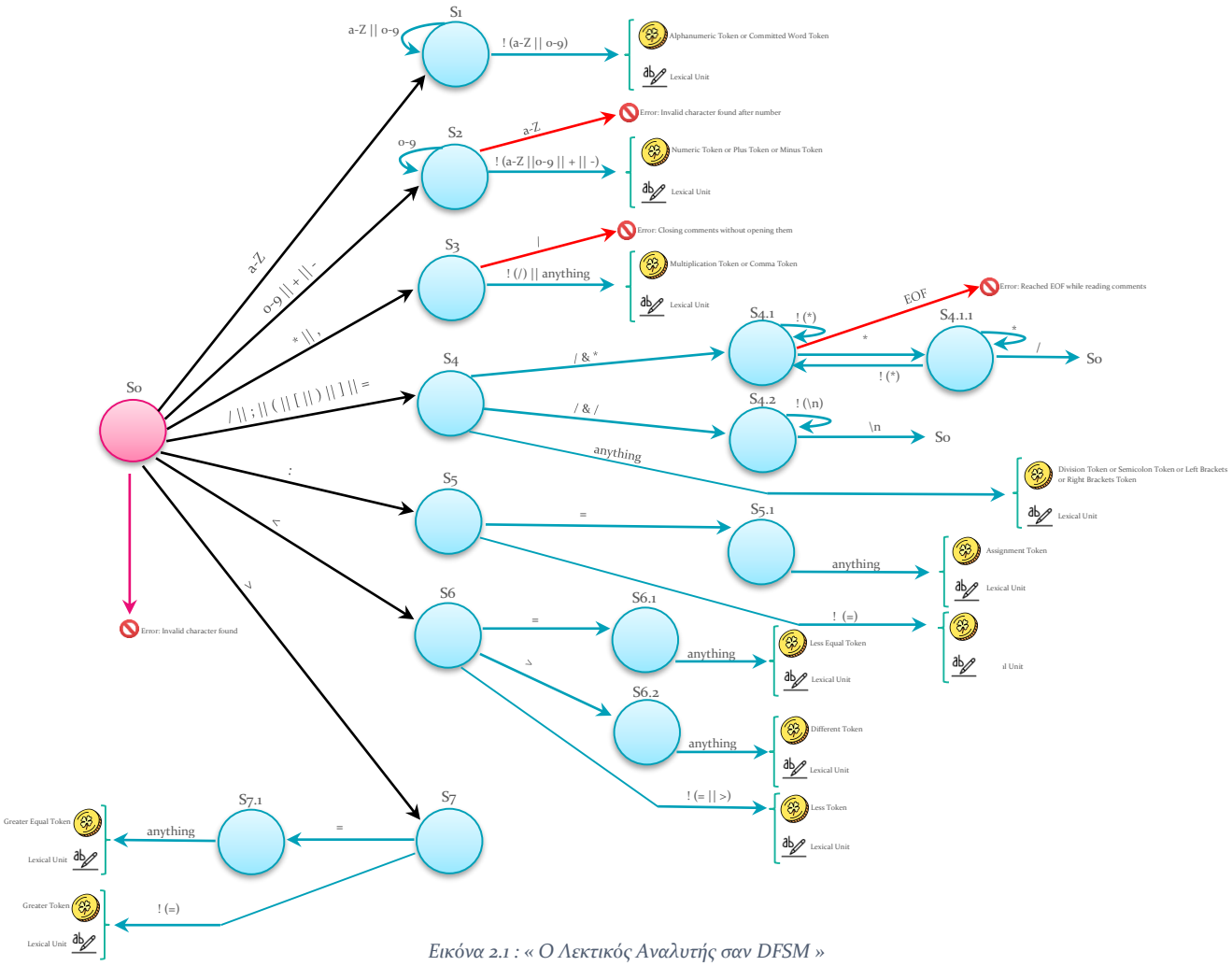


Εικόνα 2.1: « Παράδειγμα Αναγκαιότητας της Οπισθοδρόμησης »

Για τον λόγο αυτό χρησιμοποιούμε την συνάρτηση **`backtrack()`** η οποία αυτό που κάνει είναι να γυρίζει τον δείκτη αρχείου που αφορά τον πηγαίο κώδικα κατά μία θέση πίσω. Έτσι κάνουμε βέβαιο σε κάθε περίπτωση, πως όταν κληθεί ξανά η συνάρτηση **`lex()`** θα ληφθεί υπόψην η λεκτική μονάδα που προηγουμένως είχε παραλειφθεί (π.χ. Εικόνα 2.1).

[2.α] Ο ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ ΣΑΝ DFSM

Η γραμματική που περιγράφει την λεκτική ανάλυση είναι κανονική, συνεπώς θα μπορούσαμε να σχεδιάσουμε ένα αιτιοκρατικό πεπερασμένο αυτόματο (Deterministic Finite State Machine) λοιπόν για να περιγράψουμε την διαδικασία. Γνωρίζουμε άλλωστε από την θεωρία υπολογισμού πως μια γλώσσα είναι κανονική άν και μόνο άν γίνεται δεκτή απο ένα πεπερασμένο αυτόματο. Ακολουθεί εικονική αναπράσταση του **`DFSM`**:



Εικόνα 2.1 : « Ο Λεκτικός Αναλυτής σαν DFSM »

ΚΑΤΑΣΤΑΣΗ S_i	ΕΙΣΟΔΟΣ e	$\delta(S_i , e)$	ΣΧΟΛΙΑ
S_0	a-z ή A-Z	S_1	*Πρόκειται είτε για αλφαριθμητικό (π.χ. ονομα μεταβλητής) Είτε για κάποια δεσμευμένη λέξη (π.χ. program).*
S_1	a-z ή A-Z ή 0-9	S_1	*Παραμένει στην κατασταση όσο βρίσκει τους χαρακτήρες Και όσο το μήκος της λέξης είναι <= 30.*
S_1	! (a-z ή A-Z ή 0-9)	$S_{0\text{-final}}$	*Πηγαίνει πίσω στην S_0 οταν δει κάτι διαφορετικό απο αυτά τα σύμβολα και τερματίζει (τελική κατάσταση).*
S_0	0-9 ή + ή -	S_2	*Πρόκειται για αριθμό.*
S_2	0-9	S_2	*Παραμένει στην κατάσταση όσο βρίσκει αριθμούς 0-9 και όσο ο αριθμός είναι μικρότερος του 3276 και μεγαλύτερος του -32767.*
S_2	!(a-z ή A-Z ή 0-9 ή + ή -)	$S_{0\text{-final}}$	*Επιστροφή στην S_0 όταν δεί κάτι διαφορετικό απο αριθμό και γράμμα (αν ειχε δει αριθμό πριν) ή οτιδήποτε αν ειχε δει + η - .*
S_0	* ή ,	S_3	*Πρόκειται είτε για πολλαπλασιασμό, είτε κόμμα, ή ίσως κάποιο σφάλμα εάν ακολουθεί / (κλείσιμο σχολίων χωρίς έναρξη).*
S_3	!(/) ή οτιδήποτε	$S_{0\text{-final}}$	*Εάν δεν ακολουθεί / και ακολουθεί οτιδήποτε άλλο πήγαινε στην τελική κατάσταση.*
S_0	/ ή ; ή = ή (ή [ή) ή]	S_4	*Εάν διαβαστεί κάτι απο αυτά τα σύμβολα προχώρα στην κατάσταση S_4 όπου πρέπει να περιμένουμε για το τί ακολουθεί.*
S_4	*	$S_{4.1}$	*Εάν ήσουν στην S_4 επειδή το σύμβολο ήταν το / και διαβάσεις * τότε πήγαινε στην $S_{4.1}$ (πρόκειται για σχόλια).*
$S_{4.1}$	! (*)	$S_{4.1}$	*Όσο δεν βρίσκεις το σύμβολο * παρέμεινε στην ίδια κατάσταση οποιοδήποτε σύμβολο και να διαβάζεις.*
$S_{4.1}$	*	$S_{4.1.1}$	*Εάν δείς το σύμβολο * πήγαινε στήν $S_{4.1.1}$ γιατί ίσως να ακολουθεί το σύμβολο / (τερματισμός σχολίων). –Κατάσταση Μνήμης*
$S_{4.1.1}$	! (*)	$S_{4.1}$	*Εάν δείς οτιδήποτε εκτός απο * τότε επέστρεψε στην κατάσταση $S_{4.1}$ διότι έχεις χάσει πλέον το σύμβολο * που είδες προηγουμένως.*
$S_{4.1.1}$	*	$S_{4.1.1}$	*Όσο βλέπεις το σύμβολο * παρέμεινε στην ίδια κατάσταση.*
$S_{4.1.1}$	/	$S_{0\text{-final}}$	*Εάν δείς το σύμβολο / τότε τα σχόλια τελειώσαν. Προχώρησε στην τελική κατάσταση.*
S_4	/	$S_{4.2}$	*Εάν ήσουν στην S_4 επειδή το σύμβολο ήταν το / και διαβάσεις / τοτε πήγαινε στην $S_{4.2}$ (πρόκειται για σχόλια μέχρι \n).*
$S_{4.2}$	! (\n)	$S_{4.2}$	*Όσο δεν βρίσκεις τον χαρακτήρα \n παρέμεινε στην ίδια κατάσταση οποιοδήποτε σύμβολο και να διαβάζεις.*
$S_{4.2}$	\n	$S_{0\text{-final}}$	*Εάν διαβάσεις το \n τότε η γραμμή τελείωσε άρα και τα σχόλια. Πήγαινε στην τελική κατάσταση.*
S_4	!(* ή /) αν είχες δεί / ή οτιδήποτε άλλο αν δεν είχες δει /	$S_{0\text{-final}}$	*Εάν είχες δεί / και το σύμβολο δεν είναι * ή / , ή , αν ήσουν στην S_4 επειδή είχες δει ; ή = ή (ή) ή [ή] και δείς οτιδήποτε, τότε πήγαινε στην τελική κατάσταση.*
S_0	:	S_5	*Εάν δείς το σύμβολο : πήγαινε στην S_5 . Μπορεί να ακολουθεί = μπορεί και όχι. –Κατάσταση μνήμης*
S_5	=	$S_{5.1}$	*Εάν δείς το σύμβολο = πρόκειται για τον τελεστή της ανάθεσης.*
$S_{5.1}$	οτιδήποτε	$S_{0\text{-final}}$	*Εάν δείς οτιδήποτε πήγαινε στην τελική κατάσταση.*
S_5	! (=)	$S_{0\text{-final}}$	*Εάν δεις οτιδήποτε εκτός απο το σύμβολο = τότε πήγαινε στην τελική κατάσταση. Πρόκειται για το σύμβολο : .*
S_0	<	S_6	*Εάν δεις το σύμβολο > πήγαινε στην κατάσταση S_6 . Υπάρχουν αρκετά ενδεχόμενα ανάλογως το σύμβολο που ακολουθεί.*
S_6	=	$S_{6.1}$	*Πρόκειται για τον τελεστή <= .*
$S_{6.1}$	οτιδήποτε	$S_{0\text{-final}}$	*Πήγαινε στην τελική κατάσταση ότι και αν διαβάσεις.*
S_6	>	$S_{6.2}$	*Πρόκαιται για τον τελεστή <> .*
$S_{6.2}$	οτιδήποτε	$S_{0\text{-final}}$	*Πήγαινε στην τελική κατάσταση ότι και αν διαβάσεις.*
S_6	! (> || =)	$S_{0\text{-final}}$	*Αν δείς οτιδήποτε εκτός απο > ή = πήγαινε στην τελική κατάσταση. Πρόκειται για τον τελεστή <.*
S_0	>	S_7	*Από την κατάσταση S_0 αν δείς το σύμβολο > πήγαινε στην S_7.*
S_7	=	$S_{7.1}$	*Εάν δεις το σύμβολο = πήγαινε στην κατάσταση $S_{7.1}$. Πρόκειται για τον τελεστή >=.*
$S_{7.1}$	οτιδήποτε	$S_{0\text{-final}}$	*Οτιδήποτε και να διαβάσεις πήγαινε στην τελική κατάσταση.*
S_7	! (=)	$S_{0\text{-final}}$	*Αν δεις οτιδήποτε εκτός απο = πήγαινε στην τελική κατάσταση. Πρόκειται για τον τελεστή >.*

Πίνακας 2.1 : « Περιγραφή της Συνάρτησης Μεταβάσεων δ του DFSM»

Η μηχανή μας περιγράφεται απο την πεντάδα $M_{EEL} = (K, \Sigma, \delta, s, F)$ όπου

$K = \{ S_0 , S_1 , S_2 , S_3 , S_4 , S_{4.1} , S_{4.1.1} , S_{4.2} , S_5 , S_{5.1} , S_6 , S_{6.1} , S_{6.2} , S_7 , S_{7.1} , S_{err}\}$
: Το πεπερασμένο σύνολο καταστάσεων της μηχανής.

$\Sigma = \{$ a-z , A-Z , 0-9 , + , - , * , / , > , < , <> , <= , >= , = , , , : , ; , := , (,) , [,] $\}$
: Το αλφάβητο της μηχανής.

$s \in K = S_0$: Η αρχική κατάσταση της μηχανής (στον πίνακα $S_{0\text{-final}} = S_0$).

$F \subseteq K = \{ S_0 \}$: Το σύνολο τελικών καταστάσεων (στον πίνακα $S_{0\text{-final}} = S_0$).

δ : Η συνάρτηση μετάβασης όπου είναι μια συνάρτηση από το $K \times \Sigma$ στο K (βλ. Πίνακα 2.1).

Παρατηρούμε πως στην περίπτωση των σχολίων, είτε μέσω των συμβόλων `/* ... */` είτε μέσω του συμβόλου `//` , ο λεκτικός αναλυτής δεν παράγει κάποια λεκτική μονάδα, αλλα επιστρέφει στην αρχική κατάσταση. Στην υλοποίηση, για να το πετύχουμε αυτό χρησιμοποιήσαμε το εργαλείο της αναδρομής. Δηλαδή σε κάθε περίπτωση όταν επρόκειτο να μεταβούμε στην αρχική κατάσταση, ο `lex()` επιστρέφει τον εαυτό του. Έτσι, παραλείπεται το κομμάτι των σχολίων και παράγεται η επόμενη λεκτική μονάδα που ζητείται απο τον συντακτικό αναλυτή.

[3] ΣΤΑΔΙΟ Β: ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Έχοντας ολοκληρώσει την λεκτική ανάλυση είμαστε έτοιμοι να ξεκινήσουμε την δημιουργία του συντακτικού αναλυτή. Ο συντακτικός αναλυτής αποτελείται από μια ομάδα συναρτήσεων. Συγκεκριμένα, για κάθε κανόνα που έχει η γραμματική μας δημιουργούμε και μία συνάρτηση η οποία αυτό που κάνει προς το παρόν, είναι να επιβεβαιώνει πως ο πηγαίος κώδικας ακολουθεί κατα γράμμα τους κανόνες μας. Στα σημεία που χρειάζεται καλεί τον λεκτικό αναλυτή για να μας προσφέρει μια λεκτική μονάδα και έπειτα ελέγχει με απλές δομές ελέγχου (**`if statements`**) εάν όντως το τερματικό σύμβολο είναι σωστό, άρα και αποδεκτό. Σε διαφορετική περίπτωση και μόνο εκεί που κρίνεται απαραίτητο, ενημερώνει τον χρήστη με μήνυμα λάθους και τερματίζει την μετάφραση.

Οι περισσότεροι κάνονες **`BNF`** περιέχουν στο σώμα τους κλήση σε άλλους κανόνες. Αυτό μεταφράζεται στην υλοποίηση ως κλήση στην αντίστοιχη συνάρτηση.

```
# -- <PROCORFUNCBODY> ::= <FORMALPARS> <BLOCK> -- #
def PROCORFUNCBODY(name):

  global line

  FORMALPARS(name)
  BLOCK(name)
```

Εικόνα 2.2 : « Παράδειγμα Υλοποίησης Κανόνα »

Η εμφάνιση μηνύματος λάθους γίνεται στην περίπτωση που ο λεκτικός αναλυτής μας παράξει κάποιο τερματικό σύμβολο το οποίο όμως δεν μπορεί να αντιστοιχιστεί με κάποιο απο αυτά που ανήκουν στην γραμματική της γλώσσας. Για παράδειγμα, έστω ο κανόνας **`<A> ::= a<B>`** και εντός της συνάρτησης **`A`** ο λεκτικός αναλυτής μας επιστρέφει τον χαρακτήρα **`b`**. Εμείς, περιμέναμε να βρούμε το σύμβολο **`a`** και συνεπώς οφείλουμε να ενημερώσουμε τον χρήστη με αντίστοιχο μήνυμα λάθους και φυσικά να τερματίσουμε την μετάφραση.

Στην πρώτη φάση, ο συντακτικός αναλυτής δεν χρειάζεται να εξοπλιστεί με περεταίρω εργαλεία. Όμως η δουλειά του είναι λιγάκι πιο σύνθετη. Όπως είδαμε και στην Εικόνα 1.1 απο αυτόν εξαρτάται η

παραγωγή του ενδιάμεσου κώδικα, η δημιουργία του πίνακα συμβόλων καθώς και η παραγωγή του τελικού κώδικα.

Αυτό που μπορούμε να εγγυηθούμε στην παρούσα φάση είναι πως ο κώδικας που έχει συνταχθεί απο τον χρήστη είναι λεκτικά και συντακτικά ορθός για την γλώσσα EEL.

[4] ΣΤΑΔΙΟ Γ: ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

Η φάση της παραγωγής ενδιάμεσου κώδικα, είναι στην ουσία ο συνδετικός κρίκος που ενώνει την ανάλυση και την σύνθεση ενός μεταφραστή. Για την γλώσσα EEL ο ενδιάμεσος κώδικας παράγεται στην μορφή τετράδων και αποθηκεύεται προσωρινά σε λίστα μέχρι την παραγωγή του τελικού κώδικα. Επίσης, όταν αυτό είναι εφικτό, παράγεται και ισοδύναμος κώδικας σε γλώσσα **ANSI C**. Εάν ο πηγαίος κώδικας περιέχει δημιουργία και κλήσεις σε διαδικασίες η συναρτήσεις τότε ο κώδικας σε **C** δεν παράγεται.

Για το στάδιο της παραγωγής του ενδιάμεσου κώδικα χρειαστήκαμε ορισμένες βοηθητικές συναρτήσεις καθώς και μια κλάση για τις τετράδες μας.

```
class Quad:

  def __init__(self, label, op, arg_1, arg_2, res):

    self.label  = label
    self.op     = op
    self.arg_1  = arg_1
    self.arg_2  = arg_2
    self.res    = res

  def __str__(self):

    return "(" + str(self.label) + ": " + str(self.op) + ", " + str(self.arg_1) + ", " \
      + str(self.arg_2) + ", " + str(self.res) + ")"
```

Εικόνα 3.1 : « Η Κλάση των Τετράδων»

Όπως βλέπουμε κάθε τετράδα είναι της μορφής **(label, op, arg1, arg2, res)**. Όπου **label** ο αύξων αριθμός της τετράδας (η αρίθμηση ξεκινά απο το 0). Τέλος κάθε αντικείμενο **Quad** έχει και μια μέθοδο **toString** η οποία χρησιμοποιήθηκε για την εύρεση λαθών καθώς και για το γράψιμο σε

αρχεία με σχετική ευκολία. Σχετικά με τις συναρτήσεις που αφορούν τον ενδιάμεσο κώδικα έχουμε τις εξής επτά (7):

- **gen_quad(op, x, y, res)**
  - **/* Δημιουργία τετράδας και τοποθέτηση στον πινακα των τετράδων */**
- **next_quad()**
  - **/* Επιστρέφει τον αριθμό του label της επόμενης τετράδας που θα δημιουργηθεί */**
- **new_temp()**
  - **/* Δημιουργεί και επιστρέφει μια νέα προσωρινή μεταβλητή */**
- **empty_list()**
  - **/* Δημιουργεί και επιστρέφει μια άδεια λίστα */**
- **make_list(item)**
  - **/* Δημιουγεί και επιστρέφει μια λίστα με το item εντός της */**
- **merge_list(list_a, list_b)**
  - **/* Ενώνει τις δύο λίστες */**
- **back_patch(quadlist, res)**
  - **/* Ενημερώνει το πεδίο res των τετράδων της quadlist */**

Στο σημείο αυτό, καλό θα ήταν να αναφερθούμε στο «λεξιλόγιο» που χρησιμοποιούμε για τις τετράδες του ενδιάμεσου κώδικα που παράγεται. Ακολουθεί αναλυτική περιγραφή (Πίνακας 4.1)

ΤΕΤΡΑΔΑ	ΣΗΜΑΣΙΑ
(“op”, “a”, “b”, “c”)	c = a op b
(“relop”, “a”, “b”, “label”)	If a relop b then goto label
(“:=”, “x”, “_”, “z”)	z = x
(“jump”,”_”,”_”,”label”)	goto label
(“begin_block”,”_”,”_”,”name”)	Αρχή συνάρτησης/διαδικασίας
(“end_block”,”_”,”_”,”name”)	Τέλος συνάρτησης/διαδικασίας
(“halt”,”_”,”_”,”_”)	Τερματισμός προγράμματος
(“par”,”x”,”mode”,”_”)	Πέρασμα παραμέτρου με τύπο mode
(“call”, “name”, “_”, “_”)	Κλήση συνάρτησης/διαδικασίας
(“ret”,”x”,”_”,”_”)	return x (για συναρτήσεις)

Πίνακας 4.1 : « Οι Τετράδες και η Σημασία τους »

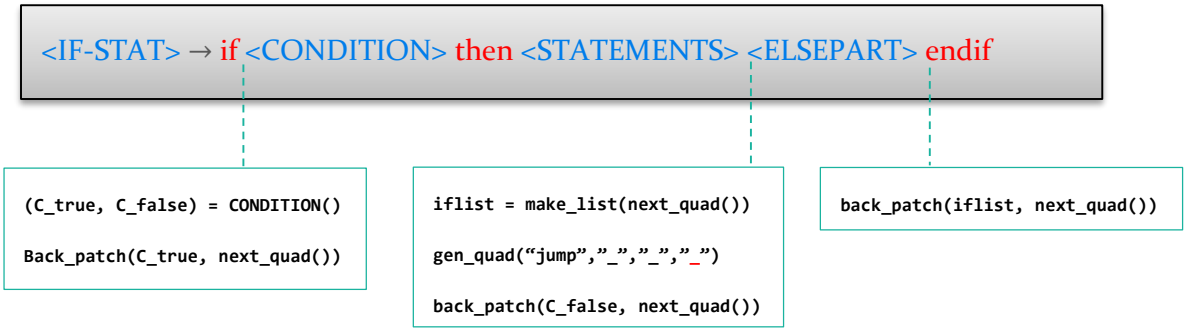
Όπου:

- op = { + , - , * , / }
- relop = { > , < , >= , <= , <>, = }
- mode = { in : περασμά με τιμή, inout : πέρασμα με αναφορά, ret : επιστρεφόμενη τιμή }

Το δυσνόητο κομμάτι της φάσης αυτής ήταν στο να κατανοήσουμε πως παράγουμε κώδικα ο οποίος θά εκτελεστεί και θα συμπληρωθεί αργότερα και όχι άμεσα. Έχοντας διασαφηνίσει το τοπίο πλέον, ξεκινάμε να τροποποιούμε κατάληλα τις ήδη υπάρχουσες συναρτήσεις του συντακτικού αναλυτή στα σημεία που κρίνεται απαραίτητο. Έτσι σε συνδιασμό και με τον

λεκτικό αναλυτή που καλείται επανειλημμένα εξοπλίζουμε τον πρώτο, με το εργαλείο της παραγωγής του ενδιάμεσου κώδικα.

Τέλος, για να γίνει καλύτερα κατανοητή η διαδικασία αυτή, παρουσιάζουμε ένα παράδειγμα παραγωγής ενδιάμεσου κώδικα (παρμένο απο τον μεταφραστή που υλοποιήσαμε). Η γλώσσα μας λοιπόν υποστηρίζει μεταξύ άλλων την εντολή ελέγχου **if**. Έχουμε:



Εικόνα 3.2 : « Παραγωγή Ενδιάμεσου Κώδικα για την Εντολή if »

Αρχικά, η λογική που έχουμε προγραμματίζοντας την συνάρτηση για να παράγει ενδιάμεσο κώδικα είναι πως πάντα οτιδήποτε χρειαζόμαστε θα το έχουμε οπωσδήποτε έτοιμο από χαμηλότερα επίπεδα. Έτσι κι εδώ, αποθηκεύουμε τις δύο λίστες που θα μας επιστρέψει ο κανόνας <CONDITION> και γνωρίζοντας πως οι τετράδες που έρχονται, έχουν κενό το πεδίο res τους, κάνουμε **back_patch** την λίστα που περιέχει τις τετράδες που έχουν αποτιμηθεί ως αληθείς, στην επόμενη τετράδα που θα δημιουργηθεί (από τον κανόνα <STATEMENTS>). Όπως γνωρίζουμε η εντολή **if** εκτελείται εάν η συνθήκη είναι αληθής. Έπειτα εντός του **if** κρατάμε σε μία λίστα (**iflist**) την επόμενη τετράδα που θα δημιουργηθεί. Η επόμενη τετράδα είναι η **("jump","_","_","_")** που όπως παρατηρούμε δεν γνωρίζουμε ακόμη τον προορισμό. Αυτός είναι και ο λόγος που αποθηκεύεται προσωρινά στην **iflist**, για να συμπληρωθεί (με την **back_patch**) αργότερα που θα γνωρίζουμε το σημείο του **jump**. Επίσης κάνουμε **back_patch** την λίστα που περιέχει τις τετράδες που έχουν αποτιμηθεί ως ψευδείς, στην τετράδα που θα δημιουργηθεί (από τον κανόνα <ELSEPART>). Τέλος, λίγο πριν το **endif**, γνωρίζουμε το μέρος στο οποίο πρέπει να συμπληρωθεί το κενό **jump** που δημιουργήσαμε προηγουμένως. Γι'αυτό εκτελούμε **back_patch(iflist,**

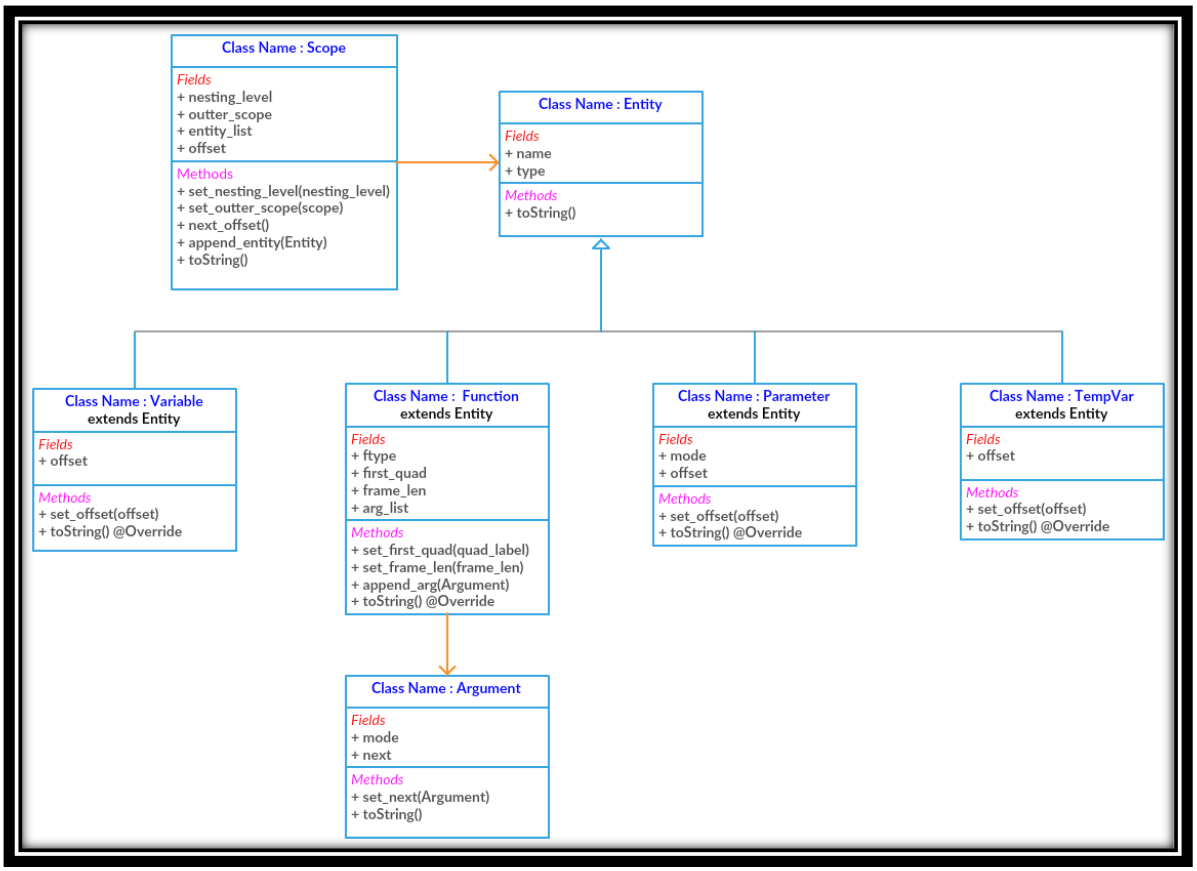
next_quad()) ώστε να παρακάμψουμε το <ELSEPART> και να βγούμε εν τέλει απο την **if**.

[5] ΣΤΑΔΙΟ Δ: ΣΧΕΔΙΑΣΜΟΣ ΠΙΝΑΚΑ ΣΥΜΒΟΛΩΝ

Κατά την διάρκεια της μετάφρασης υπάρχει η αναγκαιότητα για την συγκέντρωση πληροφοριών σχετικά με τα ονόματα (αυτά που στην γραμματική μας αναπαρίστανται ώς **id**), που εμφανίζονται εντός του κώδικα που έχει συνταχθεί απο τον χρήστη στην γλώσσα μας. Τα id αυτά, στην γλώσσα EEL μπορεί να είναι μεταβλητές (**variables**), παράμετροι σε διαδικασία ή συνάρτηση (**parameters/arguments**), σναρτήσεις ή διαδικασίες (**functions**) καθώς και προσωρινές μεταβλητές (**temporary variables**) που στην υλοποίηση μας ονομάζουμε οντότητες (**entities**). Την λύση σε αυτήν την περίπτωση δίνει ο πίνακας συμβόλων (**symbol table**). Ο πίνακας συμβόλων είναι μια δομή της οποίας το μέγεθος μεταβάλλεται - αυξομειώνεται δυναμικά κατά την μετάφραση του προγράμματος. Παρεμβάλλεται στο στάδιο παραγωγής ενδιάμεσου κώδικα και το στάδιο παραγωγής τελικού κώδικα. Εντός του βρίσκεται κάθε φορά όλη η απαραίτητη πληροφορία που χρειάζεται μία συνάρτηση ή διαδικασία.
Επίπλέον, ο πίνακας συμβόλων αποθηκεύεται σε μία λίστα η οποία καλείται **scopes** (λίστα εμβέλειας). Σαν εμβέλεια ορίζεται η δομική μονάδα του προγράμματος η οποία περιέχει δηλώσεις μιας η περισσοτέρων οντοτήτων.

Για την υλοποίηση του πίνακα συμβόλων χρησιμοποιήσαμε το εργαλείο της κληρονομικότητας που μας παρέχει μια γλώσσα αντικειμενοστρεφούς προγραμματισμού όπως είναι η **Python**. Όπως αναφέραμε στην προηγούμενη παράγραφο, τα αναγνωριστικά (**id**) είναι όλα οντότητες για το πρόγραμμα. Οι οντότητες (**entities**) όμως διαφέρουν η μία απο την άλλη (**functions/variables/parameters/tempvars**). Έτσι, δημιουργήσαμε μία μητρική κλάση την Entity, την οποία κληρονομούν οι κλάσεις **Variable**, **Function**, **Parameter** και **Tempvar**. Επίσης χρησιμοποιήσαμε δύο (2) επιπλέον κλάσεις, την κλάση **Argument** η οποία σχετίζεται με την **Function** καθώς οι οντότητες Function (συναρτήσεις/διαδικασίες) μπορεί να έχουν κάποια ορίσματα [π.χ. **function foo (in a, in b)**]. Η τελευταία κλάση είναι αυτή της εμβέλειας, που την ονομάσαμε **Scope**. Εντός της βρίσκεται

πληροφορία σχετικά με το τρέχον επίπεδο φωλιάσματος, το εξωτερικό επίπεδο φωλιάσματος και επίσης σχετικά με τις οντότητες που ανήκουν σε αυτήν. Για την καλύτερη αναπραάσταση παραθέτουμε ένα διάγραμα σε **UML** (Unified Modeling Language) που δείχνει τις εξαρτήσεις (dependencies) μεταξύ των προαναφερθέντων κλάσεων.



Εικόνα 5.1 : « Διάγραμμα UML για τον Πίνακα Συμβόλων »

Επίσης, χρειαστήκαμε και ορισμένες βοηθητικές συναρτήσεις μέσω των οποίον δομείται ουσιαστικά ο πίνακας συμβόλων και επίσης εκτελείται έλεγχος για ορισμένες παραβιάσεις περιορισμών (όπως για παράδειγμα η δήλωση δύο μεταβλητών με το ίδιο όνομα). Οι συναρτήσεις που δημιουργήσαμε είναι οι ακόλουθες δέκα (10):

- `print_scopes()`
  - `/* εκτυπώνει τον τρέχοντα πίνακα συμβόλων */`
- `duplicate_entity(entity_name, entity_type, nesting_level)`
  - `/* ελέγχει αν στο δωθέν βάθος υπάρχει οντότητα με το ίδιο δωθέν όνομα και τύπο */`
- `redefinition_of_parameter(entity_name, nesting_level)`
  - `/* ελέγχει αν στο δωθέν βάθος υπάρχει παράμετρος με το ίδιο ονομα */`
- `search_entity(entity_name, entity_type)`
  - `/* ψάχνει στον πίνακα συμβόλων και επιστρέφει την δωθήσα οντότητα`
- `add_new_scope()`
  - `/* προσθέτει νέο Scope() στον πίνακα με τις εμβέλειες scopes */`

- `add_new_variable(variable_name)`
  - `/* προσθέτει την δωθήσα μεταβλητή στο τρέχον βάθος φωλιάσματος */`
- `add_new_function(f_or_p, name)`
  - `/* προσθέτει τημ δωθήσα συνάρτηση ή διαδικασία στο τρέχον βάθος φωλιάσματος */`
- `update_fp_frame_length(name, frame_len)`
  - `/* ενημερώνει το πεδίο frame_len της δωθήσας οντότητας */`
- `update_fp_first_quad(name, first_quad_label)`
  - `/* ενημερώνει την πρώτη τετράδα της δωθήσας οντότητας */`
- `add_new_arguement_fp(fpname, mode)`
  - `/* προσθέτει νέο όρισμα στην δωθήσα οντότητα με τύπο mode */`

Στο σημείο αυτό είμαστε έτοιμοι να εξοπλήσουμε με ένα επιπλέον εργαλείο τον συντακτικό αναλυτή, τον πίνακα συμβόλων. Έτσι μένει να βρούμε τα σημεία, και συγκεκριμένα τους κανόνες στους οποίους θα γίνουν οι ενημερώσεις, προσθήκες και αφαιρέσεις στον πίνακα. Αρχικά, στην συνάρτηση PROGRAM (του κανόνα <PROGRAM> της γραμματικής) βάζουμε στον πίνακα το πρώτο scope (αυτό της main) και ενημερώνουμε το βάθος φωλιάσματος στο μηδέν. Έπειτα στην συνάρτηση BLOCK (του κανόνα <BLOCK> της γραμματικής) ενημερώνουμε τα πεδία first_quad και frame_length αφού εκτελεστούν οι κανόνες που εμπεριέχονται (DECLARATIONS, SUBPROGRAMS, STATEMENTS) για τις αντίστοιχεις συναρτήσεις/διαδικασίες. Στην περίπτωση της main τα δύο αυτά στοιχεία τα κρατάμε ξεχωριστά σε δυο μεταβλητές. Αυτό γίνεται στο συγκεκριμένο σημείο γιατί πολύ απλά, τότε μόνο είναι η στιγμή που μπορούμε να γνωρίζουμε ακριβώς αυτά τα στοιχεία μιας και στο τέλος της BLOCK έχει ολοκληρωθεί η παραγωγή ενδιάμεσου κώδικα τη συγκεκριμένη συνάρτηση/διαδικασία που εξετάζουμε. Στην συνάρτηση VARLIST (του κανόνα <VARLIST> της γραμματικής) προσθέτουμε κάθε φορά που συναντάμε ένα id μια μεταβλητή (Variable) στο τρέχον βάθος φωλιάσματος ενώ στην συνάρτηση PROCORFUNC (του κανόνα <PROCORFUNC> της γραμματικής) προσθέτουμε νέο scope στον πίνακα με τις εμβέλειες, καθώς βρήκαμε νέα συνάρτηση ή διαδικασία. Εάν αυτό που συναντάμε είναι συνάρτηση τότε στο τρέχον βάθος φωλιάσματος προσθέτουμε οντότητα Function με τύπο “ftype = function” ενώ εάν είναι διαδικασία προσθέτουμε Function με τύπο “ftype = procedure”. Τέλος στην συνάρτηση FORMALPARITEM (του κανόνα <FORMALPARITEM>) προσθέτουμε στην κατάλληλη συνάρτηση τα ορίσματα και στο τρέχον βάθος τις παραμέτρους που συναντάμε.

[6] ΣΤΑΔΙΟ Ε: ΠΑΡΑΓΩΓΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ

Η παραγωγή του τελικού κώδικα είναι το τελικό στάδιο μετάφρασης του αρχικού προγράμματος. Ο μεταφραστής μας είναι πλέον έτοιμος να συντάξει το ισοδύναμο πρόγραμμα σε γλώσσα **assembly**. Στην υλοποίηση μας ο μεταφραστής παράγει assembly για τον RISC (Reduced Instruction Set Computer) μικροεπεξεργαστή MIPS (Microprocessor without Interlocked Pipeline Stages).

Για την παραγωγή του τελικού κώδικα αυτό που γίνεται είναι η μετατροπή κάθε τετράδας που έχει παραχθεί προηγουμένως (στο στάδιο παραγωγής του ενδιάμεσου κώδικα) στην ισοδύναμη πλειάδα εντολών της **assembly** του MIPS. Αυτό που επιλέγουμε να κάνουμε στην πραγματικότητα όμως, είναι να στέλνουμε κάθε block ενδιάμεσου κώδικα για παραγωγή, αντί για όλες τις εντολές μαζεμένες. Ας παρατηρήσουμε για λιγάκι τον κανόνα <BLOCK> της γραμματικής μας. Ευθύς αμέσως θα οδηγηθούμε στο συμπέρασμα, πως με την ολοκλήρωση αυτού του κανόνα, όλες οι απαραίτητες εντολές για την εκάστοτε συνάρτηση/διαδικασία έχουν παραχθεί και βρίσκονται στην λίστα που κρατάει τις τετράδες (**code_quads**). Το μόνο που χρειαζόμαστε για την παραγωγή του τελικού κώδικα είναι η πληροφορία σχετικά με τα αναγνωριστικά (**ids**) που βρίσκονται σε αυτές τις τετράδες, κάτι που όμως έχουμε επίσης έτοιμο στον πίνακα συμβόλων που δημιουργήσαμε προηγουμένως. Έτσι λοιπόν στέλνονται για παραγωγή τετράδες που «φράσονται» άνω και κάτω από τις τετράδες **(“begin_block”, “_”, “_”, “x”) και (“end_block”,”_”,”_”,”x”)** αντίστοιχα, όπου **x** το όνομα συνάρτησης/διαδικασίας.

Ολοκληρώνοντας, και σε αυτήν την φάση, χρειαστήκαμε την βοήθεια ορισμένων βοηθητικών συναρτήσεων για την παραγωγή του τελικού κώδικα καθώς και για τον σημασιολογικό έλεγχο του αρχικού κώδικα που συντάχθηκε απο τον χρήστη. Οι βοηθητικές συναρτήσεις που χρησιμοποιήσαμε είναι οι εξής οκτώ (8):

- **search_for_variable(variable_name)**
  - **/*ψάχνει απο το παρόν βάθος και προς τα πίσω για την δωθήσα οντότητα στον πίνακα συμβόλων */**
- **gnlvcode(non_local_var)**
  - **/* ψάχνει την δωθήσα μη τοπική μεταβλητή και την αποθηκεύει στον καταχωρητή $t0 */**
- **loadvr(value,register)**
  - **/* αποθηκεύει την δωθείσα τιμή σε δωθέν καταχωρητή { $t0 … $t9 } */**
- **storerv(register,value)**

    - **/* αποθηκεύει την τιμή του δωθέν καταχωρητή στην θέση μνήμη της δωθήσας μεταβλητής */**
- **generate_assembly(code_quad, value)**
    - **/* παράγει τον ισοδύναμο κώδικα σε assembly για δωθείσα τετράρα που ανήκει σε δωθείσα συνάρτηση/διαδικασία */**
- **do_semantic_check_for_rets(name, from here)**
    - **/* εκτελεί σημασιολογικό έλεγχο για return εντολές για δωθείσα συνάρτηση/διαδικασία */**
- **do_semantic_check_for_pars(from_here)**
    - **/* εκτελεί σημασιολογικό έλεγχο για block απο τετράδες σχετικά με το πέρασμα παραμέτρων */**
- **parameter_check(fun_or_proc, actual_pars, found_pars)**
    - **/* ελέγχει εάν οι δύο δωθείσες λίστες παραμέτρων είναι ίδιες */**

[7] ΕΧΤΡΑ: ERRORS ΚΑΙ ΠΑΡΑΓΩΓΗ ΑΡΧΕΙΩΝ

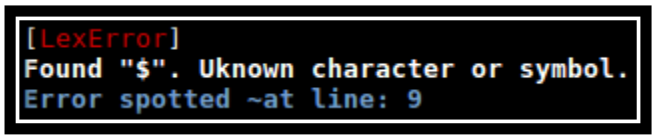
Ολοκληρώνοντας την παρουσίαση του μεταφραστή μας, χρήσιμο θα ήταν να αναφερθούμε στα μηνύματα σφαλμάτων που εμφανίζει ο μεταφραστής όταν αυτά εντοπιστούν καθώς και στον τρόπο με τον οποίο δημιουργεί συγκεκριμένα αρχεία.

Αρχικά, για την εκτύπωση μηνύματος λάθους χρησιμοποιείται απο όλες τις οντότητες του μεταφραστή, η συνάρτηση **error_display(arg, output,line)** που λειτουργεί σαν ένα μικρό **switch**. Συγκεκριμένα, αναλόγως την τιμή που έχει η παράμετρος **arg** εμφανίζει διαφορετικό τύπο μηνύματος με την παράμετρο τύπου **string**, **output** και επίσης παρέχει ενημέρωση σχετικά με το που περίπου έγινε το λάθος μέσω της παραμέτρου **line**. Για $\boldsymbol{arg} \in [\mathbf{1}, \mathbf{2}]$ το λάθος αφορά την λεκτική ανάλυση, για $\boldsymbol{arg} = \mathbf{3}$ το λάθος αφορά την συντακτική ανάλυση, για $\boldsymbol{arg} = \mathbf{4}$ το λάθος αφορά αδυναμία εύρεσης του αρχείου πηγαίου κώδικα που έγραψε ο χρήστης, για $\boldsymbol{arg} = \mathbf{5}$ λάθος που αφορά τον πίνακα συμβόλων ενώ αν $\boldsymbol{arg} = \mathbf{6}$ το λάθος αφορά την παραγωγή τελικού κώδικα.

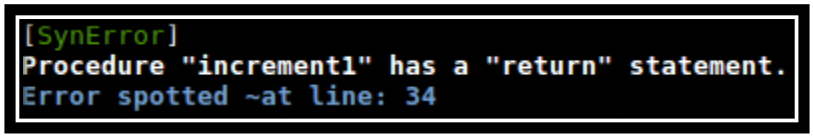
Τέλος, για την παραγωγή ενδιάμεσου κώδικα σε γλώσσα **ANSI C** χρησιμοποιούνται τέσσερις (4) συναρτήσεις. Η πρώτη είναι η $\boldsymbol{generate\_intermediate\_ansi\_c}()$ η οποία γράφει στο αρχείο ($.\boldsymbol{c}$) κάποια σχόλια σχετικά με την προέλευση του αρχείο και προσθέτει τις απαραίτητες βιβλιοθήκες που χρειάζονται. Έπειτα καλεί για κάθε μια τετράδα που έχει δημιουργηθεί την $\boldsymbol{to\_ansi\_c}(\boldsymbol{quad})$ η οποία μετατρέπει σε ισοδύναμες εντολές σε γλώσσα **ANSI C** την τετράδα **quad**. Στην περίπτωση που το quad είναι αυτό της $\boldsymbol{main}()$ ψάχνει επίσης και για τις απαραίτητες μεταβλητές που πρέπει να δηλωθούν σαν ακέραιοι ($\boldsymbol{int}$), μιας και μόνο ακεραίους υποστηρίζει η

γλώσσα μας, μέσω της ***find_declarations***() σε συνδιασμό με την ***make_declare_string***(***declare***).

Κλείνοντας, παρουσιάζουμε έναν απο κάθε τύπο μηνύματος που εμφανίζει ο μεταφραστής μας.

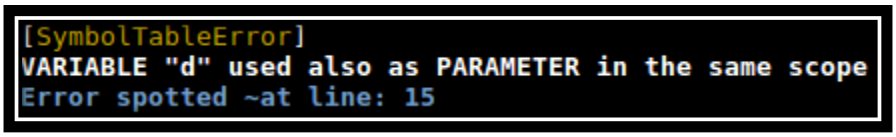


Εικόνα 7.1 : « Σφάλμα κατά την Λεκτική Ανάλυση »

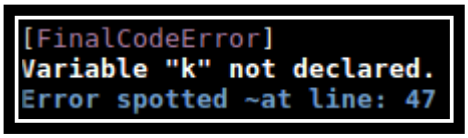


Εικόνα 7.2 : « Σημασιλογικό Σφάλμα κατά την Συντακτική Ανάλυση »

Εικόνα 7.3 : « Αδυναμία Εύρεσης Αρχείου Πηγαίου Κώδικα »



Εικόνα 7.4 : « Σημασιολογικό Σφάλμα κατά την Ενημέρωση του Πίνακα Συμβόλων »



Εικόνα 7.5 : « Σφαλμά κατά την Παραγωγή Τελικού Κώδικα