

2012

Φαίδων Μπούζμπας 1527  
Ζουμπαλίδης Έντυκαρ 1483

## [ΜΕΤΑΦΡΑΣΤΕΣ]

Τελική Αναφορά περιγραφής υλοποίησης μεταφραστή της γλώσσας Του

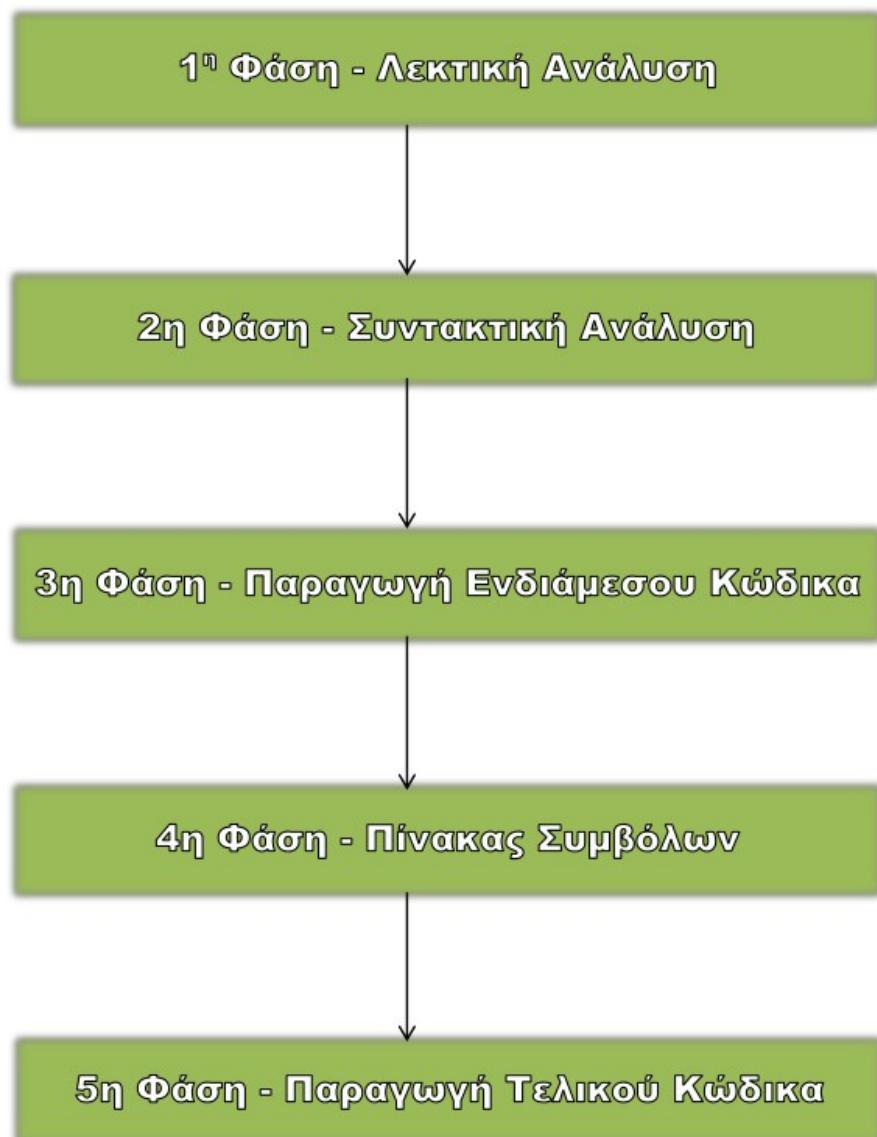
Έκδοση 1.0

## **Περιεχόμενα**

1.	Εισαγωγή στον Μεταφραστή .....	4
2.	Εισαγωγή στη γλώσσα προγραμματισμού του .....	6
2.1.	Λεκτικές Μονάδες .....	6
2.2.	Τύποι και δηλώσεις μεταβλητών .....	7
2.3.	Τελεστές και εκφράσεις .....	7
2.4.	Εντολές .....	7
2.5.	Υποπρογράμματα .....	9
2.6.	Μετάδοση παραμέτρων .....	9
2.7.	Μορφή προγράμματος .....	10
3.	Λεκτική Ανάλυση .....	11
3.1.	Λεκτικές Μονάδες .....	11
3.2.	Αυτόματο καταστάσεων .....	12
3.3.	Πίνακας Καταστάσεων .....	16
3.4.	Συναρτήσεις και μεταβλητές της Λεκτικής Ανάλυσης .....	17
4.	Συντακτική Ανάλυση .....	20
4.1.	Η γραμματική της τοу .....	20
4.2.	Συναρτήσεις της Συντακτικής Ανάλυσης .....	22
5.	Παραγωγή Ενδιάμεσου Κώδικα .....	24
5.1.	Ενδιάμεση Γλώσσα .....	24
5.2.	Συναρτήσεις και μεταβλητές της Παραγωγής Ενδιάμεσου Κώδικα .....	25
5.3.	Ενδιάμεσος Κώδικας .....	28
6.	Πίνακας Συμβόλων .....	33
6.1.	Δομή Πίνακα Συμβόλων .....	33
6.2.	Συναρτήσεις και μεταβλητές του Πίνακα Συμβόλων .....	36
7.	Παραγωγή Τελικού Κώδικα .....	42
7.1.	Η γλώσσα μηχανής .....	42
7.2.	Μετάφραση ενδιάμεσου κώδικα σε γλώσσα μηχανής .....	44
8.	Χρήση Μεταφραστή .....	52
	ΠΑΡΑΡΤΗΜΑ: Ευρετήριο συναρτήσεων και μεταβλητών .....	53

## **1. Εισαγωγή στον Μεταφραστή**

Ο μεταφραστής μας υλοποιείται στο πλαίσιο 5 φάσεων σύμφωνα με το παρακάτω διάγραμμα:



Παραλείπουμε τις φάσεις Βελτιστοποίησης Ενδιάμεσου Κώδικα και Βελτιστοποίησης Τελικού Κώδικα, ενώ η φάση της Σημασιολογικής Ανάλυσης ενσωματώνεται μέσα στις υπόλοιπες φάσεις του μεταφραστή.

Στην 1<sup>η</sup> Φάση (Λεκτική Ανάλυση), ο μεταγλωττιστής δέχεται ως είσοδο ένα αρχικό πρόγραμμα γραμμένο σε γλώσσα του με τη μορφή μίας συμβολοσειράς χαρακτήρων και δίνει ως έξοδο το ίδιο πρόγραμμα με τη μορφή μίας συμβολοσειράς λεκτικών μονάδων, οι οποίες είναι τα τερματικά σύμβολα της γραμματικής που περιγράφουν τη σύνταξη της γλώσσας του στην 2<sup>η</sup> φάση (Συντακτική Ανάλυση).

Στην 2<sup>η</sup> Φάση (Συντακτική Ανάλυση), ελέγχεται αν το αρχικό πρόγραμμα ανήκει στη γλώσσα του, της οποίας η σύνταξη ορίζεται από μία δεδομένη γραμματική χωρίς συμφραζόμενα. Η είσοδος στον συντακτικό αναλυτή είναι το αρχικό πρόγραμμα με τη μορφή μίας ακολουθίας λεκτικών μονάδων και η έξοδός του είναι το συντακτικό δέντρο που αντιστοιχεί στο πρόγραμμα που εισήχθη, ή μία ένδειξη ότι το αρχικό πρόγραμμα δεν είναι συντακτικά ορθό με έξοδο κατάλληλων διαγνωστικών μηνυμάτων.

Στην 3<sup>η</sup> Φάση (Παραγωγή Ενδιάμεσου Κώδικα), περιγράφεται η ενδιάμεση γλώσσα των τετράδων στην οποία μεταφράζονται οι δομές τις γλώσσας του, παράγοντας ένα ισοδύναμο με το αρχικό πρόγραμμα, με σκοπό την μετάφραση στην τελική γλώσσα.

Στην 4<sup>η</sup> Φάση (Πίνακας Συμβόλων), περιγράφεται το σύνολο των πληροφοριών που αποθηκεύονται στον πίνακα συμβόλων ο οποίος βοηθάει την υλοποίηση της παραγωγής τελικού κώδικα, καθώς και της σημασιολογικής ανάλυσης.

Στην 5<sup>η</sup> Φάση (Παραγωγή Τελικού Κώδικα), ο γεννήτορας τελικού κώδικα, δηλαδή το κομμάτι του μεταγλωττιστή που υλοποιεί τον τελικό κώδικα, αναλαμβάνει να μεταφράσει τις τετράδες του ενδιάμεσου κώδικα στην τελική γλώσσα, δηλαδή ένα σύνολο από εντολές assembly.

Ο μεταφραστής μας είναι υλοποιημένος σε γλώσσα C.

## 2. Εισαγωγή στη γλώσσα προγραμματισμού του

Η Τογ είναι μια μικρή γλώσσα προγραμματισμού. Παρόλο που οι προγραμματιστικές της ικανότητες είναι μικρές, η εκπαιδευτική αυτή γλώσσα περιέχει πλούσια στοιχεία και η κατασκευή του μεταγλωττιστή της έχει να παρουσιάσει αρκετό ενδιαφέρον, αφού περιέχονται σε αυτήν πολλές εντολές που χρησιμοποιούνται από άλλες γλώσσες, καθώς και κάποιες πρωτότυπες. Η Τογ περιέχει συναρτήσεις και διαδικασίες, μετάδοση παραμέτρων με αναφορά και τιμή, αναδρομικές κλήσεις, κλπ. Επίσης, επιτρέπει φώλιασμα στη δήλωση συναρτήσεων και διαδικασιών κάτι που λίγες γλώσσες υποστηρίζουν.

Από την άλλη όμως πλευρά, η Τογ δεν υποστηρίζει πραγματικούς αριθμούς και συμβολοσειρές αλλά και διάφορα άλλα, τα οποία συνήθως οι προγραμματιστές περιμένουν από μία γλώσσα προγραμματίσου υψηλού επιπέδου. Οι παραλήψεις αυτές έχουν γίνει ώστε να απλουστευτεί η διαδικασία κατασκευής του μεταγλωττιστή, μία απλούστευση όμως που έχει να κάνει μόνο με τη μείωση των γραμμών του κώδικα και όχι με τη δυσκολία κατασκευής του, αφού οι δομές που υποστηρίζει είναι αυτές που παρουσιάζουν περισσότερο ενδιαφέρον όσον αφορά την υλοποίηση.

### 2.1. Λεκτικές Μονάδες

Το αλφάβητο της Τογ αποτελείται από τα μικρά και κεφαλαία γράμματα της λατινικής αλφαριθμητικής («A»,...,«Z» και «a»,...,«z»), τα αριθμητικά ψηφία («0»,...,«9»), τα σύμβολα των αριθμητικών πράξεων («+», «-», «\*», «/»), το σύμβολο αύξησης («+=»), τους τελεστές συσχέτισης («<», «>», «=», «<=», «>=», «<>»), το σύμβολο ανάθεσης «:=», τους διαχωριστές («;», «(», «)», «,», «[», «]») καθώς και τα σύμβολα ομαδοποίησης εντολών («{», «}») και διαχωρισμού σχολίων («/\*», «\*/»). Μερικές λέξεις είναι δεσμευμένες:

and	call	else	for	if
in	inout	int	Input	not
or	print	program	return	void
while				

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

Τα αναγνωριστικά της γλώσσας είναι συμβολοσειρές που αποτελούνται από γράμματα και ψηφία, αρχίζοντας όμως από γράμμα. Ο μεταγλωττιστής λαμβάνει υπόψη του μόνο τα τριάντα πρώτα γράμματα. Οι λευκοί χαρακτήρες (tab, space, return) αγνοούνται και μπορούν να χρησιμοποιηθούν με οποιονδήποτε τρόπο χωρίς να επηρεάζεται η λειτουργία του μεταγλωττιστή, αρκεί βέβαια να μη βρίσκονται μέσα σε δεσμευμένες λέξεις (αναγνωριστικά, σταθερές κλπ.). Το ίδιο ισχύει και για τα σχόλια, τα οποία πρέπει να βρίσκονται ανάμεσα στα σύμβολα: /\* ... \*/.

## 2.2. Τύποι και δηλώσεις μεταβλητών

Ο μοναδικός τύπος δεδομένων που υποστηρίζει η Τογ είναι οι ακέραιοι αριθμοί. Οι ακέραιοι αριθμοί πρέπει να έχουν τιμές από -32767 έως 32767. Η δήλωση γίνεται με την εντολή int. Ακολουθούν τα ονόματα των αναγνωριστικών χωρίς καμία άλλη δήλωση, αφού γνωρίζουμε ότι πρόκειται για ακέραιες μεταβλητές. Οι μεταβλητές χωρίζονται μεταξύ τους με κόμματα. Το τέλος της δήλωσης των μεταβλητών γίνεται με το διαχωριστικό «;».

## 2.3. Τελεστές και εκφράσεις

Η προτεραιότητα των τελεστών από τη μεγαλύτερη στη μικρότερη είναι:

- 1) Μοναδικοί «not»
- 2) Πολλαπλασιαστικοί «\*», «/»
- 3) Μοναδικοί προσθετικοί «+», «-»
- 4) Δυαδικοί προσθετικοί «+», «-»
- 5) Σχεσιακοί «=», «<», «>», «<>», «<=», «>=»
- 6) Λογικοί «and», «or»

## 2.4. Εντολές

- Ειχώρηση

Id := expression

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

- Απόφασης

```
if (condition)
{
}
[ else
{ } ]
```

Η δομή απόφασης εκτιμάει εάν ισχύει η συνθήκη condition και εάν πράγματι ισχύει εκτελούνται οι εντολές που ακολουθούν. Το else δεν αποτελεί υποχρεωτικό τμήμα της εντολής και γι' αυτό βρίσκεται σε αγκύλη. Οι εντολές που το ακολουθούν εκτελούνται εάν η συνθήκη condition δεν ισχύει. Τα άγκιστρα δεν είναι υποχρεωτικά, όταν έχουμε μόνο μία εντολή.

- Επανάληψης με συνθήκη

```
while (condition)
{
}
```

Η δομή επανάληψης επαναλαμβάνει συνεχώς τις εντολές που βρίσκονται ανάμεσα στα άγκιστρα έως ότου η συνθήκη condition δεν ισχύει.

- Επανάληψης με μετρητή

```
for (id:=expression; condition; id+=constant)
{
}
```

Η δομή αρχικοποιεί τη μεταβλητή «id» και επαναλαμβάνει συνεχώς τις εντολές που βρίσκονται ανάμεσα στα άγκιστρα έως ότου η συνθήκη condition δεν ισχύει. Μετά από κάθε επανάληψη αυξάνεται η τιμή της μεταβλητής «id» κατά constant.

- Επιστροφής

```
return (expression)
```

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

- Εισόδου

input (variable)

Εισαγωγή τιμής από το πληκτρολόγιο.

- Εξόδου

print (expression)

Εμφανίζει την τιμή μίας expression στην οθόνη.

## 2.5. Υποπρογράμματα

Η Τού υποστηρίζει συναρτήσεις:

```
int id (formal_parameters)
{
    declaration of variables
    subprograms
    sequence of statements
}
```

ή διαδικασίες:

```
void id (formal_parameters)
{
    declaration of variables
    subprograms
    sequence of statements
}
```

Η «formal\_parameters» είναι η λίστα των τυπικών παραμέτρων. Οι διαδικασίες και οι συναρτήσεις μπορούν να φωλιάσουν η μία μέσα στην άλλη και οι κανόνες εμβέλειας είναι όπως της PASCAL. Η επιστροφή της τιμής μιας συνάρτησης γίνεται με τη return. Η κλήση μιας διαδικασίας γίνεται με την CALL.

## 2.6. Μετάδοση παραμέτρων

Η Τού υποστηρίζει δύο τρόπους μετάδοσης παραμέτρων:

- 1) με σταθερή τιμή. Δηλώνεται με τη λεκτική μονάδα `in`. Αλλαγές στην τιμή της δεν επιστρέφονται στον καλόν πρόγραμμα
- 2) με αναφορά. Δηλώνεται με τη λεκτική μονάδα `inout`. Κάθε αλλαγή στη τιμή της μεταφέρεται και στο πρόγραμμα που κάλεσε τη συνάρτηση.

## 2.7. Μορφή προγράμματος

```
program id
{
declaration of variables
subprograms
sequence of statements
}
```

\*

### 3. Λεκτική Ανάλυση

Στη φάση της λεκτικής ανάλυσης θέλουμε να σπάσουμε ένα πρόγραμμα γραμμένο σε γλώσσα του, σε λεκτικές μονάδες, δηλαδή τερματικά σύμβολα της γραμματικής της του.

#### 3.1. Λεκτικές Μονάδες

Αρχικά πρέπει να αναγνωρίσουμε ποιες είναι οι λεκτικές μονάδες της γλώσσας του.

Παρακάτω παρατίθεται ένας πίνακας όλων των λεκτικών μονάδων της γλώσσας του, όπου στην πρώτη στήλη είναι ο αύξων αριθμός, στη δεύτερη στήλη η λεκτική μονάδα και στην τρίτη στήλη η λέξη με την οποία αναφερόμαστε στην λεκτική μονάδα στον μεταφραστή μας.

A/A	Λεκτική Μονάδα	enum	A/A	Λεκτική Μονάδα	enum
1	A...Z , a...z	lettertk	24	<>	nonequaltk
2	0...9	digittk	25	:=	assigntk
3	+	plustk	26	/*	begincommenttk
4	-	minustk	27	*/	endcommenttk
5	*	multtk	28	program	programtk
6	/	slashtk	29	int	inttk
7	:	colontk	30	void	voidtk
8	=	equaltk	31	in	intk
9	<	lowertk	32	inout	inouttk
10	>	greatertk	33	call	calltk
11	;	semicolontk	34	return	returntk
12	,	commatk	35	input	inputtk
13	(	leftbrackettk	36	print	printtk
14	)	rightbrackettk	37	if	iftk
15	[	leftsquarebrackettk	38	else	elsetk
16	]	rightsquarebrackettk	39	while	whiletk
17	{	leftcurlybrackettk	40	for	forkt
18	}	rigthcurlybrackettk	41	not	nottk
19	eof	eoftk	42	and	andtk
20	other	othertk	43	or	ortk
21	+=	increasetk	44	Αριθμός	numtk
22	<=	lowerequaltk	45	Αναγνωριστικό	idtk
23	>=	greaterequaltk	46	declareint	declareinttk

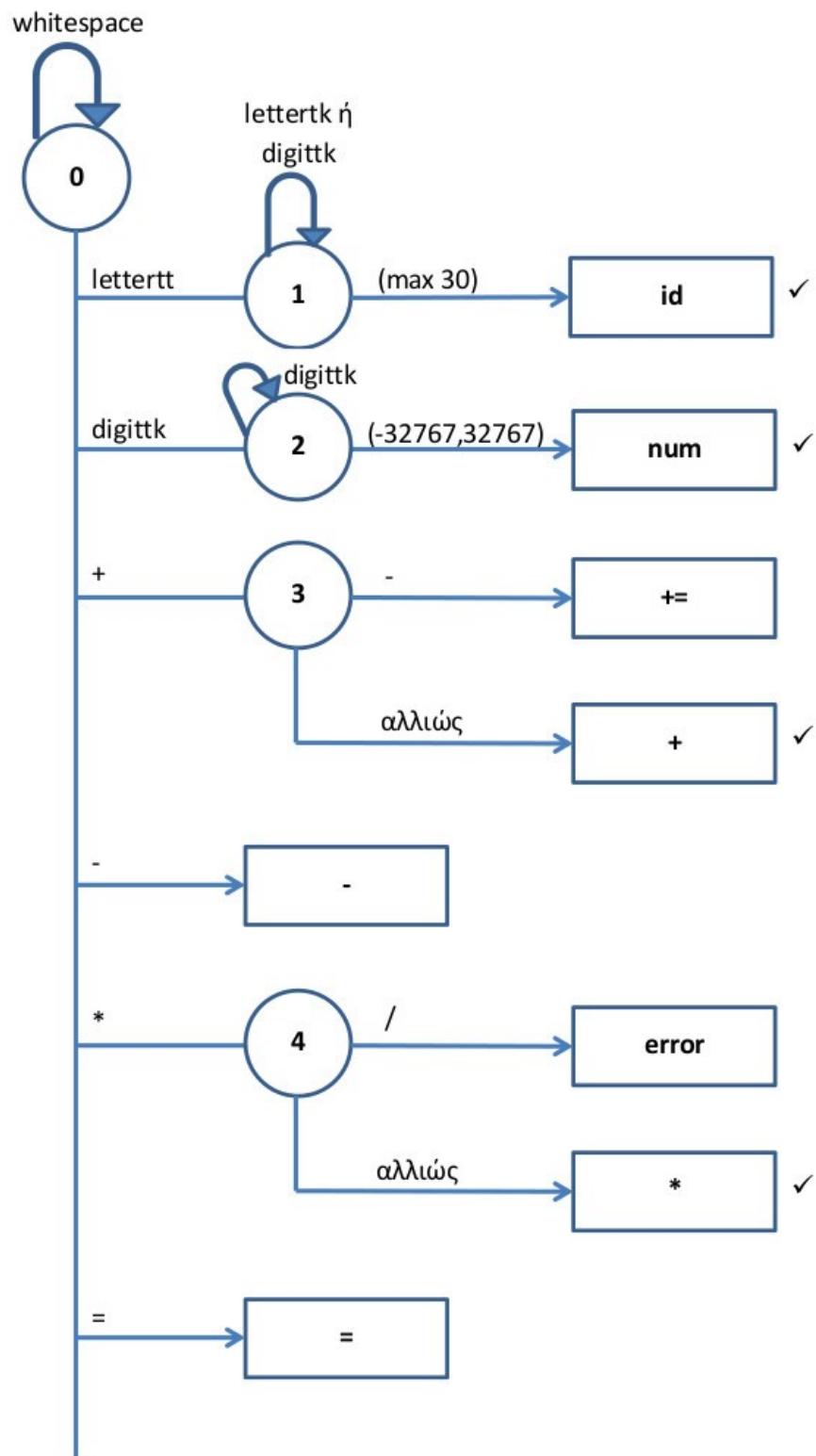
Με το «eof» εννοούμε το end of file, δηλαδή το τέλος αρχείου, με το «other» εννοούμε κάθε χαρακτήρα ο οποίος δεν ανήκει σε καμία άλλη κατηγορία του παραπάνω πίνακα, με το «Αριθμός» εννοούμε έναν αριθμό οσοδήποτε μεγάλο ή μικρό (και όχι ένα μεμονωμένο ψηφίο από 0...9), ενώ τέλος με το «Αναγνωριστικό» εννοούμε μία λέξη η οποία δεν ανήκει στο σύνολο των δεσμευμένων λέξεων που περιγράφονται στον πίνακα καθώς και στην παράγραφο 2.1. και είναι οσοδήποτε μεγάλη ή μικρή (και όχι ένα μεμονωμένο γράμμα από A...Z και a...z)

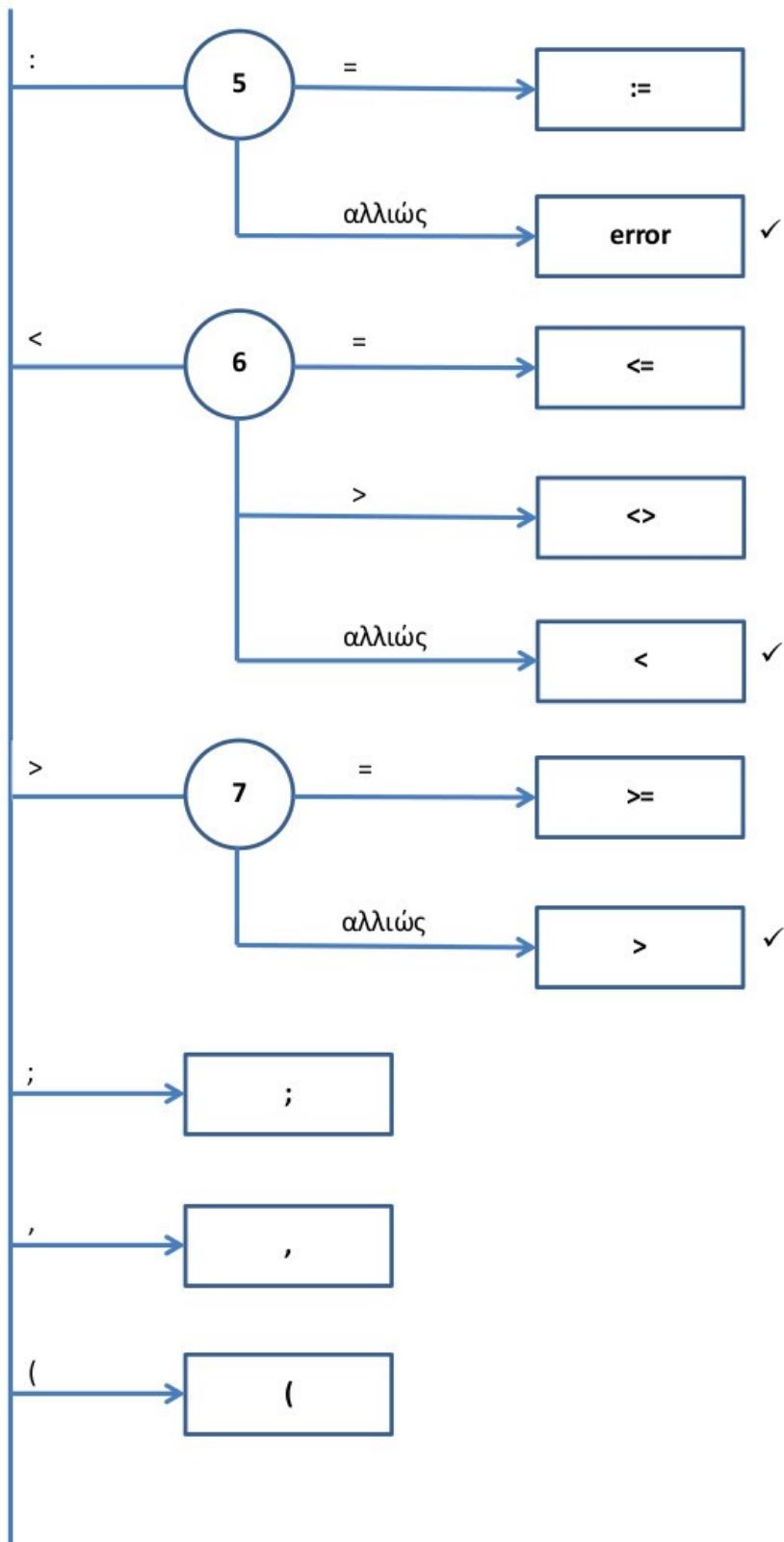
Ουσιαστικά ο πίνακας αποτελεί μέρος ενός enum και συγκεκριμένα του enum tokens\_and\_states στο πρόγραμμά μας. Κάθε λεκτική μονάδα έχει έναν μοναδικό αναγνωριστικό που περιγράφεται από την πρώτη στήλη και μπορούμε να αναφερόμαστε σε καθεμία χρησιμοποιώντας τη λέξη της τρίτης στήλης.

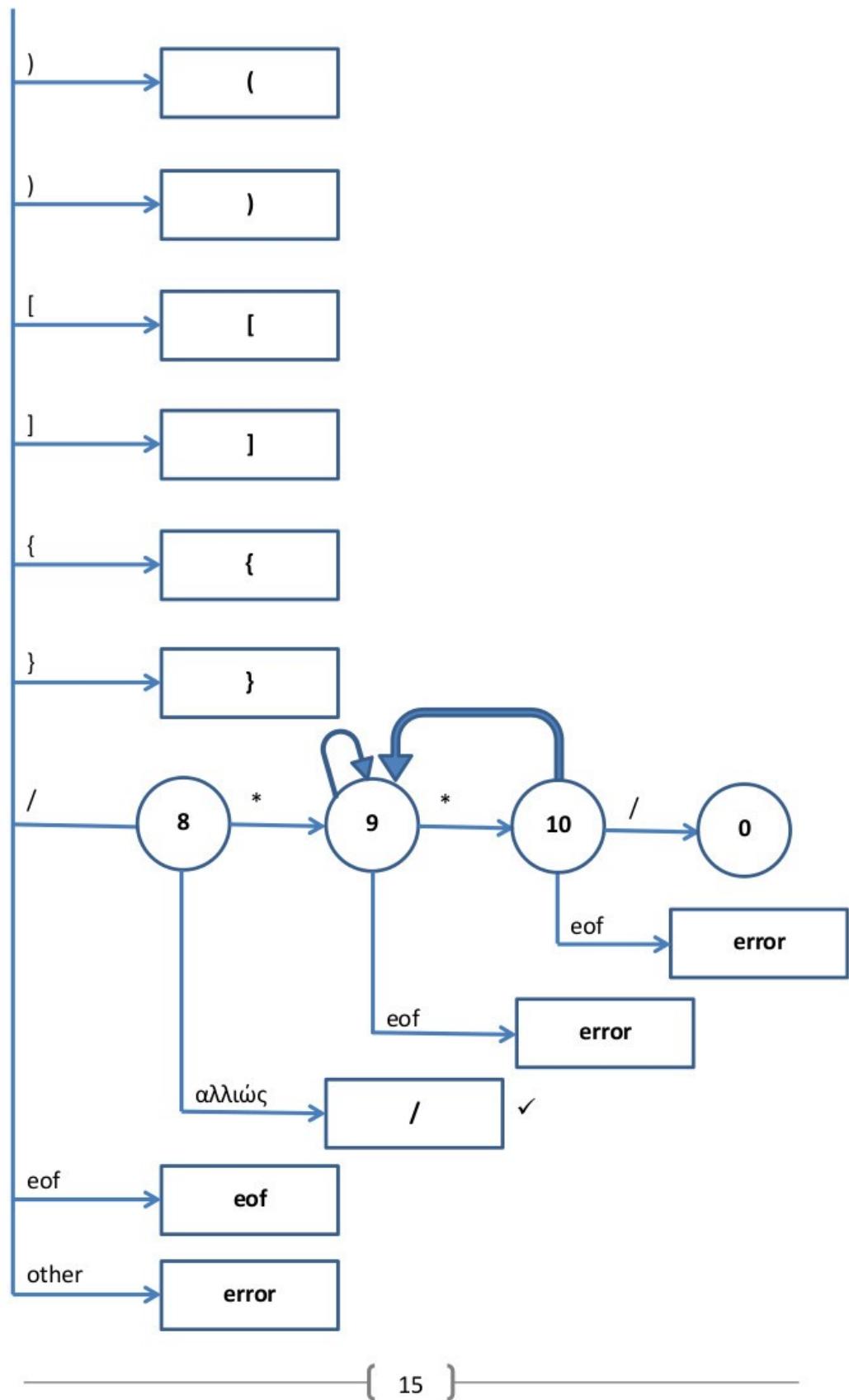
### 3.2. Αυτόματο καταστάσεων

Για την αναγνώριση των λεκτικών μονάδων, είναι απαραίτητη η κατασκευή ενός πεπερασμένου αυτόματου, το οποίο θα αναγνωρίζει δεσμευμένες λέξεις, σύμβολα της γλώσσας, αναγνωριστικά και λάθη.

Το αυτόματο που περιγράφεται στην επόμενη σελίδα αναγνωρίζει τις λεκτικές μονάδες της γλώσσας του:







Μερικές φορές για να αναγνωρισθεί μία λεκτική μονάδα καταναλώνουμε ένα χαρακτήρα από την επόμενη λεκτική μονάδα. Αυτός ο χαρακτήρας πρέπει να ενσωματωθεί στην επόμενη λεκτική μονάδα, το οποίο γίνεται με την οπισθοδρόμηση. Οι τελικές καταστάσεις στο πεπερασμένο αυτόματο που περιγράψαμε, στις οποίες γίνεται οπισθοδρόμηση σημειώνονται με το σύμβολο ✓.

### 3.3. Πίνακας Καταστάσεων

Το πεπερασμένο αυτόματο που περιγράψαμε μπορεί να υλοποιηθεί εύκολα χρησιμοποιώντας έναν πίνακα καταστάσεων, όπου στην μία του διάσταση θα έχει όλες τις πιθανές καταστάσεις και στην άλλη του διάσταση θα έχει την κάθε πιθανή είσοδο χαρακτήρα που διαβάζει κάθε φορά. Έτσι για κάθε κελί του πίνακα θα οδηγούμαστε είτε σε μία νέα κατάσταση, είτε σε μία τελική κατάσταση όπου θα αναγνωρίζουμε μία λεκτική μονάδα η οποία στο πρόγραμμά μας αναφέρεται ως «OK» (παρόλο που στον παρακάτω πίνακα καταστάσεων δίνεται η λεκτική μονάδα που αναγνωρίζεται σε κάθε περίπτωση), είτε σε ένα σφάλμα που συμβολίζεται με «error».

Παρακάτω δίνεται ο πίνακας καταστάσεων:

Είσοδοι\Καταστάσεις	letter	digit	+	-	*	/	:	=	<	>	;	,	(	)	[	]	{	}	eof	other
0	1	2	3	-	4	8	5	=	6	7	;	,	(	)	[	]	{	}	OK	error
1	1	1																	id	
2	num	2																	num	
3	+	+	+	+	+	+	+	+	+=	+	+	+	+	+	+	+	+	+	+	+
4	*	*	*	*	*	er	*	*	*	*	*	*	*	*	*	*	*	*	*	*
5			error					:=										error		
6	<	<	<	<	<	<	<	<=	<	<>	<	<	<	<	<	<	<	<	<	
7	>	>	>	>	>	>	>	>=	>	>	>	>	>	>	>	>	>	>	>	
8	/	/	/	/	9	/	/	/	/	/	/	/	/	/	/	/	/	/	/	
9	9	9	9	9	10	9	9	9	9	9	9	9	9	9	9	9	9	9	9	
10	9	9	9	9	9	0	9	9	9	9	9	9	9	9	9	9	9	9	9	

Η αναφορά στην διάσταση των εισόδων γίνεται με την χρήση της στήλης enum στον πίνακα στην παράγραφο των Λεκτικών Μονάδων 3.1. ενώ για την αναφορά στη διάσταση των καταστάσεων γίνεται πάλι με τη χρήση enum όπως περιγράφεται στον επόμενο πίνακα:

Κατάσταση	enum
0	state0=0
1	state1=1
2	state2=2
3	state3=3
4	state4=4
5	state5=5
6	state6=6
7	state7=7
8	state8=8
9	state9=9
10	state10=10

Τέλος για να αρχικοποιήσουμε κάθε κελί του πίνακα καταστάσεων είτε χρησιμοποιούμε το enum της κατάστασης αν οδηγούμαστε σε κάποια κατάσταση, είτε το enum της λεκτικής μονάδας αν οδηγούμαστε σε λεκτική μονάδα. Στην περίπτωση του «OK» και του «error» χρησιμοποιούμε εκ νέου enum σύμφωνα με τον παρακάτω πίνακα:

Κατάσταση	enum
OK	OK=-1
error	error=-2

### 3.4. Συναρτήσεις και μεταβλητές της Λεκτικής Ανάλυσης

- FILE \*fp

Καθολική μεταβλητή η οποία είναι ένας δείκτης στο αρχείο σε γλώσσα του το οποίο θέλουμε να μεταφράσουμε.

- int T[11][20]

Καθολική μεταβλητή που αποτελεί τον πίνακα μεταβάσεων.

- struct nexttoken token

Ένα καθολικό struct στο οποίο είναι αποθηκευμένη ανά πάσα στιγμή η επόμενη λεκτική μονάδα.

To struct ορίζεται ως εξής:

```

struct nexttoken{
    int type;
    char strval[82];
};

```

όπου `type` είναι μία τιμή με βάση τα enum των λεκτικών μονάδων και `strval` η αλφαριθμητική τιμή της λεκτικής μονάδας σε περίπτωση που πρόκειται για αναγνωριστικό ή αριθμό.

- `static int line=1;`

Καθολική μεταβλητή, η οποία ορίζει σε ποια γραμμή του αρχείου εισόδου βρίσκεται ο λεκτικός αναλυτής, βοηθώντας έτσι το χρήστη να διαπιστώσει σε ποια γραμμή υπάρχει σφάλμα, σε περίπτωση που η μετάφραση οδηγήσει σε λάθος.

- `void create_transition_table();`

Συνάρτηση η οποία δημιουργεί τον πίνακα μεταβάσεων και τον αποθηκεύει στον πίνακα `T[11][20]`.

- `static void lex();`

Η κύρια συνάρτηση του λεκτικού αναλυτή. Παίρνει το αρχείο και το διαβάζει χαρακτήρα προς χαρακτήρα, σύμφωνα με το πεπερασμένο αυτόματο που περιγράψαμε στην παράγραφο 3.2.. Κάθε φορά που εκτελείται διαβάζει χαρακτήρα προς χαρακτήρα το αρχείο εισόδου από το σημείο που είχε σταματήσει μέχρι να φτάσει σε τελική κατάσταση. Όταν φτάνει σε τελική κατάσταση, αν είναι έγκυρη, αποθηκεύει τις κατάλληλες τιμές στο `struct nexttoken token`, ενώ αν οδηγηθεί σε κατάσταση σφάλματος, εμφανίζει κατάλληλο διαγνωστικό μήνυμα με την γραμμή του σφάλματος και τερματίζει την εκτέλεση του μεταφραστή.

Αν φτάσει σε τελική κατάσταση και χρειαστεί οπισθοδρόμηση χρησιμοποιεί τη συνάρτηση `int ungetc (int character, FILE *stream);`

- `int check_reserved(char token[]);`

Εφόσον η συνάρτηση `lex()` βρει μία λεκτική μονάδα η οποία είναι αλφαριθμητικό, αναλαμβάνει αυτή η συνάρτηση να ελέγχει αν είναι κάποια από τις δεσμευμένες λέξεις και να την επιστρέψει, αλλιώς να επιστρέψει το αλφαριθμητικό ως αναγνωριστικό.

- void checkbuflength(int state, char buf[], int pos, int line);

Εφόσον η συνάρτηση `lex()` βρει μία λεκτική μονάδα η οποία είναι αλφαριθμητικό ή αριθμός, αναλαμβάνει αυτή η συνάρτηση να ελέγχει σε περίπτωση που είναι αλφαριθμητικό να μην έχουμε ξεπεράσει τους 30 χαρακτήρες που επιτρέπει η γλώσσα και να βγάλουμε σε διαφορετική περίπτωση κατάλληλη προειδοποίηση, ενώ αν πρόκειται για αριθμό να μην έχει βγει εκτός των επιτρεπόμενων ορίων (-32767,32767) και να βγάλουμε σε διαφορετική περίπτωση κατάλληλο σφάλμα.

- void print\_enums(int type);

Πρόκειται για βοηθητική συνάρτηση η οποία αν ενεργοποιηθεί από τη `main` (εξ' ορισμού είναι σε σχόλια) εκτυπώνει την κάθε λεκτική μονάδα που επιστρέφει ο λεκτικός αναλυτής.

Στην `main` προτού ξεκινήσει η λεκτική ανάλυση, αναζητά το αρχείο το οποίο ο χρήστης ζήτησε να μεταφραστεί. Το αρχείο το δέχεται ως όρισμα μέσα από το τερματικό και αν δεν έχει οριστεί ή δεν έχει βρεθεί τερματίζει η διαδικασία με κατάλληλο μήνυμα.

Αν το αρχείο έχει οριστεί και βρεθεί, δημιουργείται ο πίνακας καταστάσεων και στη συνέχεια όταν χρειαστεί η συντακτική ανάλυση που θα περιγράψουμε παρακάτω κάποια λεκτική μονάδα θα τρέχει τον λεκτικό αναλυτή.

## 4. Συντακτική Ανάλυση

Στη φάση της συντακτικής ανάλυσης θέλουμε να ελέγξουμε ότι το αρχικό πρόγραμμα ανήκει στη γλώσσα του με βάση τη γραμματική της την οποία θα περιγράψουμε παρακάτω.

Στο τέλος είτε θα ολοκληρωθεί επιτυχώς η συντακτική ανάλυση, είτε αν οποιοδήποτε σημείο βρούμε ότι δεν ανήκει στην γραμματική εκτυπώνεται ένα κατάλληλο μήνυμα λάθους και το πρόγραμμα τερματίζει.

### 4.1. Η γραμματική της toy

Παρακάτω παρατίθεται η γραμματική της γλώσσας του. Με έντονη επισήμανση είναι τα τερματικά σύμβολα.

```
<PROGRAM>      ::= program ID <BLOCK>
<BLOCK>        ::= { <DECLARATIONS> <SUBPROGRAMS> <SEQUENCE> }
<DECLARATIONS> ::= declareint <VARLIST> ;
<VARLIST>      ::= ε | ID ( , ID )*
<SUBPROGRAMS>  ::= ( <PROCORFUNC> ) *
<PROCORFUNC>   ::= void ID <FUNCBODY> |
                    int ID <FUNCBODY>
<FUNCBODY>      ::= <FORMALPARS> <BLOCK>
<FORMALPARS>   ::= ( <FORMALPARLIST> | ε )
<FORMALPARLIST> ::= <FORMALPARITEM> ( , <FORMALPARITEM> )*
<FORMALPARITEM> ::= in ID | inout ID
<SEQUENCE>      ::= <STATEMENT> ( ; <STATEMENT> )*
<BRACKETS_SEQ> ::= { <SEQUENCE> }
<BRACK_OR_STAT> ::= <BRACKETS-SEQ> | <STATEMENT>
```

```

<STATEMENT>      ::= ε | 
                    <ASSIGNMENT_STAT> |
                    <IF_STAT> |
                    <WHILE_STAT> |
                    <FOR_STAT> |
                    <INPUT_STAT> |
                    <PRINT_STAT> |
                    <CALL_STAT> |
                    <RETURN_STAT>

<ASSIGNMENT_STAT> ::= ID := <EXPRESSION>

<IF_STAT>        ::= if (<CONDITION>) <BRACK_OR_STAT> <ELSEPART>
<ELSEPART>       ::= ε | else <BRACK_OR_STAT>
<WHILE_STAT>     ::= while (<CONDITION>) <BRACK_OR_STAT>
<FOR_STAT>       ::= for (ID:=<EXPRESSION>; <CONDITION> ;
                           ID+=<OPTIONAL_SIGN> CONSTANT) <BRACK_OR_STAT>

<INPUT_STAT>     ::= input (ID)
<PRINT_STAT>     ::= print (<EXPRESSION>)
<CALL_STAT>      ::= call ID <ACTUALPARS>
<ACTUALPARS>     ::= ( <ACTUALPARLIST> | ε ) | ε
<ACTUALPARLIST>  ::= <ACTUALPARITEM> ( , <ACTUALPARITEM> )*
<ACTUALPARITEM>  ::= in <EXPRESSION> | inout ID
<RETURN_STAT>    ::= return (<EXPRESSION>)
<CONDITION>      ::= <BOOLETERM> (or <BOOLETERM>)*
<BOOLETERM>      ::= <BOOLFATOR> (and <BOOLFATOR>)*
<BOOLFATOR>      ::= not [<CONDITION>] | [<CONDITION>] |
                           <EXPRESSION> <RELATIONAL_OPER> <EXPRESSION>
<EXPRESSION>      ::= <OPTIONAL_SIGN> <TERM> ( <ADD_OPER> <TERM> )*
<TERM>            ::= <FACTOR> ( <MUL_OPER> <FACTOR> )*

```

```

<FACTOR>      ::= CONSTANT | (<EXPRESSION>) | ID <ACTUALPARS>
<RELATIONAL_OPER> ::= = | < ( ε | = | > ) | > ( ε | = )
<ADD_OPER>     ::= + | -
<MUL_OPER>     ::= * | /
<OPTIONAL_SIGN> ::= ε | <ADD_OPER>

```

## 4.2. Συναρτήσεις της Συντακτικής Ανάλυσης

Ουσιαστικά με το που καλέσουμε το συντακτικό αναλυτή, αυτός οδηγείται από τη γραμματική που περιγράφηκε παραπάνω, ώστε να αποφανθεί αν είναι συντακτικά ορθό το πρόγραμμα ή έχει λάθο. Το κάθε αριστερό μέρος της γραμματικής είναι και μία αυτοτελής συνάρτηση, που περιγράφει τα βήματα της συντακτικής ανάλυσης του δεξιού της μέρους. Αν δεν οδηγηθεί σε ένα τερματικό σύμβολο, θα οδηγείται σε ένα μη τερματικό σύμβολο που είναι πάλι μία συνάρτηση της γραμματικής. Κάθε φορά που θα βρίσκει ένα τερματικό σύμβολο θα καλεί τη συνάρτηση `lex()` η οποία θα φέρει την επόμενη λεκτική μονάδα.

- `syntax () ;`

Στη `main` με το που ανοίξουμε ένα αρχείο τογ καλείται η συνάρτηση που θα μας οδηγήσει στη συντακτική ανάλυση. Καλείται η `lex()` η οποία φέρνει την επόμενη λεκτική μονάδα και μπαίνει στην πρώτη συνάρτηση της συντακτικής ανάλυσης, δηλαδή την `program()`.

Οι επόμενες συναρτήσεις ουσιαστικά πρόκειται για το αριστερό κομμάτι της γραμματικής της `toy`, οπότε αναφέρονται απλά ονομαστικά (όσες συναρτήσεις περιέχουν ορίσματα, αυτά θα χρειαστούν σε επόμενες φάσεις):

- `void syntax();`
- `void program();`
- `void block(char *, quadlist *);`
- `void declarations();`
- `void varlist();`
- `void subprograms();`
- `void procorfunc();`
- `void funcbody(char *);`

- void formalpars();
- void formalparlist();
- void formalparitem();
- void sequence();
- void brackets\_seq();
- void brack\_or\_stat();
- void statement();
- void assignment\_stat();
- void if\_stat();
- void elsepart();
- void while\_stat();
- void for\_stat();
- void input\_stat();
- void print\_stat();
- void call\_stat();
- void actualpars(char \*, char \*);
- void actualparlist();
- parlist \* actualparitem(parlist \*);
- void return\_stat();
- void condition(tflist \*);
- void boolterm(tflist \*);
- void boolfactor(tflist \*);
- void expression(char \*);
- void term(char \*);
- void factor(char \*);
- void relational\_oper(char \*);
- void add\_oper();
- void mul\_oper();
- void optional\_sign(char \*);

## 5. Παραγωγή Ενδιάμεσου Κώδικα

Στη φάση της παραγωγής ενδιάμεσου κώδικα θέλουμε να μεταφράσουμε το αρχικό πρόγραμμα σε ένα ισοδύναμο πρόγραμμα γραμμένο σε μία ενδιάμεση γλώσσα, ώστε στη συνέχεια να μεταφραστεί στην τελική γλώσσα.

### 5.1. Ενδιάμεση Γλώσσα

Η ενδιάμεση γλώσσα είναι ένα σύνολο από τετράδες οι οποίες περιγράφουν όλες τις δομές και εντολές της γλώσσας *toy* και αποτελούνται από έναν τελεστή και τρία τελούμενα.

Παρακάτω παρατίθενται όλες οι τετράδες τις ενδιάμεσης γλώσσας:

- Assign:                    $:=, \ x, \ _, \ z$   
Εκχώρηση του *x* στο *z*, δηλαδή  $z := x$
- Unconditional jump:    $\text{jump}, \ _, \ _, \ z$   
*jump* στην εντολή με αριθμό *z*
- Conditional jump:      $\text{relop}, \ x, \ y, \ z$   
Το *relop* αντιστοιχεί σε έναν τελεστή σύγκρισης, δηλαδή  $<,>,<,>,<=,>=$  και αν η σύγκριση *x* *relop* *y* ισχύει κάνε *jump* στην εντολή με αριθμό *z*, αλλιώς προχώρα παρακάτω
- Σημείωση μεταβλητής  
για συνάρτηση:        $\text{par}, \ x, \ m, \ _$   
*x* είναι μία μεταβλητή που περνάει σαν παράμετρος στην κλήση μίας συνάρτησης, ενώ *m* είναι ο τρόπος περάσματος, δηλαδή CV (με τιμή), REF (με αναφορά) και RET (επιστροφή)
- Είσοδος δεδομένων:    $\text{input}, \ _, \ _, \ z$   
Δέχεται μία είσοδο από το πληκτρολόγιο και την εκχωρεί στην μεταβλητή *z*
- Έξοδος δεδομένων:    $\text{print}, \ _, \ _, \ z$   
Εκτυπώνει στην οθόνη την τιμή της μεταβλητής *z*
- Κλήση συνάρτησης:    $\text{call}, \ \text{name}, \ _, \ _$   
Κλήση της συνάρτησης με όνομα *name*

- Επιστροφή από συνάρτηση: `ret, _, _, z`  
Επιστρέφει από την κληθείσα συνάρτηση στη καλούσα την τιμή z και τερματίζεται η κληθείσα συνάρτηση
- Αρχή προγράμματος  
ή υποπρογράμματος: `begin_block, name, _, _`  
Δημιουργείται όταν ξεκινάει ένα πρόγραμμα ή ένα υποπρόγραμμα με όνομα name
- Τέλος προγράμματος  
ή υποπρογράμματος: `end_block, name, _, _`  
Δημιουργείται όταν τελειώνει ένα πρόγραμμα ή ένα υποπρόγραμμα με όνομα name
- Τέλος προγράμματος: `halt, _, _, _`  
Δημιουργείται όταν τελειώνει ο κώδικας του κυρίως προγράμματος

## 5.2. Συναρτήσεις και μεταβλητές της Παραγωγής Ενδιάμεσου Κώδικα

- `quad *aquad=NULL;`

Για να αποθηκεύσουμε κάθε μία τετράδα δημιουργούμε μία συνδεδεμένη λίστα όπου πρώτο στοιχείο είναι αυτό ο δείκτης στο struct quad, όπου αυτή η δομή έχει την εξής μορφή:

```
typedef struct linkedquad{
    char label[40];
    char op[40];
    char op1[40];
    char op2[40];
    char op3[40];
    struct linkedquad *next;
}quad;
```

και αποθηκεύει τον αύξον αριθμό της ετικέτας, τα τέσσερα κομμάτια της τετράδας, καθώς και την επόμενη τετράδα.

- `quad *lastquad=NULL;`

Δείχνει στην τελευταία τετράδα της λίστας τετράδων.

- `char NextQuad[7] = "99";`

Δείχνει από ποια είναι η ετικέτα της τετράδας που θα δημιουργηθεί, ώστε να αποθηκευτεί στο struct \*quad.

- `int temp_cnt=0;`

Στη διάρκεια του προγράμματος είναι πιθανό να χρειαστεί να δημιουργηθούν προσωρινές μεταβλητές. Αυτές ορίζονται ως `T_x`, όπου `x` ένας αριθμός που μας τον δίνει αυτή η μεταβλητή.

- `char temp_var[5];`

Περιέχει το όνομα μίας προσωρινής μεταβλητής που έχουμε δημιουργήσει.

- `typedef struct listofquads{  
 quad *link;  
 struct listofquads *next;  
}quadlist;`

Στη διάρκεια της παραγωγής ενδιάμεσου κώδικα θα χρειαστεί να σημειώσουμε κάποιες τετράδες, καθώς δεν έχουμε αρκετή πληροφορία για να τις συμπληρώσουμε κατάλληλα τη στιγμή της δημιουργίας τους. Μέσω αυτής της δομής μπορούμε να δημιουργήσουμε μία συνδεδεμένη λίστα (τη λίστα ετικετών τετράδων), όπου ο κάθε κόμβος δείχνει σε κάποια συγκεκριμένη τετράδα, ώστε όταν έχουμε την απαραίτητη πληροφορία για τις σημειωμένες τετράδες να τις συμπληρώσουμε.

- `typedef struct actualparlist{  
 char name[40];  
 char method[5];  
 struct actualparlist *next;  
}parlist;`

Μία δομή για να αποθηκεύει όλες τις παραμέτρους που χρησιμοποιούμε στην κλήση μίας συνάρτησης, η οποία είναι απαραίτητη για ειδικές περιπτώσεις συναρτήσεων μέσα σε συναρτήσεις (πχ. `max(in (max(in a, in b)), in (max(in c, in d)))` ).

- `void nextquad();`

Ενημερώνει τη μεταβλητή `char NextQuad[7]` ποιος είναι ο αριθμός της επόμενης ετικέτας.

- `void genquad(char *a, char *b, char *c, char *d);`

Δημιουργεί μία νέα τετράδα και την τοποθετεί στο τέλος της συνδεδεμένης λίστας τετράδων.

- `void newTemp();`

Αυξάνει τη μεταβλητή `temp_cnt` κατά ένα και αποθηκεύει στη μεταβλητή `temp_var` μία νέα προσωρινή μεταβλητή `T_temp_cnt`.

- `quadlist * emptylist();`

Δημιουργεί μία κενή λίστα ετικετών τετράδων και την επιστρέφει.

- `quadlist * makelist(char *label);`

Δημιουργεί μία λίστα ετικετών τετράδων με έναν κόμβο που δείχνει σε μία τετράδα με ετικέτα ίση με `label`. Στο τέλος επιστρέφει τη λίστα ετικετών τετράδων.

- `quadlist * merge(quadlist *list1, quadlist *list2);`

Δέχεται ως όρισμα δύο λίστες ετικετών τετράδων και ενώνει την δεύτερη στο τέλος της πρώτης. Στο τέλος επιστρέφει την πρώτη λίστα που περιέχει και την δεύτερη.

- `void backpatch(quadlist *list, char *x);`

Δέχεται ως όρισμα μία λίστα ετικετών τετράδων και για κάθε κόμβο της πηγαίνει στην αντίστοιχη τετράδα στην οποία δείχνει και κάνει τον τέταρτο στοιχείο της τετράδας ίσο με το όρισμα `x`. Με αυτόν τον τρόπο συμπληρώνουμε τις απαραίτητες πληροφορίες σε τετράδες που είχαμε σημειώσει στη λίστα ετικετών τετράδων επειδή δεν είχαμε αρκετές πληροφορίες όταν αυτές είχαν παραχθεί.

- `void print_quadlist();`

Βοηθητική συνάρτηση, η οποία καλείται από την `main` και εκτυπώνει όλη τη λίστα τετράδων.

### 5.3. Ενδιάμεσος Κώδικας

Για τις περισσότερες συναρτήσεις την συντακτικής ανάλυσης θα πρέπει να τροποποιηθούν, προσθέτοντας σε αυτές κομμάτια κώδικα (που συμβολίζονται ως ετικέτες {pX}, τα οποία αποτελούνται από κατάλληλες χρήσεις των συναρτήσεων που περιγράφηκαν παραπάνω για τη δημιουργία όλων των τετράδων.

- `<PROGRAM> ::= program ID {p1} <BLOCK>`

{p1}: `programlist=makelist(nextquad());  
genquad("jump","","","","");`

- `<BLOCK> ::= { <DECLARATIONS> <SUBPROGRAMS> {p1}  
<SEQUENCE> {p2} }`

{p1}: `genquad("begin_block",blockname,"","",");  
if(programlist!=NULL)  
 backpatch(programlist,nextquad());`

{p2}: `if(programlist!=NULL)  
 genquad("halt","","","",");  
genquad("end_block",blockname,"","",");`

- `<ASSIGNMENT_STAT> ::= ID := <EXPRESSION> {p1}`

{p1}: `genquad(":=",Eplace,"",idplace);`

- `<IF_STAT> ::= if (<CONDITION>) {p1} <BRACK_OR_STAT> {p2}  
<ELSEPART> {p3}`

{p1}: `backpatch(condtrue,nextquad());`

{p2}: `iflist=makelist(nextquad());  
genquad("jump","","","",");`

{p3}: `backpatch(condfalse,NextQuad);`

- <WHILE\_STAT> ::= **while** {p1} (<CONDITION>) {p2}  
                   <BRACK\_OR\_STAT> {p3}
- {p1}: condquad=nextquad();  
 {p2}: backpatch(condtrue,nextquad());  
 {p3}: genquad("jump","","","",condquad);  
       backpatch(condfalse,nextquad());
- 
- <FOR\_STAT> ::= **for** (ID:=<EXPRESSION>; {p1} <CONDITION> ;  
                   {p2} ID+=<OPTIONAL\_SIGN> {p3} CONSTANT) {p4}  
                   <BRACK\_OR\_STAT> {p5}
- {p1}: genquad(":",Eplace,"","",idplace);  
       condquad=nextquad();  
 {p2}: assignquad=nextquad();  
 {p3}: genquad("+",idplace,Tplace,idplace);  
 {p4}: genquad("jump","","","",condquad);  
       backpatch(condtrue,nextquad());  
 {p5}: backpatch(condfalse,nextquad());
- 
- <INPUT\_STAT> ::= **input** (ID) {p1}
- {p1}: genquad("input","","","",idplace);
- 
- <PRINT\_STAT> ::= **print** (<EXPRESSION>) {p1}
- {p1}: genquad("print","","","",Eplace);
- 
- <ACTUALPARS> ::= ( <ACTUALPARLIST> | ε ) {p1} | ε
- {p1}: genquad("par",w,"RET","");  
       genquad("call",idplace,"","",");

- <ACTUALPARLIST> ::= <ACTUALPARITEM> ( , <ACTUALPARITEM>  
                          )\* {p1}

{p1}: while(parlist!=NULL)  
      genquad("par",par\_name,par\_method,"");

- <ACTUALPARITEM> ::= in <EXPRESSION> {p1} | inout ID {p2}

{p1}: par\_name=Eplace;  
      par\_method=CV;

{p2}: par\_name=Eplace;  
      par\_method=REF;

- <RETURN\_STAT> ::= return (<EXPRESSION>) {p1}

{p1}: genquad("ret","","","",Eplace);

- <CONDITION> ::= <BOOLTERM> {p1} (or {p2} <BOOLTERM> {p3}  
                          )\*

{p1}: Btrue=Q1true;  
      Bfalse=Q1false;

{p2}: backpatch(Q1false,nextquad());

{p3}: Btrue=merge(Btrue,Q2true);  
      Bfalse=Q2false;

- <BOOLTERM> ::= <BOOLFACTOR> {p1} (and {p2} <BOOLFACTOR>  
                          {p3} )\*

{p1}: Qtrue=R1true;  
      Qfalse=R1false;

{p2}: backpatch(Q1true,nextquad());

{p3}: Qfalse=merge(Qfalse,R2false);  
      Qtrue=R2true;

- <BOOLFACTOR> ::= **not** [<CONDITION>] **{p1}** | [<CONDITION>] **{p2}** | <EXPRESSION> <RELATIONAL\_OPER> <EXPRESSION> **{p3}**

{p1}: Rtrue=Bfalse;  
Rfalse=Btrue;

{p2}: Rtrue=Btrue;  
Rfalse=Bfalse;

{p3}: Rtrue=makelist(nextquad());  
genquad(relop,E1place,E2place,"");  
Rfalse=makelist(nextquad());  
genquad("jump","","","","");

- <EXPRESSION> ::= <OPTIONAL\_SIGN> <TERM> ( <ADD\_OPER> <TERM> **{p1}** )\* **{p2}**

{p1}: w=newTemp();  
if(oper=="+")  
 genquad("+",T1place,T2place,w);  
else if(oper=="-")  
 genquad("-",T1place,T2place,w);  
T1place=w;

{p2}: Eplace=T1place;

- <TERM> ::= <FACTOR> (<MUL\_OPER> <FACTOR> **{p1}** )\* **{p2}**

{p1}: w=newTemp();  
if(oper=="\*")  
 genquad("\*",T1place,T2place,w);  
else if(oper=="/")  
 genquad("/",T1place,T2place,w);  
F1place=w;

{p2}: Tplace=F1place;

- <FACTOR> ::= {p1} CONSTANT | (<EXPRESSION>) {p2} | {p3}  
ID {p4} <ACTUALPARS>

{p1}: Fplace=CONSTANT;

{p2}: Fplace=Eplace;

{p3}: idplace=ID;

{p4}: if (ACTUALPARS==ε)  
Fplace=idplace;

- <RELATIONAL\_OPER> ::= = {p1} | < ( ε {p2} | = {p3} | > {p4}  
) | > ( ε {p5} | = {p6} )

{p1}: relop="=";

{p2}: relop "<" ;

{p3}: relop "<=" ;

{p4}: relop "<>" ;

{p5}: relop ">" ;

{p6}: relop ">=" ;

- <OPTIONAL\_SIGN> ::= ε | <ADD\_OPER> {p1}

{p1}: if (oper=="-")

Tplace="-";

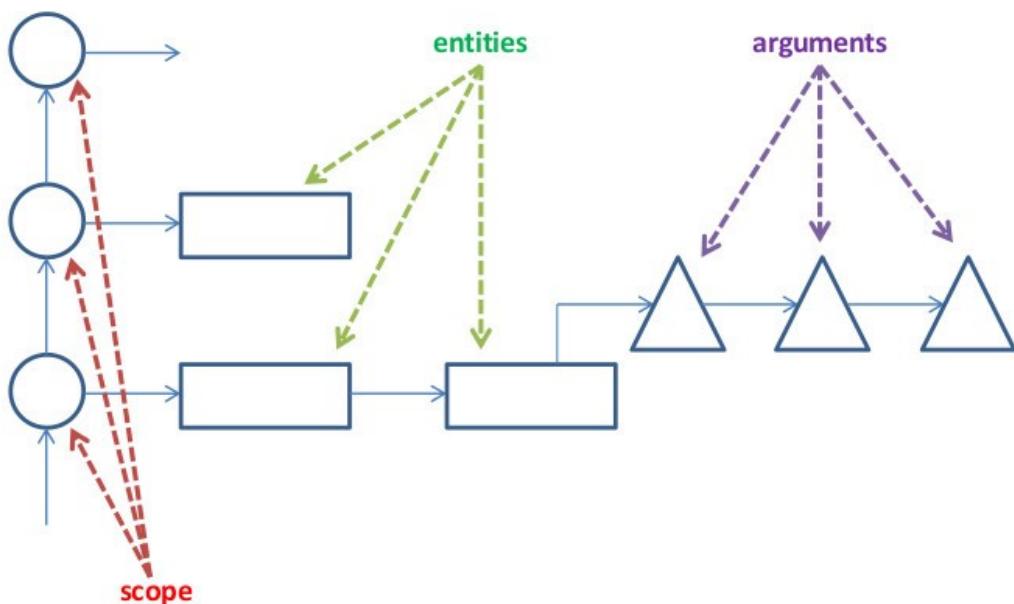
Σε κάθε περίπτωση τα programlist, iflist, condtrue, condfalse, Btrue, Bfalse, Qtrue, Qfalse, Q1true, Q1false, Q2false, Rtrue, Rfalse, R1true, R1false είναι δομές τύπου \*quadlist, τα Eplace, Tplace, T1place, Fplace, F1place, w, idplace, condquad, assignquad, relop, oper είναι τύπου char, ενώ το parlist είναι μία δομή parlist με τα αντίστοιχα πεδία par\_name και par\_method που αντιστοιχούν στα name και method που είναι τύπου char.

## 6. Πίνακας Συμβόλων

Στη φάση του πίνακα συμβόλων θέλουμε να δημιουργήσουμε μία δομή στην οποία θα αποθηκεύουμε πληροφορίες για τα ονόματα που εμφανίζονται στο αρχικό πρόγραμμα ώστε ο μεταγλωττιστής να τα χρησιμοποιεί κατά τη διάρκεια της μεταγλώττισης. Με την έννοια ονόματα αναφερόμαστε στο ίδιο το πρόγραμμα, μεταβλητές, υποπρογράμματα όπως συναρτήσεις και διαδικασίες, παράμετροι υποπρογραμμάτων, τύποι δεδομένων, ετικέτες εντολών και μήκος εγγραφήματος δραστηριοποίησης.

### 6.1. Δομή Πίνακα Συμβόλων

Στην δομή του πίνακα συμβόλων θέλουμε ουσιαστικά να υλοποιήσουμε το παρακάτω σχήμα:



**scope:** Πρόκειται για την εμβέλεια στην οποία βρισκόμαστε (πχ. στην main, ή σε κάποια συνάρτηση).

Πεδία:

- Όνομα scope

- Επίπεδο φωλιάσματος (πόσο απέχει από την main)
- Δείκτης σε πρώτο entity
- Δείκτης σε επόμενο scope

**entity:** Πρόκειται για κάποιο όνομα που βρίσκεται στο συγκεκριμένο scope του entity (παρακάτω εξηγείται ο τύπος του κάθε ονόματος)

Πεδία:

- Όνομα entity (το όνομα που έχει μέσα στο πρόγραμμα)
- Τύπος entity (μεταβλητή, συνάρτηση, προσωρινή μεταβλητή, παράμετρος)

**Μεταβλητή:**

- offset (Απόσταση από την αρχή του scope που βρίσκεται σε bytes – 4 bytes για κάθε μεταβλητή)

**Συνάρτηση:**

- Τύπος (Συνάρτηση ή διαδικασία)
- Δείκτης στο πρώτο argument
- Μήκος εγγραφήματος δραστηριοποίησης (το μήκος όλου του scope σε bytes)
- Ετικέτα πρώτης εκτελέσιμης τετράδας (σε ποια ετικέτα του ενδιάμεσου κώδικα θα κάνει jump ώστε να εκτελέσει την κώδικα της συνάρτησης)

**Προσωρινή μεταβλητή:**

- offset

**Παράμετρος:**

- offset
- Τρόπος περάσματος (με τιμή – CV ή με αναφορά – REF)

**argument:** Πρόκειται για τις παραμέτρους της συνάρτησης η οποία τα περιέχει.

Πεδία:

- Όνομα argument (το όνομα που έχει μέσα στο πρόγραμμα)
- Μέθοδος περάσματος (με τιμή – CV ή με αναφορά – REF)
- Δείκτης σε επόμενο argument

Παρατηρήσεις:

- Το offset κάθε entity (εκτός από τις συναρτήσεις που δεν έχουν) ορίζεται ως η απόσταση αυτού του entity από την αρχή του scope στο οποίο βρίσκεται, αλλά

με μικρότερη τιμή του 12 και μήκος πάντα ίσο με 4 bytes. Δηλαδή το πρώτο entity που έχει offset θα έχει 12 bytes, το επόμενο 16 bytes κοκ. Ο λόγος που ξεκινάμε από τα 12 bytes είναι γιατί χρειαζόμαστε 12 bytes για κάποιες ειδικές πληροφορίες του εγγραφήματος δραστηριοποίησης.

- Ως εγγράφημα δραστηριοποίησης για ένα scope ορίζουμε το κομμάτι της μνήμης που αποθηκεύει όλες τις πληροφορίες παραμέτρων, αποτελεσμάτων, μεταβλητών και συναρτήσεων για αυτό το scope. Το framelength (μήκος εγγραφήματος δραστηριοποίησης) είναι το μέγεθος του εγγραφήματος δραστηριοποίησης σε bytes. Το εγγράφημα δραστηριοποίησης δηλαδή έχει την εξής μορφή:

4 bytes		διεύθυνση επιστροφής συνάρτησης όταν αυτή τελειώσει
4		σύνδεσμος προσπέλασης (διεύθυνση στοίβας του πατέρα)
4		διεύθυνση στην οποία θα επιστρέψει το αποτέλεσμα
4 για κάθε παράμ.	παράμετροι (με τη σειρά εμφανίσεως αυτών)	
4 για κάθε μεταβλ.	τοπικές μεταβλητές	
4 για κάθε προσωρ.	προσωρινές μεταβλητές	

- Για να περιγράψουμε τον τύπο του entity, τη μέθοδο περάσματος παραμέτρου, τον τύπο συνάρτησης (συνάρτηση ή διαδικασία) και τη μέθοδο περάσματος argument χρησιμοποιούμε enum με βάση τον παρακάτω πίνακα:

Κατάσταση	enum
variable	enumvar
function	enumfunc
tempvar	enumtempvar
parameter	enumpar
CV	enumCV
REF	enumREF
procedure	enumprocedure
function	enumfunction
in	enumin
inout	enuminout

## 6.2. Συναρτήσεις και μεταβλητές του Πίνακα Συμβόλων

- scope \*ascope=NULL;

Καθολική μεταβλητή που είναι ένας δείκτης τύπου scope που δείχνει στο πρώτο scope που δημιουργείται (δηλαδή την main).

Η δομή scope ορίζεται ως εξής:

```
typedef struct new_scope{
    char scope_name[31];
    int nestingLevel;

    struct new_entity *first_ent;
    struct new_scope *next_scope;
}scope;
```

Για τα entities χρησιμοποιείται η παρακάτω δομή:

```
typedef struct new_entity{
    char ent_name[40];
    int type;
    union{
        struct new_variable *avar;
        struct new_function *afunc;
        struct new_tempvar *atempvar;
        struct new_parameter *apar;
    }ent_type;
    struct new_entity *next_ent;
}entity;
```

Για κάθε τύπο entity υπάρχει και μία διαφορετική δομή.

- Για μεταβλητή:

```
typedef struct new_variable{
    int offset;
}variable;
```

- Για συνάρτηση:

```
typedef struct new_function{
    int type;
    char starting_quad[7];
    int framelength;
```

```
    struct new_argument *first_arg;
} function;
```

- Για προσωρινή μεταβλητή:

```
typedef struct new_tempvar{
    int offset;
} tempvar;
```

- Για παράμετρο:

```
typedef struct new_parameter{
    int offset;
    int method;
} parameter;
```

Για τα arguments χρησιμοποιείται η παρακάτω δομή:

```
typedef struct new_argument{
    char arg_name[40];
    int method;
    struct new_argument *next_arg;
} argument;
```

- var\_list \*hlp\_var\_list=NULL;

Μία καθολική μεταβλητή που είναι ένας δείκτης τύπου var\_list που δείχνει σε μία λίστα η οποία μας βοηθάει να τοποθετήσουμε τις μεταβλητές του προγράμματος κατά τα δήλωσή τους στα entities.

Η δομή var\_list ορίζεται ως εξής:

```
typedef struct varq{
    char var_name[40];
    struct varq *next;
} var_list;
```

- par\_list \*hlp\_par\_list=NULL;

Μία καθολική μεταβλητή που είναι ένας δείκτης τύπου par\_list που δείχνει σε μία λίστα η οποία μας βοηθάει να τοποθετήσουμε τις παραμέτρους του προγράμματος στα entities τους.

Η δομή par\_list ορίζεται ως εξής:

```

typedef struct parq{
    char par_name[40];
    int par_method;
    struct parq *next;
} par_list;

```

- `par_list *hlp_arg_list=NULL;`

Μία καθολική μεταβλητή που είναι ένας δείκτης τύπου `par_list` που δείχνει σε μία λίστα η οποία μας βοηθάει να τοποθετήσουμε στα arguments μίας συνάρτησης τον τρόπο περάσματος του κάθε arguments.

- `int Level=0;`

Καθολική μεταβλητή που δείχνει το επίπεδο φωλιάσματος στο οποίο βρισκόμαστε. Εξ ορισμού επίπεδο φωλιάσματος ίσο με ένα έχει ή main και αυξάνεται όσο μπαίνουμε σε πιο βαθύ επίπεδο.

- `int entoff=12;`

Το offset στο οποίο βρισκόμαστε για τα entities. Αρχικά είναι ίσο με 12, αφού εκεί μπορεί να μπει το πρώτο offset ενός entity.

- `int argoff=12;`

Αντίστοιχα το offset στο οποίο βρισκόμαστε για τα arguments.

- `scope * add_new_scope(scope *rootscope, char *sc_name);`

Προσθέτει ένα νέο scope στο τέλος της λίστας των scope, δίνοντας του ως όνομα το `sc_name`, αυξάνοντας και το επίπεδο φωλιάσματος.

- `scope * add_new_entity(scope *rootscope, char *name, int entity_type);`

Προσθέτει ένα νέο entity στο τελευταίο scope, δίνοντας του ως όνομα το `name` και δημιουργώντας την κατάλληλη δομή με βάση τον τύπο του entity (πχ. αν πρόκειται για προσωρινή μεταβλητή θα δημιουργήσει ένα `atempvar struct`) με τη βοήθεια συναρτήσεων που περιγράφονται παρακάτω.

- `scope * add_new_argument(scope *rootscope, char *name, int type);`

Προσθέτει ένα νέο argument στο τελευταίο entity (που πρόκειται για είδος συνάρτησης) στο τελευταίο scope και του δίνει ως όνομα το name και ως μέθοδο περάσματος το type.

- `variable * add_new_variable();`

Σε περίπτωση που προσθέτουμε ένα entity τύπου μεταβλητής καλούμε αυτή τη συνάρτηση για να δημιουργήσει ένα τέτοιο πεδίο.

- `function * add_new_function(int type);`

Σε περίπτωση που προσθέτουμε ένα entity τύπου συνάρτησης καλούμε αυτή τη συνάρτηση για να δημιουργήσει ένα τέτοιο πεδίο.

- `tempvar * add_new_tempvar();`

Σε περίπτωση που προσθέτουμε ένα entity τύπου προσωρινής μεταβλητής καλούμε αυτή τη συνάρτηση για να δημιουργήσει ένα τέτοιο πεδίο.

- `parameter * add_new_parameter(int type);`

Σε περίπτωση που προσθέτουμε ένα entity τύπου παραμέτρου καλούμε αυτή τη συνάρτηση για να δημιουργήσει ένα τέτοιο πεδίο.

- `var_list * add_var_to_list(var_list *root, char *name);`

Προσθέτει σε μία στο τέλος μία λίστας var\_list μία μεταβλητή με όνομα name. Με αυτόν τον τρόπο δημιουργεί μία λίστα με όλες τις μεταβλητές που δηλώνονται σε μία συνάρτηση ή στη main.

- `par_list * add_par_to_list(par_list *root, char *name, int method);`

Προσθέτει στο τέλος μίας λίστας par\_list μία παράμετρο με όνομα name και μέθοδο περάσματος method. Με αυτόν τον τρόπο δημιουργεί μία λίστα με όλες τις παραμέτρους μίας συνάρτησης που πρέπει να προστεθούν στο scope της.

- `scope * add_help_var_list_to_scope(scope *rootscope, var_list *rootvarscope);`

Προσθέτει στο τελευταίο scope μία λίστα var\_list ως entities με τις κατάλληλες πληροφορίες. Ουσιαστικά παίρνει τις μεταβλητές που έχουν δηλωθεί στη συνάρτηση ή στη main και για κάθε μία προσθέτει ένα entity μεταβλητής καλώντας την αντίστοιχη συνάρτηση.

- `scope * add_help_par_list_to_scope(scope *rootscope, par_list *rootparscope);`

Προσθέτει στο τελευταίο scope μία λίστα `par_list` ως `entities` με τις κατάλληλες πληροφορίες. Ουσιαστικά παίρνει τις παραμέτρους μίας συνάρτησης που έχει κληθεί και για κάθε μία προσθέτει ένα `entity` παραμέτρου με την κατάλληλη μέθοδο περάσματος καλώντας την αντίστοιχη συνάρτηση.

- `scope * add_help_arg_list_to_scope(scope *rootscope, par_list *rootparscope);`

Προσθέτει στο τελευταίο `entity` του τελευταίου `scope` (που είναι τύπου συνάρτηση) μία λίστα `par_list` ως `arguments` με τις κατάλληλες πληροφορίες. Ουσιαστικά παίρνει τις παραμέτρους τις συνάρτησης που έχει κληθεί και για κάθε μία προσθέτει ένα `argument` με την κατάλληλη μέθοδο περάσματος καλώντας την αντίστοιχη συνάρτηση.

- `entity *assign_starting_quad(entity *anentity, char *sq);`

Προσθέτει στο `entity` `anentity` την ετικέτα της πρώτης εκτελέσιμης τετράδας, δηλαδή την `sq`. Προφανώς και αυτό το `entity` είναι τύπου συνάρτησης και έτσι ορίζουμε σε ποια τετράδα θα πρέπει να πάει το πρόγραμμα ώστε να εκτελεστεί ο κώδικας αυτής της συνάρτησης που κλήθηκε.

- `entity *assign_framelength(scope *rootscope, entity *anentity);`

Βρίσκει το `framelength` του προτελευταίου `scope` μετρώντας το `offset` του και το εικωρεί στο `entity` `anentity` που προφανώς είναι τύπου συνάρτησης.

- `entity *searchentity(scope *rootscope, char *name);`

Αναζητά αναδρομικά το `entity` με όνομα `name` και επιστρέφει ένα δείκτη σε αυτό αν βρεθεί, διαφορετικά την τιμή `NULL`. Αυτό σημαίνει ότι θα αρχίσει να αναζητά από το τελευταίο `scope` και θα γυρνάει προς το πρώτο. Στον πίνακα συμβόλων χρειαζόμαστε αυτή τη συνάρτηση ώστε να βρούμε σε ποιο `entity` θα πρέπει να οριστεί το `framelength`.

- `scope * delete_last_scope(scope *rootscope);`

Διαγράφει το τελευταίο `scope` από τη λίστα και μειώνει και το επίπεδο φωλιάσματος στο οποίο βρισκόμαστε. Η συνάρτηση καλείται κάθε φορά που

τελειώνει η ανάλυση της συνάρτησης ή της main στις οποίας το scope βρισκόμασταν.

- `void print_symbol_table(scope *rootscope);`

Βοηθητική συνάρτηση η οποία εκτυπώνει αναδρομικά την εικόνα του πίνακα συμβόλων τη στιγμή που την καλούμε. Αυτό σημαίνει ότι εκτυπώνει πρώτα το τελευταίο scope και μετά προχωράει προς το πρώτο. Στο μεταφραστή μας έχουμε επιλέξει να την εκτελούμε πριν από το τέλος και τη διαγραφή κάθε scope.

Οι συναρτήσεις που περιγράψαμε καλούνται μέσα από τις συναρτήσεις της συντακτικής ανάλυσης στα σημεία στα οποία κρίνεται απαραίτητο ώστε να δημιουργηθεί ο πίνακας συμβόλων.

## 7. Παραγωγή Τελικού Κώδικα

Στη φάση της παραγωγής τελικού κώδικα παίρνουμε τη λίστα τετράδων που δημιουργήσαμε στον σπηλαίο της παραγωγής του ενδιάμεσου κώδικα και την μεταφράζουμε σε κατάλληλες εντολές γλώσσας μηχανής τις οποίες αποθηκεύουμε σε ένα αρχείο το οποίο μπορεί να εκτελεστεί ώστε να τρέξει το πρόγραμμα που έχει φτιαχτεί σε γλώσσα του.

### 7.1. Η γλώσσα μηχανής

Η γλώσσα μηχανής έχει 255 καταχωρητές που συμβολίζονται ως R[0], R[1], R[2], ..., R[255].

Ως stack pointer (SP) θεωρούμε τον R[0], δηλαδή ένας δείκτης που δείχνει στην αρχή της στοίβας του εγγραφήματος δραστηριοποίησης για τη συνάρτηση που μεταφράζουμε κάθε φορά.

Ως program counter (PC) θεωρούμε το σύμβολο ‘\$', δηλαδή έναν μετρητή που έχει αποθηκευμένο τον αριθμό της επόμενης εντολής στην οποία βρισκόμαστε.

Παράλληλα η γλώσσα μηχανής μπορεί να κάνει πρόσβαση και σε θέσεις τις μνήμης ως εξής: M[address]. Η έκφραση μπορεί επίσης να περιέχει και τη διεύθυνση κάποιου καταχωρητή για να γίνει πρόσβαση σε κάποια θέση μνήμης: M[R[255]].

Παρακάτω δίνονται οι εντολές της γλώσσας μηχανής:

- ini tr ,όπου tr ένας καταχωρητής

Διαβάζει έναν ακέραιο που πατήθηκε από το πληκτρολόγιο και τον τοποθετεί στον καταχωρητή tr.

- outi so1 ,όπου so1 ένας καταχωρητής

Εκτυπώνει στην οθόνη το περιεχόμενο του καταχωρητή so1.

- addi tr,so1,so2

Εκτελεί την εκχώρηση  $tr := so1 + so2$

- subi tr,so1,so2

Εκτελεί την εκχώρηση  $tr := so1 - so2$

- $muli tr, so1, so2$

Εκτελεί την εκχώρηση  $tr := so1 * so2$

- $divi tr, so1, so2$

Εκτελεί την εκχώρηση  $tr := so1 / so2$

Σε όλες τις περιπτώσεις των τεσσάρων βασικών πράξεων τα  $tr$ ,  $so1$  και  $so2$  ήταν καταχωρητές

- $movi tr, so1$

Εκτελεί την εκχώρηση  $tr := so1$

Στην περίπτωση αυτή η αντιστοιχία  $tr$  και  $so1$  είναι η εξής:

tr	so1
μνήμη	καταχωρητής
καταχωρητής	μνήμη
καταχωρητής	αριθμός
καταχωρητής	R[0]

Εν ολίγοις απαγορεύεται και ο  $tr$  και ο  $so1$  να είναι πρόσβαση σε μνήμη.

- $jmp addr$  ,όπου  $addr$  είναι ένα label(ετικέτα), καταχωρητής, μνήμη ή αριθμός

Κάνει άλμα στη θέση  $addr$ .

- $cmpi so1, so2$  ,όπου  $so1$  και  $so2$  είναι καταχωρητές

Συγκρίνει τα  $so1$  και  $so2$  και ενεργοποιεί ένα από τα παρακάτω flags:

jb Av  $so1 > so2$

jbe Av  $so1 \geq so2$

ja Av  $so1 < so2$

jae Av  $so1 \leq so2$

je Av  $so1 = so2$

jne Av  $so1 \neq so2$

- condjmp w ,όπου το condjmp={je,jne,jb,jbe,ja,jae} και w ένα label(ετικέτα)

Ελέγχει αν το flag condjmp είναι ενεργοποιημένο και αν είναι εκτελεί το άλμα υπό συνθήκη στην ετικέτα w.

## 7.2. Μετάφραση ενδιάμεσου κώδικα σε γλώσσα μηχανής

Για να μεταφράσουμε τη λίστα τετράδων του ενδιάμεσου κώδικα σε γλώσσα μηχανής χρειαζόμαστε για κάθε περίπτωση τετράδας ορισμένες εντολές σε γλώσσα μηχανής οι οποίες και περιγράφονται παρακάτω.

Αρχικά θα πρέπει να παραθέσουμε τη μορφή που θα έχει το αρχείο της γλώσσας μηχανής. Για κάθε τετράδα κώδικα που θα μεταφράζεται, θα δημιουργείται μία ετικέτα Lx: και θα ακολουθούν οι εντολές σε γλώσσα μηχανής που αντιστοιχούν στην τετράδα του ενδιάμεσου κώδικα. Για παράδειγμα:

```
L10: movi R[255],R[0]
      movi R[254], offset
```

- Μεταφορά στον R[255] μίας μη τοπικής μεταβλητής

Ο κώδικας για αυτή τη μεταφορά βρίσκεται σε μία συνάρτηση που έχουμε δημιουργήσει, την gnlvcode().

```
movi R[255], M[4+R[0]]
for (μέχρι να φτάσουμε στη συνάρτηση η οποία κάλεσε τη συνάρτηση στην οποία βρισκόμαστε)
    movi R[255], M[4+R[255]]
    movi R[254], offset
    addi R[255], R[254], R[255]
```

- Φόρτωση μίας μεταβλητής ή αριθμού σε έναν καταχωρητή

Ο κώδικας για αυτή τη φόρτωση βρίσκεται σε μία συνάρτηση που έχουμε δημιουργήσει, την loadvr(v,r), όπου v μία μεταβλητή και r ένας αριθμός.

- αν v είναι αριθμός:
 

```
      movi R[r], v
```
- αν v είναι global μεταβλητή - δηλαδή του κυρίως προγράμματος:
 

```
      movi R[r], M[600+offset]
```

- αν v τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον ή προσωρινή μεταβλητή:  
`movi R[r], M[offset+R[0]]`
  
  - αν v τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον:  
`movi R[255], M[offset+R[0]]`  
`movi R[r], M[R[255]]`
  
  - αν v τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον:  
`gnlvcode()`  
`movi R[r], M[R[255]]`
  
  - αν v τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:  
`gnlvcode()`  
`movi R[255], M[R[255]]`  
`movi R[r], M[R[255]]`
- Αποθήκευση μίας μεταβλητής σε μία θέση μνήμης
- Ο κώδικας για αυτή τη φόρτωση βρίσκεται σε μία συνάρτηση που έχουμε δημιουργήσει, την `storerv(r,v)`, όπου r ένας αριθμός και v μία μεταβλητή.
- αν v είναι global μεταβλητή - δηλαδή του κυρίως προγράμματος:  
`movi M[600+offset], R[r]`
  
  - αν v τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος ίσο με το τρέχον:  
`movi M[offset+R[0]], R[r]`
  
  - αν v τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο με το τρέχον:  
`movi R[255], M[offset+R[0]]`  
`movi M[R[255]], R[r]`
  
  - αν v τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή και βάθος φωλιάσματος μικρότερο από το τρέχον ή προσωρινή μεταβλητή:  
`gnlvcode()`  
`movi M[R[255]], R[r]`

- αν ν τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος μικρότερο από το τρέχον:

```
gnlvcode()
movi R[255], M[R[255]]
movi M[R[255]], R[r]
```

- Άλμα χωρίς συνθήκη – jump,\_,\_w

```
jmp Lw
```

- Άλμα υπό συνθήκη – relop,x,y,w

```
loadvr(x, 1)
loadvr(y, 2)
cmpi R[1], R[2]
condjmp Lw
όπου comdjmp={je,jne,jb,jbe,ja,jae}
```

- Εικώρηση – :=,x,\_,z

```
loadvr(x, 1)
storerv(1, z)
```

- Αριθμητικές πράξεις – op,x,y,z όπου op={+,-,\*,/}

```
loadvr(x, 1)
loadvr(y, 2)
oper R[3], R[1], R[2]
storerv(3, z)
όπου oper={addi,subi,muli,divi}
```

- Εντολή εισόδου – input,\_,\_,x

```
ini R[1]
storerv(1, x)
```

- Εντολή εξόδου – print,\_,\_,x

```
loadvr(x, 1)
outi R[1]
```

- Επιστροφή από συνάρτηση – ret,\_,\_,x

```
loadvr(x, 1)
movi R[255], M[8+R[0]]
```

```
movi M[R[255]],R[1]
```

- Παράμετροι συνάρτησης
  - Με τιμή – par,x,CV,\_  
loadvr(x,255)  
movi M[d+R[0]], R[255]  
  
όπου d=framelength συνάρτησης που βρισκόμαστε + 12 + 4 \* (i-1) και  
i=αύξοντας αριθμός παραμέτρου συνάρτησης (αρχικά i=1)
  - Με αναφορά – par,x,REF,\_
    - Αν η καλούσα και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος και η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
movi R[255],R[0]  
movi R[254],offset  
addi R[255], R[254],R[255]  
movi M[d+R[0]], R[255]  
  
όπου offset, το offset της μεταβλητής στον πίνακα συμβόλων.
    - Αν η καλούσα και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος και η παράμετρος x έχει περαστεί στην καλούσα συνάρτηση με αναφορά:  
movi R[255],R[0]  
movi R[254],offset  
addi R[255], R[254],R[255]  
movi R[1], M[R[255]]  
movi M[d+R[0]], R[1]  
  
όπου offset, το offset της μεταβλητής στον πίνακα συμβόλων.
    - Αν η καλούσα και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος και η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
gnlvcode()  
movi M[d+R[0]], R[255]
    - Αν η καλούσα και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος και η παράμετρος x έχει περαστεί στην καλούσα συνάρτηση με αναφορά:  
gnlvcode()

```
movi R[1], M[R[255]]  
movi M[d+R[0]], R[1]
```

όπου  $d=framelength$  συνάρτησης που βρισκόμαστε  $+12 + 4 * (i-1)$   
και  $i=$ αύξοντας αριθμός παραμέτρου συνάρτησης (αρχικά  $i=1$ )

- Επιστροφή – `par,x,RET,_`

```
movi R[255], R[0]  
movi R[254], offset  
addi R[255], R[254], R[255]  
movi M[framelength+8+R[0]], R[255]
```

όπου  $offset$ , το  $offset$  της μεταβλητής στον πίνακα συμβόλων και  $framelength=framelength$  της συνάρτησης που βρισκόμαστε.

- Κλήση συνάρτησης – `call,x,_,_`

- Αν η καλούσα και η κληθείσα έχουν το ίδιο βάθος φωλιάσματος (καλούσα και κληθείσα έχουν τον ίδιο πατέρα):

```
movi R[255], M[4+R[0]]  
movi M[framelength+4+R[0]], R[255]
```

όπου  $framelength=framelength$  της συνάρτησης που βρισκόμαστε.

- Αν καλείται συνάρτηση με μεγαλύτερο βάθος φωλιάσματος από την καλούσα (η καλούσα είναι ο πατέρας της κληθείσας):

```
movi M[framelength+4+R[0]], R[0]
```

όπου  $framelength=framelength$  της συνάρτησης που βρισκόμαστε.

Στη συνέχεια και για τις δύο περιπτώσεις εκτελούμε τις παρακάτω εντολές:

```
movi R[255], framelength  
addi R[0], R[255], R[0]  
movi R[255], $  
movi R[254], 15  
addi R[255], R[255], R[254]  
movi M[R[0]], R[255]  
jmp Lstart_quad
```

όπου  $framelength=framelength$  της συνάρτησης που βρισκόμαστε και  $Lstart\_quad$  η ετικέτα πρώτης εκτελέσιμης τετράδας για τη συνάρτηση την οποία καλούμε την οποία την βρίσκουμε στον πίνακα συμβόλων.

- Επιστροφή στην καλούσα συνάρτηση

```
movi R[255], framelength  
subi R[0], R[0], R[255]
```

όπου framelength=framelength της συνάρτησης που βρισκόμαστε.

- Τέλος block – end\_block,x,\_,\_

```
jmp M[R[0]]
```

- Αρχή προγράμματος

```
jmp L0
```

δηλαδή άλμα στην πρώτη εκτελέσιμη εντολή που βρίσκεται στην ετικέτα L0. Η ετικέτα αυτή δεν έχει ακόμα δημιουργηθεί στο εκτελέσιμο αρχείο, αλλά όταν βρεθεί θα γραφτεί ως L0 και όχι με τον αύξοντα αριθμό ετικέτας που θα τις αναλογούσε ώστε το πρόγραμμα να κάνει άλμα σε αυτήν με το που ξεκινήσει.

- Τερματισμός προγράμματος – halt,\_,\_,\_

```
halt
```

Όλες αυτές οι εντολές γράφονται σε ένα αρχείο εξόδου το οποίο έχει ίδιο όνομα με το αρχικό αρχείο εισόδου σε γλώσσα του με την διαφορά ότι έχει κατάληξη .out .

### 7.3. Συναρτήσεις και μεταβλητές της Παραγωγής Τελικού Κώδικα

- FILE \*finalcode;

Καθολική μεταβλητή η οποία είναι ένας δείκτης στο αρχείο με κατάληξη .out στο οποίο θα αποθηκεύσουμε τις εντολές του τελικού κώδικα.

- quad \*tmpquad=NULL;

Καθολική μεταβλητή που ορίζει ένα δείκτη τύπου quad που δείχνει στην τετράδα από την λίστα τετράδων για την οποία θα παράξουμε τον τελικό κώδικα.

- int stackStart=600;

Καθολική μεταβλητή που δείχνει τη θέση στη μνήμη της αρχής της στοίβας του προγράμματος.

- int firstjump=1;

Καθολική μεταβλητή που αν είναι 1 δείχνει ότι το πρώτο jump στην L0 δεν έχει γίνει, ενώ αν είναι 0 δείχνει ότι έχει γίνει.

- `int L0=0;`

Καθολική μεταβλητή που αν είναι 1 δείχνει ότι πρέπει να εκτυπωθεί η L0 ετικέτα στο αρχείο.

- `void create_final_code(quadlist *programlist);`

Υλοποιεί το κομμάτι της δημιουργίας του τελικού κώδικα και λειτουργεί ως ένα τεράστιο switch. Αρχικά εκτυπώνει την κατάλληλη ετικέτα και στη συνέχεια ελέγχει σε ποια εντολή ενδιάμεσου κώδικα αντιστοιχεί η τετράδα που εξετάζουμε και αναλόγως μπαίνει στην κατάλληλη if που θα εκτυπώσει στο αρχείο τον κατάλληλο κώδικα.

- `void loadvr(char *v, int r);`

Υλοποιεί τη συνάρτηση `loadvr()` που περιγράψαμε παραπάνω εκτυπώνοντας τον κατάλληλο κώδικα.

- `void storerv(int r, char *v);`

Υλοποιεί τη συνάρτηση `storerv()` που περιγράψαμε παραπάνω εκτυπώνοντας τον κατάλληλο κώδικα.

- `void gnlvcode(int step, int offset);`

Υλοποιεί τη συνάρτηση `gnlvcode()` που περιγράψαμε παραπάνω εκτυπώνοντας τον κατάλληλο κώδικα.

- `int find_current_nesting_level(scope *rootscope);`

Βρίσκει και επιστρέφει το επίπεδο φωλιάσματος στο οποίο βρίσκεται το πρόγραμμα.

- `int find_var_nesting_level(scope *rootscope, char *name);`

Βρίσκει και επιστρέφει το επίπεδο φωλιάσματος της μεταβλητής name. Έτσι σε συνδυασμό με το επίπεδο φωλιάσματος στο οποίο βρίσκεται το πρόγραμμα μπορεί να καταλάβει ο μεταφραστής αν η μεταβλητή που κάθε φορά εξετάζει έχει το ίδιο ή διαφορετικό επίπεδο φωλιάσματος με το τρέχον και να μπει στην κατάλληλη περίπτωση.

- `int find_offset(scope *rootscope, char *name);`

Βρίσκει και επιστρέφει μέσω του πίνακα συμβόλων το offset της μεταβλητής name, ενώ αν δεν βρεθεί επιστρέφει την τιμή 0 που δηλώνει ότι η μεταβλητή name δεν έχει δηλωθεί οπότε εκτυπώνεται και κατάλληλο μήνυμα σφάλματος.

- `int find_framelength(scope *rootscope);`

Βρίσκει και επιστρέφει μέσω του πίνακα συμβόλων το framelength της συνάρτησης στην οποία βρισκόμαστε.

- `void find_starting_quad(scope *rootscope, char *start_quad, char* call_name);`

Βρίσκει την ετικέτα της πρώτης εκτελέσιμης εντολής της συνάρτησης call\_name μέσω του πίνακα συμβόλων και την αποθηκεύει στην start\_quad.

- `int case_father_call(scope *rootscope, char *call_name, int nest_lev);`

Αναζητά αν η συνάρτηση call\_name ανήκει στην περίπτωση όπου καλείται συνάρτηση με μεγαλύτερο βάθος φωλιάσματος από την καλούσα (η καλούσα είναι ο πατέρας της κληθείσας).

- `int case_myself_call(scope *rootscope, char *call_name, int nest_lev);`

Αναζητά αν η συνάρτηση call\_name ανήκει στην περίπτωση όπου η καλούσα και η κληθείσα έχουν το ίδιο βάθος φωλιάσματος (καλούσα και κληθείσα έχουν τον ίδιο πατέρα) και η κληθείσα είναι η ίδια η καλούσα (ο εαυτός της).

- `int case_brother_call(scope *rootscope, char *call_name, int nest_lev);`

Αναζητά αν η συνάρτηση call\_name ανήκει στην περίπτωση όπου η καλούσα και η κληθείσα έχουν το ίδιο βάθος φωλιάσματος (καλούσα και κληθείσα έχουν τον ίδιο πατέρα) και η κληθείσα είναι αδερφός της καλούσας.

## 8. Χρήση Μεταφραστή

Για να μεταγλωττιστεί ο μεταφραστής χρειάζεται να γίνει `compile`, οπότε εκτελούμε:

```
gcc final_code_2012.c -o final_code_2012
```

Στη συνέχεια για να μεταφράσει ένα αρχείο σε γλώσσα του, πχ το `test.toy` εκτελούμε:

```
final_code_2012 test.toy
```

Αν έχει ολοκληρωθεί σωστά η μετάφραση και δεν υπήρχαν συντακτικά σφάλματα θα έχει δημιουργηθεί το αρχείο `test.out` με τον τελικό κώδικα.

Το αρχείο αυτό μπορεί να εκτελεστεί με την εντολή:

```
metasim test.out
```

με την προϋπόθεση ότι έχει εγκατασταθεί ο `metasim` στον υπολογιστή μας.

## ΠΑΡΑΡΤΗΜΑ: Ευρετήριο συναρτήσεων και μεταβλητών

Παρακάτω παρατίθεται το ευρετήριο όλων των μεταβλητών και συναρτήσεων που χρησιμοποιήθηκαν στον μεταφραστή:

*aquad.....	24	case_brother_call .....	50
*ascope .....	35	case_father_call .....	50
*finalcode .....	48	case_myself_call .....	50
*fp.....	16	check_reserved .....	17
*hlp_arg_list .....	37	checkbuflength .....	18
*hlp_par_list .....	36	condition .....	22
*hlp_var_list .....	36	create_final_code .....	49
*lastquad.....	24	create_transition_table .....	17
*tmpquad=NULL .....	48	declarations .....	21
actualparitem .....	22	delete_last_scope .....	39
actualparlist .....	22	elsepart .....	22
actualpars .....	22	emptylist .....	26
add_help_arg_list_to_scope	39	entoff .....	37
add_help_par_list_to_scope	39	expression .....	22
add_help_var_list_to_scope	38	factor .....	22
add_new_argument .....	37	find_current_nesting_level	49
add_new_entity .....	37	find_framelength .....	50
add_new_function .....	38	find_offset .....	50
add_new_parameter .....	38	find_starting_quad .....	50
add_new_scope .....	37	find_var_nesting_level .....	49
add_new_tempvar .....	38	firstjump .....	48
add_new_variable .....	38	for_stat .....	22
add_oper .....	22	formalparitem .....	22
add_par_to_list .....	38	formalparlist .....	22
add_var_to_list .....	38	formalpars .....	22
argoff .....	37	funcbody .....	21
assign_framelength .....	39	genquad .....	26
assign_starting_quad .....	39	gnlvcode .....	49
assignment_stat .....	22	if_stat .....	22
backpatch .....	26	input_stat .....	22
block .....	21	L0 .....	49
boolfactor .....	22	Level .....	37
boolterm .....	22	lex .....	17
brack_or_stat .....	22	line .....	17
brackets_seq .....	22	loadvr .....	49
call_stat .....	22	makelist .....	26

merge.....	26	searchentity.....	39
mul_oper.....	22	sequence.....	22
newTemp .....	26	stackStart.....	48
nextquad.....	25	statement.....	22
NextQuad[7] .....	25	storerv.....	49
optional_sign.....	22	subprograms .....	21
printEnums.....	18	syntax.....	21
print_quadlist.....	26	T[11][20].....	16
print_stat.....	22	temp_cnt.....	25
print_symbol_table.....	40	temp_var[5] .....	25
procOrfunc.....	21	term.....	22
program.....	21	token.....	16
relational_oper.....	22	varlist.....	21
return_stat.....	22	while_stat.....	22