ΜΕΤΑΦΡΑΣΤΕΣ

Προγραμματιστική άσκηση, εαρινό εξάμηνο 2019

Ιωάννης Γεωρβασίλης, ΑΜ: 2656

Ανδρέας Σιδεράς, ΑΜ: 2813

## ΕΙΣΑΓΩΓΗ

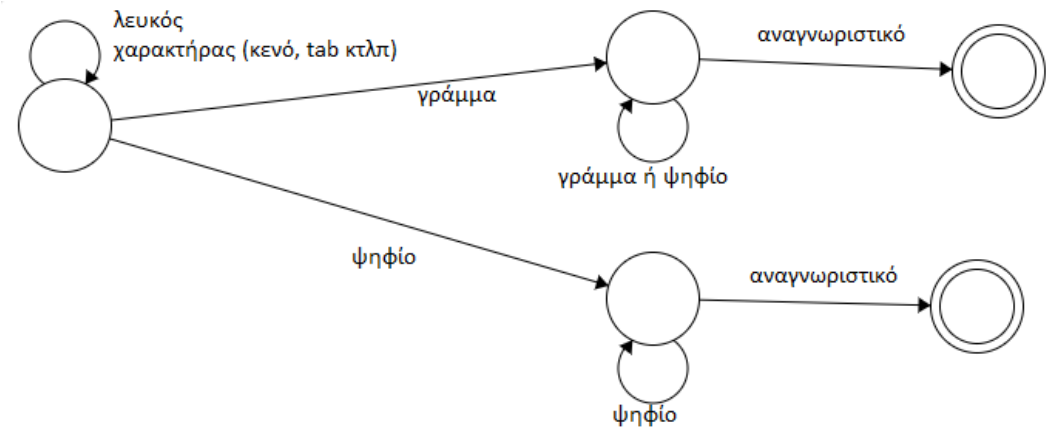
Στην παρούσα προγραμματιστική άσκηση καλούμαστε να υλοποιήσουμε πλήρως έναν compiler για μια προγραμματιστική γλώσσα που μας δόθηκε, τη starlet. Ως compiler ορίζεται ένα πρόγραμμα υπολογιστή που διαβάζει κώδικα γραμμένο σε μια γλώσσα προγραμματισμού (πηγαία γλώσσα) και τον μεταφράζει σε ισοδύναμο κώδικα σε γλώσσα μηχανής (assembly). Η assembly που θα πράγει ο compiler μας τρέχει στην αρχιτεκτονική του MIPS. Ώστε να εκπληρωθούν οι εκπαιδευτικές απαιτήσεις του μαθήματος, αλλα και να διατηρηθεί η πολυπλοκότητα του κώδικά μας σε χαμηλό επίπεδο, η υλοποίηση του project διαιρέθηκε σε διάφορα αυτονομημένα αλλα όχι ανεξάρτητα μέρη. Το πρώτο μέρος αναφέρεται ως *λεκτική και σημασιολογική ανάλυση,* ακολουθούν η παραγωγή *ενδιάμεσου κώδικα, ο πίνακας συμβόλων* και καταλήγουμε στον στόχο μας, την παραγωγή *τελικού κώδικα.*

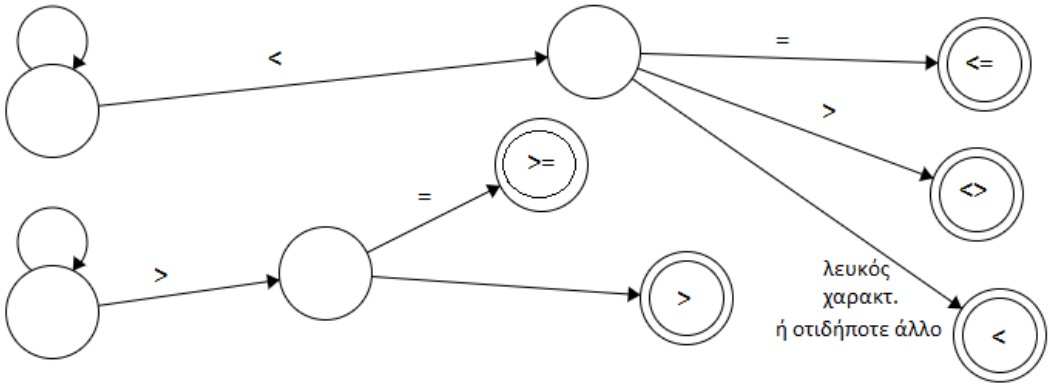
Κάθε γλώσσα προγραμματισμού διαθέτει το δικό της αλφάβητο και τη δική της γραμματική, σύμφωνα με τα οποία ολοκληρώσαμε τα παραπάνω στάδια. Η δομή δηλαδή της γλώσσας αποτελεί το θεμέλιο πάνω στο οποίο στήσαμε τον μεταγλωττιστή μας. Στη συνέχεια θα γίνει αναλυτική παρουσίαση των βημάτων που ακολουθήσαμε για κάθε ξεχωριστό μέρος του project.

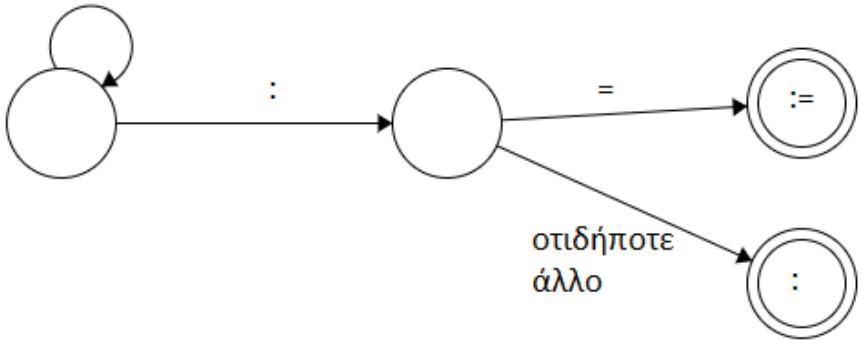
## ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Ο λεκτικός αναλυτής είναι η βάση του μεταγλωττιστή μας. Είναι μια συνάρτηση, που κάθε φορα που καλείται, διαβάζει το αρχείο .stl και επιστρέφει μια λεκτική μονάδα. Ως λεκτική μονάδα χαρακτηρίζεται κάθε χαρακτήρας ή συνδυασμός χαρακτήρων που είναι αποδεκτός απο τη γλώσσα προγραμματισμού μας. Για παράδειγμα μια λεκτική μονάδα είναι το "<>", αλλα όχι το "<!". Το τι αποτελεί και το τι δεν αποτελέι λεκτική μονάδα καθορίζεται απο την δομή και την γραμματική της γλώσσας προγραμματισμού μας. Ο λεκτικός αναλυτής διαβάζει γράμμα-γράμμα το πηγαίο πρόγραμμα και αναγνωρίζει ή οχι μια λεκτική μονάδα. Αν αναγνωρίσει μια λεκτική μονάδα, την επιστρέφει στη συνάρτηση του *συντακτικού αναλυτή* (το οποίο είναι το επόμενο επίπεδο της μεταγλώτισσης). Σε περίπτωση που βρεί κάποια

μη αποδεκτή λεκτική μονάδα (όπως δώσαμε ως παράδειγμα πιο πάνω), διακόπτει τη μεταγλώτιση και εμφανίζει κατάλληλο μύνημα λάθους. Ο λεκτικός αναλυτής εσωτερικά λειτουργεί σαν ένα αυτόματο καταστάσεωντο οποίο ξεκινά από μία αρχική κατάσταση, με την είσοδο κάθε χαρακτήρα αλλάζει κατάσταση έως ότου συναντήσει μία τελική κατάσταση. Το αυτόματο καταστάσεων αναγνωρίζει δεσμευμένες λέξεις (π.χ. if,while), σύμβολα της γλώσσας (π.χ. "+" , ";" ) , αναγνωριστικά και σταθερές (π.χ. counter, a12, 32768) και λάθη (π.χ. κλείσιμο σχολίων χωρίς να έχουν ανοίξει προηγουμένως). Βασικές καταστάσεις του αυτομάτου μας φαίνονται παρακάτω (αριστερότερη η αρχική κατάσταση state0):



Στο μεταγλωτιστή μας, η συνάρτηση που αφορά τον λεκτικό αναλυτή, ονομάζεται lex(). Η lex(), κάθε φορά που καλείται ενημερώνει τις global μεταβλητές token (η οποία παίρνει την τιμή της λεκτικής μονάδας που μόλις διαγιγνώσθηκε) και tokentk (η οποία είναι μια περιγραφή της λεκτικής μονάδας). Ως περιγραφή της λεκτικής μονάδας, ορίσαμε μερικές σφαιρικές μεταβλητές στον κώδικά μας με κατάληξη tk. Για παράδειγμα ορίσαμε την idtk, την οποία βάζουμε στην tokentk, κάθε φορά που διαβάζουμε όνομα συνάρτησης ή μεταβλητή (συνυπολογίζουμε και τις παραμέτρους ως μεταβλητές). Αυτό μας βοηθάει στο να ξέρουμε τι ακριβώς έχουμε διαβάσει, ανεξάρτητα απο το όνομά του (οι συναρτήσεις ενός προγράμματος έχουν διαφορετικά ονόματα, εμείς πρέπει να κρατάμε πως αυτό που διαβάσαμε ήταν όνομα συνάρτησης). Το αυτόματο του λεκτικού αναλυτή το υλοποιούμε ως έναν δισδιάστατο πίνακα καταστάσεων array, όπου κάθε στοιχείο είναι μια κατάσταση. Η αλλαγή μια κατάστασης του αυτομάτου Α σε Β, υποδηλώνει πως βρισκόμασταν στο array[A] και πήγαμε στο array[B]. Ο πίνακας αυτός είναι πίνακας ακεραίων. Κάθε κατάσταση (περιεχόμενα του πίνακα) state0, state1....state9 έχει αρχικοποιηθεί σε τιμές 0....9 αντίστοιχα. Επίσης έχουν ορισθεί οι τελικές καταστάσεις OK,OK2....OK5 με τιμές -2....-5 και η κατάσταση λάθους Error στην τιμή -1. Έτσι λοιπόν, η βασική λειτουργία της lex() καταλήγει να είναι σύγκριση ακεραίων. Χρειαζόμαστε επίσης μια μεταβλητή input, ώστε να κρατάμε τον επόμενο χαρακτήρα που διαβάστηκε. Η συνάρτηση newSymbol(), επιστρέφει έναν ακέραιο για κάθε γράμμα του αλφαβήτου της Starlet, ώστε να γίνουν οι απαραίτητες συγκρίσεις με όσα περιγράψαμε πιο πάνω.

Τη στιγμή που καλείται η lex() αρχικοποιούνται οι μεταβλητές token και tokentk στο κενό, ενω μια μεταβλητή state που είναι δείκτης στην κατάσταση που βρισκόμαστε, παίρνει την τιμή της αρχικής κατάστασης (state0). Στην συνέχεια εισερχόμαστε σε εναν βρόγχο while με συνθήκη εξόδου η state να βρεθεί σε κάποια τελική (OK) κατάσταση ή στην κατάσταση λάθους (Error). Με τις μεταβλητές μας αρχικοποιημένες, ήρθε η ωρα να διαβάσουμε έναν χαρακτήρα απο το πηγαίο πρόγραμμα το οποίο βάζουμε σε μια μεταβλητή input. Πλέον έχουμε την είσοδο, αρα είμαστε σε θέση να αλλάξουμε την κατάσταση του αυτομάτου μας. Αυτό γίνεται με την εντολή state=array[state][newSymbol(input)]. Σε πολλές περιπτώσεις, όπως όταν διαβάζουμε χαρακτήρες, θα πρέπει να <<κοιτάξουμε>> τι βρίσκεται αμέσως μετά τον χαρακτήρα που καταναλώσαμε. Για παράδειγμα, στην περίπτωση που διαβάζουμε αλφαριθμητικό (σύνολο χαρακτήρων) θα πρέπει να κρατάμε κάπου όσους χαρακτήρες έχουμε διαβάσει μέχρι να έρθει λευκός χαρακτήρας, όπου θα πάμε σε τελική (OK) κατάσταση. Το ίδιο συμβαίνει όταν έχω διαβάσει ένα "<" και δεν γνωρίζω εαν είναι κάποιο απο τα " <> , <= " κτλπ. Συνεπώς χρησιμοποιούμε μεταβλητή προσωρινής αποθήκευσης buffer. Αν όλα κυλήσουν ομαλά, μετά απο εναλλαγή καταστάσεων, θα βρεθούμε σε κατάσταση αποδοχής OK, θα εξέλθουμε απο τον βρόγχο και η συνάρτηση θα τελειώση την εκτέλεσή της έχοντας τις

σφαιρικές μεταβλητές token και tokentk ενημερωμένες. Σε διαφορετική περίπτωση θα τυπώσει μύνημα λάθους και την γραμμή στην οποία εμφανίστηκε. Συνεπώς εδω υπολογίζουμε και τις αλλαγές των γραμμών (μετράμε σε ποιά γραμμή βρισκόμαστε). Τέλος, να αναφέρουμε πως η lex() κάνει και διαχείριση των σχολίων του προγραμματιστή αγνωώντας οτιδήποτε υπάρχει ανάμεσα σε /* και */ (σε πολλαπλές γραμμές) ή μετά απο // (μόνο την τρέχουσα γραμμή), ενω επίσης διαγινώσθει eof (end of file) στην περίπτωση που δεν υπάρχει κάτι άλλο να διαβάσει.

## ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Η υλοποίηση του συντακτικού αναλυτή είναι βασισμένη στη "γραμματική" της γλώσσας Starlet. Οι κανόνες της γραμματικής αυτής (γραμματική χωρίς συμφραζόμενα) ουσιαστικά υποδεικνύουν τις συναρτήσεις που πρέπει να υλοποιηθούν στο κώδικα, έχοντας ώς απαραίτητη προϋπόθεση τις λεκτικές μονάδες που παράγει ο λεκτικός αναλυτής. Για να ολοκληρωθεί με επιτυχία η συντακτική ανάλυση, του προγράμματος που έφτιαξε ο χρήστης, θα πρέπει η σύνταξη του προγράμματος να ακολουθεί επακριβώς τους κανόνες της γραμματικής, ειδάλως μυνήματα λάθους θα διακόπτουν την ολοκλήρωση της συντακτικής ανάλυσης. Παρακάτω εξηγούνται πιο αναλυτικά οι συναρτήσεις που υλοποιούν τους κανόνες της γραμματικής.

Παρατηρήσεις:

- Οι λέξεις των κανόνων με κίτρινο χρώμα υποδεικνύουν μια λέξη, η οποία θα πρέπει να ταιριάζει με τη λεκτική μονάδα που παρήγαγε η lex(). Για συντομία και για την αποφυγή της επανάληψης της φρασης "παραγωγή επόμενης λεκτικής μονάδας απο την συνάρτηση lex()" θα χρησιμοποιήσουμε κατευθείαν τον όρο "κλήση της lex()".
- Αντίθετα οι λέξεις μεσα στα brackets δηλώνουν πώς πρέπει μια νέα συνάρτηση να κληθέι για να συνεχιστεί και τελικά να ολοκληρωθεί η συντακτική ανάλυση.
- Κάθε φορά που ελέγχουμε την ορθότητα της λεκτικής μονάδας (πχ. If (token == intk ) ) σε σχέση με τη γραμματική της γλώσσας, είναι ανάγκη να παράξουμε άμεσα την επόμενη λεκτική μονάδα, αφού θα τη χρειαστούν είτε οι επόμενες συναρτήσεις που θα κληθούν, είτε η συνάρτηση στην οποία θα επιστρέψει, είτε ο έλεγχος ορθότητας της επόμενης λεκτικής μονάδας.

❿ Κανόνας <program> : program id [block] endprogram

Αρχικά στη συνάρτηση αυτή καλούμε την lex() και ελέγχουμε έαν η λεκτική μονάδα που παράχθηκε είναι η λέξη "program". Εφόσον ο έλεγχος ήταν ορθός παράγουμε την επόμενη λεκτική μονάδα καλώντας την lex() η οποία αναμένουμε να είναι ίδια με το όνομα του προγράμματος .stl που μεταφράζεται. Έπειτα θα

καλέσουμε τη συνάρτηση block() αφού πρώτα κληθεί η lex() . Θα ελέγξουμε εάν αυτή είναι η λέξη “endprogram” και εφόσον είναι θα ολοκληρώσουμε με επιτυχία την λεκτική ανάλυση του προγράμματος. Έαν κάποια απ τις προϋποθέσεις που αναφέρθηκαν δεν ισχύει τότε διακόπτουμε την συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους καλώντας την συνάρτηση error_comp(), για την οποία θα μιλήσουμε στο τέλος.

❿ Κανόνας <block> : [declarations] [subprograms] [statements]

Το μόνο που χρειάζεται να κάνουμε στη φάση της συντακτικής ανάλυσης στη συνάρτηση block() ειναι να καλέσουμε με τη σειρά την declerations(), την subprograms(), και την statements()*. Κάθε φορά που επιστρέφει μια συνάρτηση εκεί που κλήθηκε έχουμε εξασφαλίσει οτι η επόμενη λεκτική μονάδα έχει παραχθεί.

* Η σειρά που καλούνται υποδεικνύει το πώς ένα block κώδικα πρέπει να είναι διατυπωμένο.

❿ Κανόνας <declarations> : ( declare [varlist] ;)*

Με το που ξεκινάει η εκτέλεση της declerations() η λεκτική μονάδα έχει ήδη παραχθέι απο τη τελευταία συνάρτηση που κάλεσε το λεκτικό αναλυτή (στη συγκεκριμένη περίπτωση είναι η program()). Με αυτό το τρόπο μπορούμε να ελένξουμε κατευθείαν την ορθότητα της λεκτικής μονάδας, δηλαδή εάν αυτή είναι η λέξη “declare”. Έαν και εφόσον υπάρχει καλούμε τη lex() και στη συνέχεια τη varlist(). Μετά το τέλος της εκτέλεσης της varlist() ελέγχουμε έαν υπάρχει το σύμβολο “;” ώστε να διασφαλίσουμε τη σωστή σύνταξη.

Η διαδικασία αυτή γίνεται επαναληπτικά έως ότου να μήν βρούμε “declare”.

❿ Κανόνας <varlist> : ε | id ( , id )*

Σκοπός της συνάρτησης varlist() είναι η αναγνώριση των id που δηλώνονται ώς μεταβλητές στο block της συνάρτησης-κυρίου προγράμματος.

-1η περίπτωση (ε) : Έαν δούμε κατευθείαν το σύμβολο “;” καλούμε την lex() και επιστρέφουμε άμεσα εκεί απο όπου κλήθηκε. Αυτό σημαίνει πως δεν υπάρχουν id προς δήλωση κάτι το οποίο είναι αποδεκτό, καθώς μας το επιτρέπει η γραμματική της γλώσσας.

-2η περίπτωση ( id (, id)* ) : Χρειάζεται μια επαναληπτική διαδικασία η οποία θα σταματάει επιτυχώς μόνο όταν δούμε το σύμβολο “;” και η προηγούμενη λ.μ. ήταν τύπου id. Εάν η προηγούμενη λ.μ. δεν ήταν τύπου id και η τρέχουσα λ.μ. είναι το “,” αυτό σημαίνει πως η σύνταξη δεν είναι ορθή και επιστρέφουμε μήνυμα

λάθους. Αυτό συμβαίνει και στη περίπτωση που η προηγούμενη λ.μ. ήταν τύπου id αλλά η τρέχουσα δεν ήταν το “,”. Αντιθέτως, καλείται η lex() κάθε φορά που α) η προηγούμενη ήταν τύπου id και η τρέχουσα είναι το “,” και β) η προηγούμενη λ.μ. δεν ηταν τύπου id και η τρέχουσα δεν είναι τα σύμβολα “,” και “;”.

- Κανόνας <subprograms> : ([subprogram])*

Στη συνάρτηση subprograms() το μόνο που χρειάζεται να κάνουμε είναι να καλούμε επαναληπτικά την συνάρτηση subprogram(). Είναι εξασφαλισμένο όπως έχουμε ξαναναφέρει, οτι κάθε φορά που επιστρέφει η subprogram() έχει παραχθεί η επόμενη λ.μ.

- Κανόνας <subprogram> : function id [funcbody] endfunction

Στη συνάρτηση subprogram() ελέγχουμε αρχικά την ορθότητα της λ.μ., δηλαδή εάν είναι η λέξη “function”. Εφόσον ισχύει καλείται η lex() και ελέγχουμε εάν η λ.μ. είναι τύπου id. Εάν και αυτό ισχύει τότε θα κληθεί η συνάρτηση funcbody(). Μόλις επιστρέψειέχοντας σίγουρα παραχθεί η επόμενη λ.μ. ελέγχουμε την ορθότητά της, εάν δηλαδή είναι η λέξη “endfunction”. Εάν ισχύει τότε καλείται η lex() και η subprogram() έχει πλέον ολοκληρώσει την δουλειά της.

- Κανόνας <funcbody> : [formalpars] [block]

Στην funcbody() το μόνο που χρειάζεται στη φάση της συντακτικής ανάλυσης είναι η σειριακή κλήση των συναρτήσεων formalpars() και block().

- Κανόνας <formalpars> : ( [formalparlist] )

Στην formalpars() χρειάζεται να ελέγξουμε την ορθότητά της λ.μ. έαν είναι δηλαδή το σύμβολο “(“. Εφόσον ισχύει καλείται η lex() και στη συνέχεια η formalparlist(), όπου με την επιστροφή της θα ελεγχθεί εάν η λ.μ. είναι το σύμβολο “)” και εφόσον ισχύει τότε καλείται η lex(). Η formalpars() σε αυτό το σημείο θα έχει ολοκληρώσει τη δουλειά της.

❿ Κανόνας <formalparlist> : [formalparitem] ( , [formalparitem] )* | ε

-1η περίπτωση (ε) : Στη συνάρτηση formalparlist() αρχικά ελέγχουμε εάν η λ.μ. είναι το σύμβολο “)”. Στη περίπτωση αυτη, είναι απαραίτητο να μήν παράξουμε την επόμενη λ.μ. γιατί θα επηρεάσουμε την λειτουργία της formalpars() (της συνάρτησης δηλαδή που κάλεσε την formalparlist()), αφού και αυτή χρειάζεται για να ολοκληρώσει την δουλειά της να δεί το σύμβολο “)”.

-2η περίπτωση ( [formalparitem] ( , [formalparitem] )*) : Είναι ανάγκη να καλείται επαναληπτικά η formalparitem() και αμέσως κατά την επιστροφή της να ελέγχεται εάν η επόμενη λ.μ. είναι το σύμβολο “,”. Όσο βρίσκουμε σύμβολο “,” η διαδικασία επαναλα- μβάνεται. Η διαδικασία τερματίζει όταν αντί του συμβόλου “,” συναντήσουμε το σύμβολο “)” πράγμα που σημαίνει ότι η formalparitem() διάβασε όσα id υπήρχαν και ολοκλήρωσε τη δουλειά της. Στο σημείο αυτό η συνάρτηση formalparlist() έχει ολοκληρώσει την δουλειά της. Τέλος, σε περιπτωση που δε συναντήσουμε κάποιο απ τα σύμβολα (“,” | “)”) σημαίνει πως η σύνταξη δεν είναι σωστή και τερματίζουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους καλώντας την error_comp().

❿ Κανόνας <formalparitem> : in id | inout id | inandout id

Στη συνάρτηση formalparitem() ελέγχουμε εάν η λ.μ. είναι είτε τύπου intk, είτε τύπου inouttk είτε τύπου inandouttk. Εάν ισχύει τότε καλείται η lex() και κατόπιν ελέγχεται εάν η νέα λ.μ. είναι τύπου id. Στη περίπτωση αυτή η formalparitem() έχει ολοκληρώσει τη δουλεία της, καλεί την lex() και επιστρέφει εκεί απ’ όπου κλήθηκε. Εάν κάποιος απ τους παραπάνω ελέγχους δεν ισχύει, τότε διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας τα κατάλληλα μηνύματα λάθους.

❿ Κανόνας <statements> : [statement] ( ; [statement] )*

Στη προκειμένη συνάρτηση θα καλέσουμε κατευθείαν την statement(). Όταν επιστρέψει θα γίνει επαναληπτικά έλεγχος για το εάν η επόμενη λ.μ. είναι το σύμβολο “;”. Εάν η συνθήκη ικανοποιείται τότε ξανακαλούμε τη statement(). Όταν η συνθήκη δεν ικανοποιηθεί, τότε η συνάρτηση statements() έχει τελειώσει τη δουλειά της.

Να τονίσουμε πώς δεν καλείται η lex() πρίν την επιστροφή της συνάρτησης γιατί αυτή τη δουλειά θα την έχει κάνει η statement().

❿ Κανόνας <statement> :

Στην συνάρτηση statement() ελέγχουμε τον τύπο της λ.μ., καλούμε τη lex() και κατόπιν καλούμε την αντίστοιχη συνάρτηση. Να σημειωθεί πως ο τύπος της λ.μ. που ελέγχεται είναι η πρώτη λέξη του κανόνα που θα κληθεί. Αν η λ.μ. δεν είναι γνωστού τύπου*, σημαίνει πως είναι τύπου id, οπότε θα κληθεί η assignment_stat().

(πχ: if(token==looptk) {
lex()
loop_stat()
} )

Υπάρχει μια εξαίρεση στον έλεγχο για την λ.μ. “else” η οποία δε καλεί καμία συνάρτηση αλλά ούτε και παράγει κάποια λ.μ. καλώντας τη lex(). Ο λόγος, εξηγείται αναλυτικά στο κανόνα <if-stat>.

* iftk, dowhiletk, whiletk, looptk, exittk, forcasetk, incasetk, returntk, inputtk, printtk. Σημαντικό είναι να αναγνωρίσουμε πως αυτοί οι τύποι λ.μ. αναφέρονται στην πρώτη λέξη της σύνταξης της εντολής. Οι τελευταίες λέξεις (πχ. Endiftk, endwhiletk, κλπ) δεν μπορούν να ελεγχθούν εδώ παραμόνο μέσα στη συνάρτησή τους.

❿ Κανόνας <assignment-stat> : id := [expression]

Στην συνάρτηση assignment_stat(), αρχικά ελέγχεται αν η λ.μ. είναι τύπου id. Εφόσον ισχύει καλείται η lex() και ελέγχεται εάν η νέα λ.μ. είναι η συμβολοσειρά “:=”. Αν και αυτό ισχύει καλούμε τη lex() και στη συνέχεια τη συνάρτηση expression().

Είναι σημαντικό να αποφύγουμε το μήνυμα λάθους σε περίπτωση που η λ.μ. δεν είναι τύπου id αλλά είναι κάποια απ τις δεσμευμένες λέξεις που αναφέρονται στο κλείσιμο ενός statement. Αυτός ο έλεγχος είναι απαραίτητο να γίνει πρώτος απ όλους, στην αρχή της συνάρτησης assignment_stat().

Για παράδειγμα:

```
while (x<>0)
        x:=x-1;
        print x;
endwhile;
```

Στο συγκεκριμένο παράδειγμα, μετά την ολοκλήρωση της συντακτικής ανάλυσης του statement “print x;”, η συνάρτηση statements() θα καλούσε ξανά την statement(), η statement() επειδή το “endwhile” δεν είναι μια απ τις λ.μ. γνωστού τύπου ανοίγματος statement θα καλούσε την assignment_stat() και αυτή με τη σειρά της ελέγχωντας εάν η λ.μ. είναι τύπου id, θα έβγαζε μήνυμα λάθους καθώς θα αναγνώριζε πως είναι δεσμευμένη λέξη.

❿ Κανόνας <if-stat> : if ( [condition] ) then [statements] [elsepart] endif

Στη συνάρτηση if_stat() ελέγχουμε εάν η λ.μ. είναι το σύμβολο "(". Κατόπιν καλείται η lex() και η condition(). Με την επιστροφή της condition() ελέγχουμε εάν η λ.μ. είναι το σύμβολο ")", καλείται εκ νέου η lex() ώστε να γίνει έλεγχος της λέξης "then" και εάν όλα ώς εδώ δουλεύουν σωστά, καλούμε τη lex() και αμέσως μετά τη statements().

Σε αυτό το σημείο θα εξηγήσουμε γιατι στην συνάρτηση statement() υπάρχει ο έλεγχος για τη λέξη "else" αλλά δεν καλούμε ούτε την lex() ούτε την elsepart() όπως θα φαινόταν λογικό. Αυτο συμβαίνει γιατί θέλουμε να καλέσουμε την elsepart() μέσα στην συνάρτηση του κανόνα που ανήκει, δηλαδή την if_stat(). Η statements() θα έχει τερματίσει, είτε βλέποντας "else" είτε έχοντας ελέγξει όλα τα statements που υπάρχουν μέσα στο κανόνα της if_stat().

Αφού λοιπόν τερματίσει η statements(), καλούμε την else_part() και στη συνέχεια ελέγχουμε εάν η λ.μ. είναι η λέξη "endif".

❿ Κανόνας <elsepart> : ε | else [statements]

Στη συνάρτηση eslepart(), ελέγχουμε αν η λ.μ. είναι η λέξη "else" και στη συνέχεια καλούμε την lex() και την statements(). Σε περίπτωση που δεν δούμε τη λέξη "else" τότε η συνάρτηση δεν καλεί την lex() και επιστρέφει εκεί απο όπου κλήθηκε.

❿ Κανόνας <while-stat> : while ( [condition] ) [statements] endwhile

Στη συνάρτηση while_stat() ελέγχουμε αρχικά αν η λ.μ. είναι η λέξη "while", καλούμε την lex() και ελέγχουμε αν η νέα λ.μ. είναι το σύμβολο "(". Εάν ο έλεγχος είναι επιτυχής καλούμε την condition() και με την επιστροφή της ελέγχουμε άν η λ.μ. είναι το σύμβολο ")". Δεδομένου ότι ο έλεγχος ήταν και αυτός επιτυχής καλείται η statements() και αμέσως μετά την επιστροφή της ελέγχουμε αν η λ.μ. είναι η τελευταία απαραίτητη λέξη του κανόνα, δηλαδή η λέξη "endwhile". Καλούμε τη lex() αν ισχύει ο προηγούμενος έλεγχος και η συνάρτησή while_stat() επιστρέφει εκεί απ όπου κλήθηκε. Τέλος, αν κάποιος απ τους ελέγχους δεν ήταν επιτυχής διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <do-while-stat> : dowhile [statements] enddowhile ( [condition] )

Στη συνάρτηση do_while_stat() καλούμε απευθείας την statements() και αμέσως μετά την επιστροφή της ελέγχουμε άν η λ.μ. είναι η λέξη "enddowhile". Καλούμε την lex() και ο έλεγχος αυτή τη φορά γίνεται για το σύμβολο "(". Αν ισχύει,

καλούμε την condition(), αμέσως μετά ελέγχουμε αν η λ.μ. είναι το σύμβολο “)”. Η συνάρτησή επιστρέφει επιτυχώς εκεί απ όπου κλήθηκε αφού πρώτα καλέσουμε την lex(). Τέλος, αν κάποιος απ τους ελέγχους δεν ήταν επιτυχής διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <loop-stat> : loop [statements] endloop

Στη συνάρτηση loop_stat() καλούμε απευθείας την statements(). Κατά την επιστροφή της ελέγχουμε άν η λ.μ. είναι η λέξη “endloop” και εφόσον ισχύει η συνάρτηση loop_stat() επιστρέφει επιτυχώς εκεί απ όπου κλήθηκε αφού πρώτα καλέσουμε την lex().

Αν δεν ισχύει ο έλεγχος για την λέξη “endloop” διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <exit-stat> : exit

Στη συνάρτησή exit_stat() δεν χρειάζεται κάποιος επιπλέον έλεγχος, στη φάση της συντακτικής ανάλυσης. Απλώς επιστρέφει εκεί απ όπου κλήθηκε. Είναι ανάγκη να δημιουργηθεί μια συνάρτηση exit_stat() παρόλα αυτά, γιατί θα φανεί πολύ χρήσιμο στην φάση παραγωγής του ενδιάμεσου κώδικα.

❿ Κανόνας <forcase-stat> : forcase ( when ( [condition] ) : [statements] )*
default : [statements] enddefault endforcase

Στη συνάρτησή forcase_stat() πρώτα πρώτα ελέγχουμε επαναληπτικά την ύπαρξη της λέξης “when”. (*) Κάθε φορά που συναντάμε τη λέξη “when” καλούμε τη lex() και ελέγχουμε εάν η νέα λ.μ. είναι το σύμβολο “(“. Και πάλι καλείται η lex(), στη συνέχεια η condition() και με την επιστροφή της ελέγχουμε αν η λ.μ. είναι το σύμβολο “)”. Εφόσο και αυτο ισχύει καλείται εκ νέου η lex() και ο έλεγχος αυτή τη φορά, αφορά το σύμβολο “:”. Αν ο έλεγχος ισχύει καλούμε την statements(). Αυτή η διαδικασία όπως είπαμε γίνεται επαναληπτικά μέχρι να μήν συναντήσουμε την λέξη “when”. Τα μηνύματα λάθους αφορούν τους ελέγχους που γίνονται αφού συναντήσουμε “when”. (*) Εάν δεν υπάρχει, ελέγχουμε την ύπαρξη της λέξης “default” καθώς είναι απαραίτητη λ.μ. μέσα στο κανόνα της forcase. Εφόσον δούμε τη λέξη “default” καλείται η lex() και ελέγχεται εάν η νέα λ.μ. είναι το σύμβολο “:”. Καλείται η statements() και μετά την επιστροφή της ελέγχουμε εάν η επόμενη λ.μ. είναι η λέξη “enddefault”. Εάν ισχύει και αυτό μας απομένει μόνο να ελέγξουμε την λέξη “endforcase”. Η συνάρτησή εδώ καλεί την lex() και επιστρέφει εκεί απ όπου

κλήθηκε. Σε οποιαδήποτε διαφορετική περίπτωση διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

(*): η ίδια ακριβώς διαδικασία πραγματοποιείται και στην συνάρτηση incase_stat().

❿ Κανόνας <incase-stat> : incase ( when ( [condition] ) : [statements] )* endincase

Στη συνάρτησή incase_stat() αρχικά δουλεύουμε με τον ίδιο επαναληπτικό τρόπο όπως και στη συνάρτηση forcase_stat() (*). Εκτελούμε την ίδια ακριβώς διαδικασία με την εξαίρεση του ότι όταν βγούμε απ τον επαναληπτικό έλεγχο ύπαρξης λέξης “when” ελέγχουμε κατευθείαν εάν η λ.μ. είναι η λέξη “endincase”. Εφόσον ισχύει, καλούμε την lex() και επιστρέφει η συνάρτηση εκεί απ όπου κλήθηκε. Τέλος, επιστρέφουμε μήνυμα λάθους μόνο στη περίπτωση που δέν δούμε τη λέξη “endincase”.

❿ Κανόνας <return-stat> : return [expression]

Στη συνάρτησή return_stat() καλούμε αμέσως την expression(), αφού η λέξη “return” έχει διαβαστεί απ τη συνάρτησή statement().

❿ Κανόνας <print-stat> : print [expression]

Στη συνάρτησή print_stat() καλούμε αμέσως την expression(), αφού η λέξη “print” έχει διαβαστεί απ τη συνάρτησή statement().

❿ Κανόνας <input-stat> : input id

Στη συνάρτησή input_stat() ελέγχουμε αν η λ.μ. είναι τύπου id. Εφόσον ισχύει καλούμε τη lex() και επιστρέφει εκεί απ όπου κλήθηκε. Αν δεν είναι τύπου id, διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <actualpars> : ( [actualparlist] )

Στη συνάρτησή actualpars() ελέγχουμε αν η λ.μ. είναι το σύμβολο “(“. Εφόσον ισχύει καλείται lex() και στη συνέχεια η actualparlist(). Κατά την επιστροφή

της ελέγχουμε εάν η λ.μ. είναι το σύμβολο “)”. Εφόσον και αυτο ισχύει καλείται η lex() και και επιστρέφει εκεί απ όπου κλήθηκε. Τέλος, αν κάποιος απ τους ελέγχους δεν ήταν επιτυχής διακόπτουμε τη συντακτική ανάλυση εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <actualparlist> : [actualparitem] ( , [actualparitem] )* | ε

Στη συνάρτησή actualparlist() καλούμε απευθείας την actualparitem(). Μετά την επιστροφή της ελέγχουμε επαναληπτικά αν η λ.μ. της είναι το σύμβολο “,”. Όσο βρίσκουμε σύμβολο “,” η acualparitem() καλείται και αυτή. Πρίν απο τη κλήση της όμως ελέγχουμε άν η λ.μ. είναι το σύμβολο “)” ώστε να εμφανίσουμε μήνυμα λάθους. Μετά το τέλος της επαναληπτικής διαδικασίας ελέγχουμε αν η λ.μ. είναι μια απ τις λεξεις “in”,”inout”, ”inandout”. Αυτός ο έλεγχος γίνεται για να πιάσουμε τυχών λάθος πέρασμα παραμέτρων. Εξηγείται αναλυτικά στο κανόνα της actualparitem.

❿ Κανόνας <actualparitem> : in [expression] | inout id | inandout id

Στη συνάρτησή actualparitem() ελέγχουμε τις εξής περιπτώσεις:

-1η περίπτωση: Άν η λ.μ. είναι μια απ τις λέξεις: “inout”,”inandout” τότε καλείται η lex() και ελέγχεται εάν η νέα λ.μ. είναι τύπου id. Αν ισχύει καλείται και πάλι η lex() και η συνάρτησή επιστρέφει εκεί απ όπου κλήθηκε.

-2η περίπτωση: Άν η λ.μ. είναι η λέξη “in”, τότε καλέιται η lex() και στη συνέχεια η expression().

-3η περίπτωση: Μόνο άν η λ.μ. δεν είναι το σύμβολο “)” εμφανίζουμε μήνυμα λάθους. Αν είναι πάει να πεί πώς δεν υπήρχαν ορίσματα κατά τη κλήση της συνάρτησης.

Αφού η actualparitem αναγνωρίζει μόνο τις παραπάνω περιπτώσεις ειναι ανάγκη να εμφανίσουμε μήνυμα λάθους αν ανάμεσα απ τις παραμέτρους δεν υπάρχει το σύμβολο “,”. Για το λόγο αυτό ο έλεγχος γίνεται στην συνάρτησή actualaparlist().

❿ Κανόνας <condition> : [boolterm] ( or [boolterm] )*

Στη συνάρτησή condition() καλούμε απευθείας την boolterm(). Κατά την επιστροφή της ελέγχουμε επαναληπτικά άν η λ.μ. είναι η λέξη “or” και καλούμε τη lex() και την boolterm() εκ νέου. Η επαναληπτική διαδικασία σταματάει όταν δεν βρούμε λέξη “or”.

❿ Κανόνας <boolterm> : [boolfactor] ( and [boolfactor] )*

Στη συνάρτησή boolterm() καλούμε απευθείας την boolfactor(). Κατά την επιστροφή της ελέγχουμε επαναληπτικά άν η λ.μ. είναι η λέξη “and” και καλούμε τη lex() και την boolfactor() εκ νέου. Η επαναληπτική διαδικασία σταματάει όταν δεν βρούμε λέξη “and”.

❿ Κανόνας <boolfactor> : not [ [condition] ] | [ [condition] ] |
[expression] [relational-oper] [expression]

Στη συνάρτησή boolfactor διακρίνουμε τις εξής περιπτώσεις:

-1η περίπτωση ( not [ condtion() ] ): Στη περίπτωση αυτή ελέγχουμε αν η λ.μ. είναι η λέξη “not”. Αν ισχύει καλείται η lex() και ελέγχουμε αν η νέα λ.μ. είναι το σύμβολο “[“. Εφόσον και αυτό ισχύει καλούμε την lex() και την condtion() και αμέσως μετά ελέγχουμε αν η λ.μ. είναι το σύμβολο “]” Αν ισχύει τότε καλείται η lex() και η συνάρτηση επιστρέφει εκεί που κλήθηκε.

-2η περίπτωση ( [ condition() ] ): Κάνουμε την ίδια ακριβώς διαδικασία με τη περίπτωση 1, μόνο που δέν ελέγχουμε αν υπάρχει η λέξη “not”. Στις περιπτώσεις 1,2 εμφανίζομε μηνύματα λάθους και διακόπτουμε τη συντακτική ανάλυση άν κάποιος απ τους ελέγχους δεν ισχύει.

-3η περίπτωση: Στη περίπτωση αυτή απλώς καλούμε σειριακά τις συναρτήσεις expression(), relational_oper() και πάλι την expression().

❿ Κανόνας <expression> : [optional-sign] [term] ( [add-oper] [term] )*

Στη συνάρτησή expression() καλούμε απευθείας την optional_sign() και στη συνέχεια την term(). Καλούμε επαναληπτικά την add_oper() η οποία όσο βρίσκει ένα απ τα σύμβολα “+”,”-”, μας επιτρέπει να καλέσουμε την term(). Όταν ή add_oper() δεν βρεί ένα απ τα σύμβολα που προαναφέραμε τότε σταματάει και η επαναληπτική κλήση της term().

Επίσης μετά τον έλεγχο της add_oper() ελέγχουμε αν το επόμενο σύμβολο είναι πάλι ένα απ τα παραπάνω σύμβολα ώστε να εμφανίσουμε μήνυμα λάθους καθώς η γραμματική δεν το επιτρέπει.

❿ Κανόνας <term> : [factor] ( [mul-oper] [factor] )*

Στη συνάρτηση term() καλόυμε απευθείας την factor(). Καλούμε επαναληπτικά την mul_oper() η οποία όσο βρίσκει ένα απ τα σύμβολα “*”,”/”, μας επιτρέπει να καλέσουμε την factor(). Όταν ή mul_oper() δεν βρεί ένα απ τα σύμβολα

που προαναφέραμε τότε σταματάει και η επαναληπτική κλήση της factor(). Επίσης μετά τον έλεγχο της mul_oper() ελέγχουμε αν το επόμενο σύμβολο είναι πάλι ένα απ τα παραπάνω σύμβολα ώστε να εμφανίσουμε μήνυμα λάθους καθώς η γραμματική δεν το επιτρέπει.

❿ Κανόνας <factor> : constant | ( [expression] ) | id [idtail]

Στη συνάρτηση factor διακρίνουμε τις 3 παρακάτω περιπτώσεις:

- 1η περίπτωση (constant): Αν η λ.μ. είναι σταθερά (αριθμός) τότε καλούμε τη lex() και επιστρέφει η factor() εκεί απ όπου κλήθηκε.
- 2η περίπτωση ((expression())): Αν η λ.μ. είναι το σύμβολο "(" τότε καλείται η lex() και στη συνέχεια η expression(). Κατά την επιστροφή της άν η λ.μ. είναι το σύμβολο ")" τότε καλείται εκ νέου η lex() και επιστρέφει εκεί απ όπου κλήθηκε.
- 3η περίπτωση (id id_tail()): Αν η λ.μ. είναι τύπου id τότε καλείται η lex() και στη συνέχεια η id_tail().

Αν δεν ισχύει καμία απ αυτές τις περιπτώσεις τότε διακόπτουμε τη συντακτική ανάλυση και εμφανίζομε μήνυμα λάθους.

❿ Κανόνας <idtail> : ε | [actualpars]

Στη συνάρτηση idtail() ελέγχουμε αρχικά αν η λ.μ. είναι το σύμβολο "(". Αυτό ελέγχεται εδώ και όχι στη συνάρτηση factor() (θα μπορούσε να ελέγχεται και εκεί). Αν ισχύει τότε καλούμε την actualpars().

❿ Κανόνας <relational-oper> : = | <= | >= | > | < | <>

Στη συνάρτηση relational_oper() καλούμε την lex() αφού πρώτα ελέγξουμε αν η λ.μ. είναι κάποια απ τα σύμβολα "=","<=",">=",">","<","<>" και η συνάρτηση επιστρέφει εκεί που κλήθηκε. Αν δεν ισχύει τότε διακόπτουμε τη συντακτική ανάλυσης εμφανίζοντας μήνυμα λάθους.

❿ Κανόνας <add-oper> : + | -

Στη συνάρτηση add_oper() ελέγχουμε αν η λ.μ. είναι κάποια απ τα σύμβολα "+","-" επιστρέφοντας το αν ισχύει στην συνάρτηση που μας κάλεσε.

❿ Κανόνας <mul-oper> : * | /

Στη συνάρτηση mul_oper() ελέγχουμε αν η λ.μ. είναι κάποια απ τα σύμβολα “*”,”/” επιστρέφοντας το αν ισχύει στην συνάρτηση που μας κάλεσε.

❿ Κανόνας <optional-sign> : ε | [add-oper]

Στη συνάρτηση optional_sign() ελέγχουμε αν η λ.μ. είναι κάποια απ τα σύμβολα “+”,”-”. Αν ισχύει καλούε τη lex() και επιστρέφει εκεί απ όπου κλήθηκε.

## ΒΟΗΘΗΤΙΚΕΣ ΣΥΝΑΡΤΗΣΕΙΣ

❿ check_end_statement(): Ελέγχουμε εάν η λ.μ. είναι κάποια απ τις τελικές λέξεις των κανώνων της γραμματικής.

❿ trash(): Ελέγχουμε αν υπάρχουν λ.μ. έξω απ το κορμό του κυρίως προγράμματος. Αν ισχύει εμφανίζομε μήνυμα λάθους.

❿ errorc_comp(): Ανάλογα το λάθος στον έλεγχο των παραπάνω κανόνων της γραμματικής
εμφανίζομε και το ανάλογο μήνυμα λάθους. Η error_comp() λαμβάνει τις ετικέτες ώς ορίσματα απ τη συνάρτηση του κανόνα που βρέθηκε το λάθος και τυπώνει το ανάλογο μήνυμα, διακόποντας ύστερα τη συντακτική ανάλυση.

# ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ

Η παραγωγή ενδιάμεσου κώδικα είναι ουσιαστικά η ενδιάμεση μετάφραση του κώδικα που έγραψε ο χρήστης στη γλώσσα starlet πριν τη τελική μετάφραση σε γλώσσα μηχανής. Απο κάθε μια εντολή ξεχωριστά της γλώσσας starlet παράγονται αντίστοιχα, μία ή περισσότερες τετράδες. Κάθε τετράδα έχει μια ετικέτα (label), ένα τελεστή, και τρία τελούμενα.

(πχ: 1: :=, x, _, e , όπου “1:” το label της τετράδας (είναι μοναδικό για κάθε τετράδα), το “:=” αφορά το τελεστή, και τα επόμενα 3 πεδία είναι τα τελούμενα.)

Δημιουργήσαμε μια κλάση quad η οποία ώς πεδία της έχει τα παραπάνω χαρακτηριστικά με τις αντίστοιχες μεθόδους setters και getters. Επίσης, κάθε τετράδα-quad που δημιουργείται αποθηκεύεται στην λίστα των quads που έχουμε αρχικοποιήσει.

Διακρίνουμε τις εξής κατηγορίες τετράδων:

(1) Τετράδες με τελεστή αριθμητικών πράξεων:

Τετράδες οι οποίες έχουν ώς τελεστή ένα απ τα σύμβολα “+”,”-”,”*”,”/”, αναπαριστούν μια αριθμιτική πράξη μεταξύ των τελούμενων 1 και 2, με το αποτέλεσμα της πράξεις αυτής να αποθηκεύεται στο τελευταίο τελούμενο.

πχ: 10: +,a,3,b. Η τιμή της μεταβλητής a προστίθεται με την τιμή 3 και το αποτέλεσμα της πράξης αυτής αποθηκεύεται στη μεταβλητή b. (b:=a+3). “10” είναι η ετικέτα της τετράδας όπως αυτή προέκυψε κατά τη διαδικασία παραγωγής του ενδιάμεσου κώδικα.

(2) Τετράδες με τελεστή εκχώρησης:

Τετράδες οι οποίες έχουν ώς τελεστή το σύμβολο “:=”, αναπαριστούν μια εκχώρηση της τιμής του πρώτου τελούμενου στο τελευταίο τελούμενο.

πχ: 20: :=,x,_,e. Η τιμή της μεταβλητής x αποθηκεύεται στη μεταβλητή e.

(3) Τετράδες με τελεστή άλματος άνευ προϋποθέσεων:

Τετράδες οι οποίες έχουν ώς τελεστή τη λέξη “jump”, αναπαριστούν μια αναπήδηση στην ετικέτα που περιέχεται στο τελευταίο τελούμενο.

πχ: 30: jump,_,_,80. Άλμα απ την τρέχουσα ετικέτα 30 στην ετικέτα 80.

(4) Τετράδες με τελεστή συνθήκης και άλματος:

Τετράδες οι οποίες έχουν ώς τελεστή ένα απ τα σύμβολα “<”,”>”,”>=”,”<=”,”=”,”<>”, αναπαριστούν μια αναπήδηση στην ετικέτα που περιέχεται στο τελευταίο τελούμενο, εφόσον ισχύει η συνθήκη σύγκρισης μεταξύ του 1ου και 2ου τελούμενου.

πχ: 40: <=,x,y,90. Άλμα απ την τρέχουσα ετικέτα στην ετικέτα 90, άν και μόνο άν το x είναι μικρότερο ή ίσο του y.

(5) Τετράδες αρχής και τέλος ενότητας:

Τετράδες οι οποίες περιέχουν τον τελεστή begin_block δηλώνουν την αρχή ενός νέου μπλόκ κώδικα (κυρίως προγράμματος, συνάρτησης) ενώ αυτές που περιέχουν τον τελεστή end_block δηλώνουν το τέλος ενός μπλόκ κώδικα.

Τετράδες οι οποίες περιέχουν τον τελεστή halt δηλώνουν τερματισμό του προγράμματος.

(6) Τετράδες που αφορούν τη κλήση συνάρτης και πέρασμα παραμέτρων

Τετράδες οι οποίες περιέχουν τον τελεστή “par” δηλώνουν παράμετρο κατά τη κλήση μιας συνάρτησης. Άν το 2ο τελούμενο είναι η λέξη “cv” σημαίνει πως η μεταβλητή στο 1ο τελούμενο περνάει ώς παράμετρος στη συνάρτηση με τιμή. Αν είναι η λέξη “ref” σημαίνει πως η παράμετρος περνάει με αναφορά. Αν είναι η λέξη “cp” σημαίνει πως η παράμετρος περνάει με αντιγραφή. Αν είναι η λέξη “RET” σημάινει πως η μεταβλητή στο 1ο τελούμενο είναι το αποτέλεσμα της συνάρτησης που θα κληθεί.

πχ: 60: par,x,cv,_. Η μεταβλητή x θα περάσει ώς παράμετρος στη συνάρτηση με τιμή.

πχ: 61: par,T-1,RET,_. Η μεταβλητή T_1 θα περιέχει το αποτέλεσμα της συνάρτησης που θα κληθεί.

Τετράδες οι οποίες περιέχουν στον τελεστή “call” αναπαριστούν την κλήση της συνάρτησης

της οποίας το όνομα βρίσκεται στο 1ο τελούμενο.

πχ: 62: call,P1,_,_. Κλήση της συνάρτησης P1.

Τετράδες οι οποίες περιέχουν στο τελεστή τη λέξη “retv” αναπαριστούν τη τιμή που επιστρέφει μια συνάρτηση (return e;). Η τιμή αυτή βρίσκεται στη μεταβλητή που περιέχει το 1ο τελούμενο.

πχ: 63: retv,e,_,_. Η τιμή της μεταβλητής e είναι το αποτέλεσμα που θα επιστρέψει η συνάρτηση.

## (7) Τετράδες που αφορούν τις εντολές εισόδου εξόδου

Τετράδες οι οποίες περιέχουν στον τελεστή τη λέξη “inp” αναπαριστούν την εκχώρηση τιμής απ το πληκτρολόγιο στη μεταβλητή που περιέχεται στο τελευταίο τελούμενο. Η τιμή που έδωσε ο χρήστης απ το πληκτρολόγιο αποθηκεύεται στην μεταβλητή e.

Πχ: 70: inp,_,_,e.

Τετράδες οι οποίες περιέχουν στον τελεστή τη λέξη “out” αναπαριστούν το τύπωμα της τιμής της μεταβλητής που περιέχεται στο τελευταίο τελούμενο, στην οθόνη του χρήστη. Η τιμή της μεταβλητής e τυπώνεται στην οθόνη.

πχ: 71: out,_,_,e.

Είναι σημαντικό να αναφέρουμε πως η παραγωγή των τετράδων γίνεται κατά τη συντακτική ανάλυση. Αυτό επιτυγχάνεται με την προσθήκη γραμμών κώδικα στις ήδη υπάρχουσες συναρτήσεις που δημιουργήσαμε για τη κατασκευή του συντακτικού αναλυτή. Για το λόγο αυτό ήταν απαραίτητη η δημιουργία κάποιων επιπλέον συναρτήσεων οι οποίες εξηγούνται παρακάτω.

- nextquad() : Επιστρέφει την ετικέτα της επόμενης τετράδας που επρόκειτο να παραχθεί.

- genquad() : Λαμβάνει ώς ορίσματα τα χαρακτηριστικά της τετράδας (τελεστή και τελούμενα), και παράγει ένα αντικειμένο τύπου qaud προσθέτοντας στη λίστα των τετράδων.

- newtemp() : Δημιουργεί και επιστρέφει μια προσωρινή μεταβλητή. Κάθε φορά που καλείται η newtemp() η επόμενη προσωρινή μεταβλητή είναι διαφορετική απο τη προηγούμενη που παράχθηκε. (πχ: T_1, T_2, T_3,…,T_n. , όπου n η n-οστή φορά που κλήθηκε η newtemp()).

- empty_list(): Επιστρέφει μια κενή λίστα.

- make_list(x): Δημιουργεί και επιστρέφει μια λίστα η οποία περιέχει την ετικέτα x.

- merge_list(list1, list2): Συγχωνεύει τη λίστες 1 και 2 σε μία λίστα την οποία και επιστρέφει ώς αποτέλεσμα.

- backpatch(list, z): Η list είναι μια λίστα που περιέχει ετικέτες απο τετράδες που το τελευταίο τελούμενο δεν έχει συμπληρωθεί. Αυτή λοιπόν είναι και η δουλειά της backpatch(), να γεμίσει δηλαδή το τελευταίο τελούμενο των τετράδων αυτών με την ετικέτα z.

Παρακάτω θα αναλύσουμε τι πρόσθετες γραμμές κώδικα στους παραπάνω κανόνες της συντακτικής ανάλυσης:

(2) Τετράδες με τελεστή εκχώρησης:

Σε μόνο έναν κανόνας συναντάμε την εντολή εκχώρησης και αυτός είναι ο κανόνας <assignment-stat>

- Κανόνας <assignment-stat> : id := [expression]

Απ την κλήση της expression() περιμένουμε ένα αποτέλεσμα. Δε κάνουμε απλή κλήση όπως στη συντακική ανάλυση αλλά αποθηκεύομε το αποτέλεσμά της σε μια μεταβλητή E.

Πλέον, έχωντας το αποτέλεσμα της expression() μπορούμε να καλέσουμε τη genquad() ώστε να παράξουμε την τετράδα που θέλουμε.

genquad(":=",E,"__",id)

(1) Τετράδες με τελεστή αριθμητικών πράξεων:

Οι κανόνες στους οποίους συναντάμε αριθμητικές πράξεις (<add-oper>, <mul-oper>) είναι

οι: <expression> και <term>.

- Κανόνας <expression> : [optional-sign] [term] ( [add-oper] [term] )*

Απ την κλήση της term() περιμένουμε ένα αποτέλεσμα. Δε κάνουμε απλή κλήση όπως στη συντακική ανάλυση αλλά αποθηκεύομε το αποτέλεσμά της σε μια μεταβλητή T1.Αν μια εντολή περιέχει παραπάνω απο 1 πράξεις πρόσθεσης σημαίνει πως η επανάληπτική διαδικασία της εύρεσης συμβόλου “+” ή “-”, θα γίνει παραπάνω απο 1 φορά.Για το λόγο αυτό μέσα την επαναληπτική διαδικασία , πρώτα παράγουμε μια προσωρινή μεταβλητή w καλώντας την newtemp() και ύστερα αποθηκεύουμε το αποτέλεσμα της term() σε μια μεταβλητή T2. Πλέον παράγουμε την αντίστοιχη τετράδα καλώντας την genquad() και αποθηκεύοντας το αποτέλεσμα της πράξης στην νέα προσωρινή μεταβλητή.

genquad(“+” | “-” ,T1,T2, w)

Τέλος, ύστερα απο κάθε κλήση της genquad() η μεταβλητή T1 παίρνει τη τιμή της προσωρίνής μεταβλητής w η οποία έχει το αποτέλεσμα της πράξης. Έτσι αν συναντήσουμε πράξη με παραπάνω απο 2 μεταβλητές, το αποτέλεσμα των πρώτων 2 θα αποθηκευθεί στην μεταβλητή T1 και κατα την 2 επανάληψη θα έχουμε ολοκληρώσει τη διαδικασία αφού θα παραχθεί νέα τετράδα με T1 πλέον όμως το αποτέλεσμα της 1 πράξης.Η συνάρτηση expression() επιστρέφει τη τελευταία προσωρινή μεταβλητή που παράχθηκε κατά την επαναληπτική διαδικασία.

- Κανόνας <term> : [factor] ( [mul-oper] [factor] )*

Στον κανόνα term() δουλέψαμε ακριβώς όπως στον κανόνα <expression> μόνο που η επαναληπτική διαδικασία αφορά την εύρεση συμβόλου "*" ή "/". Στη προκειμένη περίπτωση η factor() επιστρέφει το δικό της αποτέλεσμα αποθηκεύοντάς το στις μεταβλητές F1,F2.Μπορεί η συνάρτηση factor() να μη περιέχει κανόνες αριθμητικών πράξεων, παρόλα αυτά είναι απαραίτητη για τη λειτουργία των παραπάνων κανόνων.

- Κανόνας <factor> : constant | ( [expression] ) | id [idtail]

Η συνάρτηση factor() επιστρέφει είτε τη σταθερά, είτε το αποτέλεσμα της κλήσης της expression() είτε τη λ.μ. τύπου id εφόσον η κλήση του κανόνα idtail() μας ειδοποιήσει οτι δεν καλείται συνάρτηση. Η idtail() μας ειδοποιεί οτι δεν υπάρχει κλήση συνάρησης αν η λίστα παραμέτρων που επιστρέφει είναι κενή.

Αν η λίστα που επιστρέφει η idtail() δεν είναι κενή, σημαίνει πως υπάρχει κλήση συνάρτησης, κάτι το οποίο θα αναλυθεί εκτενέστερα στην ενότητα (7).

(3) Τετράδες με τελεστή άλματος άνευ προύποθέσεων:

(4) Τετράδες με τελεστή συνθήκης και άλματος:

Οι κανόνες στους οποίους συναντάμε συνθήκες άλματος, δηλαδή τους κανόνες που περιέχουν τη κλήση της συνάρησης condition(), περιμένουν απ αυτήν 2 λίστες. Η πρώτη λίστα την οποία θα ονομάσουμε Btrue, περιέχει τις ετικέτες των τετράδων απ τις οποίες λείπει ο τελευταίος τελεστέος και περιμένουν να γεμίσουν

με την ετικέτα της τετράδας στην οποία θα κάνουμε άλμα αν η συνθήκη είναι αληθής. Η δεύτερη λίστα την οποία θα ονομάσουμε Bfalse, είναι ίδια με την Btrue μόνο που οι ετικέτες των τετράδων της λίστας αυτής αφορούν το άλμα στην ετικέτα της τετράδας αν η συνθήκη είναι ψευδής.Αρχικά θα μιλήσουμε για το πώς παράγονται αυτές οι λίστες Btrue,Bfalse και στη συνέχεια για τις συναρτήσεις που περιέχουν τη κλήση της condition().Ο μόνος κανόνας που περιέχει τη κλήση συνάρησης relational_oper() είναι ο <boolfactor>.

- Κανόνας <boolfactor> : not [ [condition] ] | [ [condition] ] |

  [expression] [relational-oper] [expression]

Ο κανόνας boolfactor στη περίπτωση που η λ.μ. δεν είναι ούτε “not”, ούτε “[“ καλεί απευθείας την expression(). Απ αυτή περιμένει πλέον ένα αποτέλεσμα το οποίο αποθηκεύεται στη μεταβλητή E1. Ύστερα καλεί την relational_oper() και αποθηκεύει στην μεταβλητή E2 το αποτέ-λεσμα της 2ης κλήσης της expression().Εδώ ξεκινάνε να γεμίζουν η λίστες Btrue και Bfalse με ετικέτες τετράδων που ο τελευταίος τελεστέος είναι άδειος. Ο τρόπος είναι απλός. Αποθηκεύουμε στη Btrue την ετικέτα της επόμενης τετράδας που θα παραχθεί και παράγουμε την επόμενη τετράδα καλώντας την genquad() με άδειο το τελευταίο τελεστέο.

genquad(relop,E1,E2,_)

Ύστερα αποθηκεύουμε στη Bfalse την ετικέτα της επόμενης τετράδας που θα παραχθεί η οποία δεν είναι άλλη απ την εντολή άλματος χωρίς προϋποθέσεις.

genquad(jump,_,_,_)

Με λίγα λόγια η Btrue θα περιέχει τις τετράδες που αν η συνθήκη ισχύει θα κάνει άλμα στην ετικέτα του τελευταίου τελεστέου. Αν δεν ισχύει τότε θα πάμε αμέσως στην επόμενη τετράδα που είναι εντολή άλματος χωρίς προϋποθέσεις, η οποία θα μας στείλει στην τετράδα που πρέπει να εκτελεστεί. Προσπερνάει ουσιαστικά τις τετράδες που δεν πρέπει να εκτελεστούν (πχ: αν η συνθήκη της εντολής while δεν ισχύει τότε θα πρέπει να κάνουμε άλμα στη 1η τετράδα που θα συναντήσουμε έξω απ το block της while).Στη περίπτωση που η λ.μ. είναι η “[” τότε καλείται η boolterm() και αποθηκεύουμε τις λίστες που επιστρέφει στην Brue και στη Bfalse.Αντίθετα αν η λ.μ. είναι η “not“ τότε ανιστρέφουμε τις δύο λίστες που επιστρέφει η boolterm() και τις αποθηκεύουμε στην Btrue και στη Bfalse.Και στις 3 περιπτώσεις η συνάρτηση boolfactor() επιστρέφει ώς αποτέλεσμά τις λίστες Btrue και Bfalse.

- Κανόνας <boolterm> : [boolfactor] ( and [boolfactor] )*

Ο κανόνας boolterm() περιμένει απ τη συνάρτηση boolfactor() δύο λίστες την B1true και τη B1false τις οποίες και αποθηκεύει τοπικά. Κατά την επαναληπτική διαδικασία ελέγχου της λ.μ. “and” καλούμε απευθείας την backpatch() ώστε να γεμίσουμε της άδειες τετράδες των οποίων οι ετικέτες περιέχονται στη Btrue, με την ετικέτα της επόμενης τετράδας που θα παραχθεί. Με το τρόπο αυτό, αν μια συνθήκη ικανοποιείται και συναντήσουμε “and” σημαίνει πως πρέπει να κάνουμε άλμα και στην επόμενη τετράδα που θα παραχθεί η οποία περιέχει τη 2η συνθήκη άλματος.Έπειτα λαμβάνουμε τις νέες λίστες B2true, B2true απ την εκ νέου κλήση της boolfactor() και συνενώνουμε την B2false στην B1false. Συνενώνουμε τις 2 αυτές λίστες γιατί αν έστω και μια απ τις 2 συνθήκες δεν ισχύει τότε θα πρέπει να κάνουν άλμα στην ίδια ακριβώς ετικέτα. Τέλος, θέτουμε ως B1true την B2true ώστε μετά τον επιτυχή έλεγχο όλων των συνθηκών να μεταβούμε στη 1η τετράδα που περιέχεται στο block κώδικα που θα εκτελεστεί.Η συνάρτηση boolterm() επιστρέφει ώς αποτέλεσμά τις λίστες B1true και B1false.

- Κανόνας <condition> : [boolterm] ( or [boolterm] )*

Ο κανόνας condition() περιμένει απ τη συνάρτηση boolfactor() δύο λίστες την B1true και τη B1false τις οποίες και αποθηκεύει τοπικά. Κατά την επαναληπτική διαδικασία ελέγχου της λ.μ. “or” καλούμε απευθείας την backpatch() ώστε να γεμίσουμε της άδειες τετράδες των οποίων οι ετικέτες περιέχονται στη Bfalse, με την ετικέτα της επόμενης τετράδας που θα παραχθεί. Με το τρόπο αυτό, αν μια συνθήκη δεν ικανοποιείται και συναντήσουμε “or” σημαίνει πως πρέπει να κάνουμε άλμα και στην επόμενη τετράδα που θα παραχθεί η οποία περιέχει την επόμενη συνθήκη άλματος.Έπειτα λαμβάνουμε τις νέες λίστες B2true, B2true απ την εκ νέου κλήση της boolterm() και συνενώνουμε την B2true στην B1true. Συνενώνουμε τις 2 αυτές λίστες γιατί μόνο αν όλες οι συνθήκες ισχύουν θα γίνει άλμα στη 1η τετράδα που περιέχεται στο block κώδικα που θα εκτελεστεί. Τέλος, θέτουμε ως B1false την B2false ώστε αν καμία απ τις συνθήκες δεν ισχύει να μπρούμε να κάνουμε άλμα στη 1η τετράδα έξω απ το block κώδικα της συνάρτησης που μας κάλεσε.Η συνάρτηση condition() επιστρέφει ώς αποτέλεσμά τις λίστες B1true και B1false.Να σημειωθεί πως η condtion() κάνει backpatch() όχι σε όλες τις τετράδες αλλά σε αυτές που έχει δώσει ώς αποτέλεσμα η boolterm(). Η boolterm() έχει φροντίσει να γεμίσει τις τετράδες που την αφορούν.

Πρίν αναφερθούμε στους κανόνες που περιέχουν τη κλήση της condition() απ την οποία περιμένουν ως αποτέλεσμα 2 λίστες, τις Btrue,Bfalse θα μιλησουμε για την συνάρτηση loop_stat() της οποίας η διαδικασία παραγωγής ενδιάμεσου κώδικα διαφοροποιεί κάπως τις υπόλοιπες συναρτήσεις.

- Κανόνας <loop-stat> : loop [statements] endloop

Η συνάρτηση loop_stat() πριν κάν καλέσει τη statements() αποθηκεύει σε μια προσωρινή μεταβλήτή firstQuad τη πρώτη τετράδα που θα παραχθεί καλώντας την nextquad(). Ύστερα απ την κλήση της statements() δημιουργεί μια τετράδα άλματος άνευ προϋποθέσεων με ετικέτα την firstQuad. Έτσι κάθε φορά που θα φτάνει στο τέλος θα κάνει άλμα στην αρχή του block του κώδικά της.

Πρίν τη κλήση της statements() δημιουργείτε μια κενή τοπική λίστα καλώντας την empty_list(), περνώντας την λίστα αυτή ώς παράμετρο στην statements().* Αυτή επιστρέφει μια λίστα με τις ετικέτες των τετράδων που περιέχουν εντολή άλματος άνευ προϋποθέσεων (εντολή exit). Έτσι κάνουμε backpatch() με όρισμα αυτή τη λίστα και ετικέτα την τετράδα που θα παραχθεί μετά το τέλος του block κώδικα της <loop-stat>.

* Για να κληθεί πλέον η statements() χρειάζεται ώς όρισμα τη τοπική αυτή λίστα. Συνεπώς, και η συνάρτηση statement() αλλά και όλες οι συναρτήσεις που καλούνται απο την statement() (if_stat(), while_stat(), do_while_stat(), loop_stat(), exit_stat(), forcase_stat(), incase_stat()) χρειάζονται για να κληθούν ώς όρισμα αυτή τη λίστα.

- Κανόνας <if-stat> : if ( [condition] ) then [statements] [elsepart] endif

Η συνάρτηση if_stat() λαμβάνει απ την κλήση της condition() τις λίστες Btrue,Bfalse. Εφόσον υπάρχει η λ.μ. “then” καλεί τη backpatch() με όρισμα τη λίστα Btrue και ετικέτα της επόμενης τετράδας που θα παραχθεί, καθώς η επόμενη τετράδα είναι και η 1η εντολή στο block κώδικα του κανόνα <if-stat>.

Ύστερα απ τη κλήση της statements() καλούμε τη make_list() βάζοντας σε αυτήν την επόμενη τετράδα που θα παράξουμε μετά την επιστροφή της statements(). Η επόμενη τετράδα που θα παράξουμε καλώντας την genquad() θα είναι μια εντολή άλματος άνευ προϋποθέσεων. Σε περίπτωση δηλαδή που ισχύει η συνθήκη και παραχθούν όλες οι τετράδες του block <statements> να μπορέσουμε να

κάνουμε άλμα έξω απ το block του <elsepart>. Αυτό θα γίνει με τη κλήση της backpatch() μετά τη κλήση του <elsepart>.

genquad(jump,_,_,_)

Έπειτα καλούμε τη backpatch() με όρισμα τη λίστα Bfalse και την επόμενη τετράδα που θα παραχθεί, ώστε αν οι συνθήκες δεν ικανοποιούνται να γίνει άλμα στην 1η εντολή του block κώδικα του κανόνα <elsepart>.Τέλος, μετά τη κλήση και της elsepart() καλούμε τη backpatch() με όρισμα τη λίστα που δημιουργήσαμε προηγουμένως και την ετικέτα της επόμενης τετράδας που θα παραχθεί. Η επόμενη τετράδα που θα παραχθεί θα είναι η 1η τετράδα έξω απ το block κώδικα του <if-stat>.

- Κανόνας <while-stat> : while ( [condition] ) [statements] endwhile

Η συνάρτηση while_stat() πριν κάν ελέγξει αν βρεί τη λ.μ. “while” αποθηκεύει σε μια προσωρινή μεταβλήτή firstquad τη πρώτη τετράδα που θα παραχθεί καλώντας την nextquad().Ύστερα απ τη κλήση της condition() καλούμε τη backpatch() με όρισμα τη Brue και την ετικέτα της επόμενης τετράδας που θα παραχθεί, ώστε αν ισχύσουν οι συνθήκες να κάνει άλμα στη 1η εντολή του block της <while-stat>.Τέλος, αφού γίνει ο έλεγχος για την λ.μ. “endwhile”, παράγουμε μια τετράδα άλματος άνευ προϋποθέσεων και ως ετικέτα άλματος βάζουμε τη πρώτη τετράδα που αρχικοποιήσαμε πρίν ξεκινήσει η μετάφραση της <while-stat>.genquad(jump,_,_,firstquad). Αυτό γίνεται γιατί στο τέλος τους block της <while-stat> πρέπει να μεταβούμε εκ νέου στην 1η τετράδα, στον έλεγχο συνθήκης δηλαδή.Αμέσως μετά, καλούμε τη backpatch() με όρισμα τη Bfalse και την ετικέτα της επόμενης τετράδας. Αν δεν ισχύσουν δηλαδή, οι συνθήκες να μεταβούμε εκτός του block της <while-stat>.

- Κανόνας <do-while-stat> : dowhile [statements] enddowhile ( [condition] )

Η συνάρτηση do_while_stat() πριν κάν ελέγξει αν βρεί τη λ.μ. “dowhile” αποθηκεύει σε μια προσωρινή μεταβλήτή sQuad τη πρώτη τετράδα που θα παραχθεί καλώντας την nextquad().Μετά τη κλήση της condition() και αφού ληφθούν τις λίστες Btrue,Bfalse καλείται η backpatch() με όρισμα τη Bfalse και την ετικέτα sQuad που είχε παραχθεί στην αρχή. Με το τρόπο αυτό, αν η συνθήκη δεν ισχύει μεταβένουμε εκ νέου στην 1η τετράδα του block κώδικα της <while-stat>.Τέλος, ξανακαλούμε την backpatch() με όρισμα αυτή τη φορά την Brue και την επόμενη τετράδα που θα παραχθεί, η οποία δεν είναι άλλη απ την 1η τετράδα έξω απ το block κώδικα της <while-stat>.

- Κανόνας <forcase-stat> : forcase ( when ( [condition] ) : [statements] )*
  default : [statements] enddefault endforcase

Η συνάρτηση forcase_stat() πριν κάν ελέγξει αν βρεί τη λ.μ. “when” αποθηκεύει σε μια προσωρινή μεταβλήτή firstQuad τη πρώτη τετράδα που θα παραχθεί καλώντας την nextquad(). Επίσης δημιουργεί και μια κενή λίστα εξόδου καλώντας την empty_list(). Η κλήση της condition() επιστρέφει 2 λίστες Btrue,Bfalse και απευθείας καλεί την backpatch() με όρισμα την Btrue και ετικέτα της επόμενης τετράδας που θα παραχθεί. Μετά τη κλήση της statements() παράγει μια τετράδα άλματος άνευ προϋποθέσεων και καλεί την backpatch() με όρισμα την λίστα Bfalse και ετικέτα την επόμενη τετράδα που θα παραχθεί.Κάθε φορά που βρίσκει λ.μ. “when” συνενώνει τις λίστες εξόδου κρατώντας σε αυτήν τις ετικέτες των τετράδων με εντολές άλματος άνευ προϋποθέσεων που δημιουργήσαμε παραπάνω.

genquad(jump,_,_,_)

Μόλις συναντήσουμε λ.μ. “default” και κληθεί η statements() παράγουμε ακόμη μια τετράδα άλματος άνευ προϋποθέσεων με το τελευταίο τελεστέο να περιέχει την ετικέτα firstQuad.

genquad(jump,_,_,firstQuad)

Η συνάρτηση forcase-stat() πρίν τερματίσει καλεί την backpatch() με όρισμα τη λίστα εξόδου και ετικέτα την επόμενη τετράδα που θα παραχθεί έξω απ το block κώδικα της <forcase-stat>.

- Κανόνας <incase-stat> : incase ( when ( [condition] ) : [statements] )*
  endincase

Η συνάρτηση incase-stat() έχει την εξής ιδιαιτερότητα: Μεταβαίνει εκτός της επανάληψης μόνο εάν κανένα απ τα statements δεν εκτελεσθει.Για το λόγο αυτό πρίν καν γίνει ο έλεγχος για τη λ.μ. “when” παράγουμε μια τετράδα flag καλώντας την genquad. Όταν έστω ένα απ τα statements εκτελεσθει η προσωρινή μεταβλήτή t θα πάρει τη τιμή “1” και έτσι θα καταλάβουμε πότε θα πρέπει να κάνουμε άλμα εκτός της block κώδικα της <incase-stat>.

genquad(:=,0,_,t)

Επίσης αποθηκεύουμε και σε μια προσωρινή μεταβλητή τη πρώτη τετράδα που θα παραχθεί. Αυτό γίνεται επίσης πρίν τον έλεγχο για τη λ.μ. “when”.

Μετά τη κλήση της condition() κάνουμε backpatch() με όρισμα τη Btrue και την ετικέτα της επόμενης τετράδας που θα παραχθεί και θέτουμε το flag της προσωρινής μεταβλητής να είναι 1, πράγμα που σημαίνει ότι έστω ένα statement ισχύει.

genquad(:=,1,_,t)

Αφού κληθεί και η statements() καλούμε τη backpatch() με όρισμα τη Bfalse και την ετικέτα της επόμενης τετράδας που θα παραχθεί.

Τέλος πρίν τον έλεγχο για τη λ.μ. “endincase” δημιουργούμε μια τετράδα άλματος με τη προϋπόθεση ότι η προσωρινή μεταβλητή T_1 ισούται με 1 ώστε να κάνουμε άλμα στη 1η τετράδα του block κώδικα της <incase-stat>. Άν δηλαδή το flag είναι 1 σημαίνει πως έστω και ένα statement εκτελέσθηκε και πως πρέπει να επιστρέψουμε στην αρχή της δομής <incase-stat>.

genquad("=","1",t,flagQuad)

- Κανόνας <exit-stat> : exit

Η συνάρτηση exit-stat() καλεί την genquad() παράγοντας μια τετράδα άλματος άνευ προϋποθέσεων. Κάποια στιγμή θα συμπληρωθεί ο τελευταίος τελεστέος της τετράδας αυτής με ετικέτα την 1η τετράδα εκτός της δομής <loop-stat>.

genquad(jump,_,_,_)

## (5) Τετράδες αρχής και τέλος ενότητας

- Κανόνας <block> : [declarations] [subprograms] [statements]

Η συνάρτηση block() πρίν τη κλήση της συνάρτησης statements() καλεί τη genquad() δημιουργώντας μια τετράδα αρχής ενότητας. Ώς 1ο τελούμενο θέτει το όνομα της συνάρτησης-κυρίου προγράμματος που ανήκει το block.

genquad(begin_block,blockID,_,_)

Αμέσως μετά την επιστροφή της statements() δημιουργή τη τετράδα κλεισίματος της συγκεκριμένης ενότητας που ως 1ο τελούμενο θέτει το όνομα της συνάρτησης-κυρίου προγράμματος που ανήκει το block.Τέλος, άν το το block

ανήκει στο κύριο πρόγραμμα, δημιουργείται ακόμη μια τετράδα τερματισμού προγράμματος πρίν απ τη τετράδα κλείσίματος της ενότητας.

## (7) Τετράδες που αφορούν τις εντολές εισόδου εξόδου

- Κανόνας <print-stat> : print [expression]

Η συνάρτηση print_stat() πλέον απ την κλήση της expression() περιμένει ένα αποτέλεσμα. Καλείται η genquad() λοιπόν και δημιουργείται μια τετράδα με τελεστή εξόδου και 1ο τελούμενο το αποτέλεσμα της expression().

genquad(out,E,_,_)

- Κανόνας <input-stat> : input id

Η συνάρτηση print_stat() μετά τον έλεγχο της λ.μ. τύπου id, καλεί τη genquad() και δημιουργεί μια τετράδα με τελεστή εισόδου και 1ο τελούμενο τη λ.μ. τύπου id.

## (6) Τετράδες που αφορούν τη κλήση συνάρτης και πέρασμα παραμέτρων

Προηγουμένως, στη συνάρτηση factor() αναφέραμε το τι επιστρέφεται ώς αποτέλεσμα αν η κλήση της idtail() έχει ώς αποτέλεσμα κενή λίστα, δηλαδή όταν δεν υπάρχει κλήση συνάρησης. Παρακάτω θα εξηγήσουμε το πώς δημιουργούνται οι τετράδες κλήσης συνάρτησης, περάσματος παραμέτρων, επιστροφής τιμής και επιστροφή αποτελέσματος <return-stat>.

- Κανόνας <factor> : constant | ( [expression] ) | id [idtail]

Αν η λίστα των παραμέτρων που επιστρέφει η idtail() δεν είναι κενή τότε διατρέχουμε τη λίστα αυτή και καλούμε τη genquad() με το ονόμα της παραμέτρου και το τύπο που υπάρχουν αποθηκευμένα στο τρέχων στοιχείο της λίστας.

genquad(par,list[i][0],list[i][1],_), όπου list[i] το τρέχων στοιχείο της λίστας.

Να σημειώσουμε πως η λίστα των παραμέτρων παράγεται στην συνάρτηση actualaparlist().Αφού διατρέξαμε τη λίστα καλούμε τη newtemp() ώστε να αποθηκεύσουμε στη νέα προσωρινή μεταβλητή w τη τιμή επιστροφής της συνάρτησης που θα κληθεί και αμέσως καλούμε τη genquad() για να παράξουμε τη τετράδα.

genquad(par,w,RET,_)

Τέλος καλούμε ακόμη μια φορά τη genquad() για να δημιουργήσουμε τη τετράδα κλήσης συνάρησης.

genquad(call,FN,_,_)

Στη περίπτωση αυτή η συνάρτηση factor() επιστρέφει τη νέα προσωρινή μεταβλητή w.

- Κανόνας <idtail> : ε | [actualpars]

Η συνάρτηση idtail() παίρνει ώς αποτέλεσμα απ την κλήση της συνάρτησης actualpars() μια λίστα παραμέτρων. Είτε η λίστα είναι κενή είτε περιέχει κάποιες παραμέτρους η idtail() επιστρέφει αυτή τη λίστα.

- Κανόνας <actualpars> : ( [actualparlist] )

Η συνάρτηση actualpars() παίρνει ώς αποτέλεσμα απ την κλήση της συνάρτησης actualparlist() μια λίστα παραμέτρων. Είτε η λίστα είναι κενή είτε περιέχει κάποιες παραμέτρους η actualpars() επιστρέφει αυτή τη λίστα.

- Κανόνας <actualparlist> : [actualparitem] ( , [actualparitem] )* | ε

Η συνάρτηση actualparlist() πριν τη κλήση της actualparitem() δημιουργεί μια κενή λίστα καλώντας την συνάρτηση empty_list(). Απο τη κλήση πλέον της actualparitem() λαμβανει ώς αποτέλεσμα το όνομα και το τύπο της παραμέτρου (E1: πίνακας 2 στοιχείων, E1[0]: το όνομα της παραμέτρου, E1[1]: το τύπου της παραμέτρου) το οποίο αποτέλεσμα το τοποθετεί στη λίστα που δημιούργησε. Κάθε φορά που καλέιται η actualparitem() και επιστρέφει ώς αποτέλεσμα το όνομα

και το τύπο της παραμέτρου, το αποτέλεσμα αυτό τοποθετείται στη λίστα των παραμέτρων.Με το τέλος της επαναληπτικής διαδικασίας εύρεσης λ.μ. “,”, η actualparlist() επιστρέφει τη λίστα των παραμέτρων που δημιούργησε.

- Κανόνας <actualparitem> : in [expression] | inout id | inandout id

Η συνάρτηση actualparitem() αποθηκεύει σε ένα πίνακα 2 στοιχείων το τύπο της παραμέτρου και το όνομά της το οποίο είτε είναι η λ.μ. τύπου id είτε το αποτέλεσμα της κλήσης της expression().

- Κανόνας <return-stat> : return [expression]

Η συνάρτηση return_stat() καλεί τη genquad() και δημιουργεί μια τετράδα επιστροφή τιμής (retv) με 1ο τελούμενο το αποτέλεσμα της κλήσης της expression().

genquad(retv,E,_,_)

## ΣΗΜΑΣΙΟΛΟΓΙΚΗ ΑΝΑΛΥΣΗ

Είναι απαραίτητο να τερματίζουμε τη μετάφραση του προγράμματος αν μια συνάρτηση δεν περιέχει στο block κώδικα της τουλάχιστον μια εντολή return ή αν υπάρχει εντολή return εκτός συνάρτησης.

Για το λόγο αυτό προσθέσαμε κάποιες επιπλέον ενέργειες και ελέγχους στις συναρτήσεις program(), funcbody(), statements(), statement().

- Κανόνας <funcbody> : [formalpars] [block]

Στη συνάρτηση funcbody(), πριν τη κλήση της block() δημιουργούμε μια λίστα ενός στοιχείου το οποίο περιέχει τη πληροφορία του αν βρέθηκε εντολή return στο κώδικα της συνάρτησης που επρόκειτο να μεταφραστεί. Αρχικά η τιμή δεν είναι αληθής αφού δεν έχουμε ακόμα συναντήσει κάποια.Η πληροφορία αυτή περνάει ώς παράμετρος στη κλήση της συνάρτησης block(). Η block() πλέον χρειάζεται μια παράμετρο για να μπορέσει κάποια συνάρτηση να τη καλέσει. Αυτό επιφέρει μια

σειρά αλλαγών αφού πλέον και η statements() και η statement() για να κληθούν χρειάζονται ώς παράμετρο τη πληροφορία αυτή.Αν κατά την επιστροφή της κλήσης της block() η λίστα αυτή δεν έχει ενημερωθεί, δηλαδή δε βρεθεί εντολή return (η τιμή του μοναδικού στοιχείου της λίστας είναι ακόμα ψευδης) διακόπτουμε τη μετάφραση του προγράμματος εμφανίζοντας μήνυμα λάθους.Η λίστα ενός στοιχείου που διατηρεί τη πληροφορία για το άν έχει βρεθεί εντολή return είναι εμφανής και μπορεί να ενημερωθεί απο τις συναρτήσεις που έχουν δικαίωμα να την ενημερώσουν.

- Κανόνας <statement> :

Πλέον η statement() για να κληθεί, χρειάζεται ώς παράμετρο τη πληροφορία του άν βρέθηκε μέχρι στιγμής εντολή return. Στη συνάρτηση statement() άν απο τον έλεγχο της λ.μ. διαπιστώσουμε πως είναι η λέξη “return” τότε είμαστε υποχρεωμένοι τη πληροφορία που λάβαμε ώς παράμετρο να την ενημερώσουμε, ώστε πλέον να είναι γνωστό ότι βρέθηκε εντολή return.

- Κανόνας <program> : program id [block] endprogram

Στη συνάρτησηprogram() δουλεύουμε ακριβώς όπως στην funcbody(), δημιουργώντας μια λίστα ενός στοιχείου το οποίο περιέχει τη πληροφορία του αν βρέθηκε εντολή return στο κώδικα του κυρίου προγράμματος. Η διαφοροποίηση είναι πως άν κατά την επιστροφή της κλήσης της block() η λίστα αυτή έχει ενημερωθεί, δηλαδή έχει βρεθεί εντολή return (η τιμή του μοναδικού στοιχείου της λίστας είναι ακόμα αληθής) διακόπτουμε τη μετάφραση του προγράμματος εμφανίζοντας μήνυμα λάθους.

Είναι απαραίτητο να τερματίζουμε τη μετάφραση του προγράμματος αν μια εντολή exit βρίσκεται εκτός του κανόνα <loop-stat>.Για το λόγο αυτό δημιουργούμε μια καθολική μεταβλητή μέσα στη συνάρτηση loop_stat() αρχικοποιώντας την με τη τιμή αληθής. Η τιμή ξαναγίνεται ψευδής όταν συναντήσουμε λ.μ. “endloop”.Αυτή η τιμή ελέγχεται στη συνάρτηση της exit_stat() και και σε περίπτωση που είναι

ψευδής, πράγμα που σημαίνει πως βρίσκεται εκτός του block του κανόνα <loop-stat>, τερματίζουμε τη μετάφραση εμφανίζοντας ανάλογο μήνυμα λάθους.

## ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

Κατα τη διάρκεια της μεταγλώττισης υπάρχει η ανάγκη για τη συγκέντρωση πληροφοριών σχετικά με τα ονόματα που βρίσκονται μέσα στο πρόγραμμα. Ως όνομα ορίζουμε μεταβλητές, συναρτήσεις, σταθερές, παραμέτρους και προσωρινές μεταβλητές. Η πληροφορία που αποθηκεύεται στον πίνακα συμβόλων χρησιμοποιείται κατα τη σημασιολογική ανάλυση (έλεγχος τύπων) και κατα την παραγωγή *τελικού κώδικα* (π.χ. πόσο χώρο στη μνήμη απαιτούν). Για παράδειγμα, εαν συναντήσει ο μεταγλωττιστής μας, μια δήλωση int x, θα γίνει αποθήκευση στον πίνακα συμβόλων πως το x είναι ακέραιος και κατα την παραγωγή του τελικού κώδικα, θα γίνει ανάκληση απο τον πίνακα συμβόλων (π.χ. ανάθεση τιμής στη x). Ο πίνακας συμβόλων κατασκευάζεται στη φάση της λεκτικής και σημασιολογικής ανάλυσης. Κυρίως η τροποποίηση του πίνακα γίνεται στη συντακτική ανάλυση, καθώς εκεί υπάρχει πληροφορία για το συγκεκριμένο όνομα.

Για την ομαδοποίηση των ονομάτων και την διαχείρηση της πολυπλοκότητας του προβλήματος που παρουσιάζεται, θα χρησιμοποιήσουμε **3 οντότητες**. Δημιουργούμε την κλάση **Entity**, με την οποία θα αναπαραστήσουμε μεταβλητές, συναρτήσεις, σταθερές, παραμέτρους και προσωρινές μεταβλητές. Η κλάση entity έχει τα εξής πεδία:

- name (Το ονομα)
- entitype (ο τύπος του entity, πχ Συνάρτηση)
- offset (Το ***offset** αρχικοποιείται στο 0 )
- paramType (ο τύπος περάσματος της παραμέτρου, με αναφορά, τιμή κτλπ)
- frameLength (αρχικοποιείται στο 0, ενημερώνεται βάσει το μήκος της συνάρτησης)
- startQuad (περιέχει τον αριθμό ετικέτας της πρώτης τετράδας, του begin block)
- NextEntity (δείκτης στο επόμενο Entity)
- Arguments[] (λίστα απο **arguments**, θα διευκρινιστεί στη συνέχεια)

*Το **offset** είναι ακέραιος που δείχνει την απόσταση απο τον δείκτη στοίβας **sp.** Η πρόσβαση στη στοίβα γίνεται μέσω ενός καταχωρητή που ονομάζεται δείκτης στοίβας (stack pointer), ο οποίος και επίσης χρησιμοποιείται για να δείχνει την κορυφή της στοίβας. Εαν κληθεί μια συνάρτηση, ο sp θα δείξει την αρχή της συνάρτησης(την κορυφή της στοίβας). Παρομοιως, υπάρχει και ο frame pointer, **fp,** ο οποίος δείχνει στο τέλος του πλαισίου.

Κάθε φορά που δημιουργούμε ενα αντικείμενο entity, ο constructor ενημερώνει το όνομα και δημιουργεί μια λίστα arguments[]. Τώρα θα πρέπει να διακρίνουμε περιπτώσεις. Εαν ο *συντακτικός αναλυτής* βρήκε μια μεταβλητή, πρώτα θα δημιουργήσει ενα αντικείμενο entity, και στη συνέχεια θα καλέσει τη συνάρτηση initVariable(self,offset), η οποία θα ενημερώσει το πεδίο entitype με την τιμή “variable” και το offset. Το offset για μια συνάρτηση αρχικοποιείται στο 12 και ενημερώνεται ανάλογα με το τι έχουμε συναντήσει. Αν βρούμε integer τοτε offset += 4, γιατί 4 bytes ο ακέραιος. Εαν συναντήσουμε παράμετρο καλείται η συνάρτηση initParameter(self,paramtype,offset), η οποία ενημερώνει το πεδίο entitype του αντικειμένου με την τιμή “parameter”, το πεδίο paramtype με την τιμή που δόθηκε στο σημείο που καλέστηκε (πχ πέρασμα με αναφορά, inout) και το πεδίο offset. Η τρίτη περίπτωση είναι να συναντήσουμε συνάρτηση, όπου τότε καλείται η initFunction(self,framelength,startQuad), η οποία ενημερώνει το entitype, το υπολογισμένο frameLength της και το πεδίο startQuad με τον αριθμό ετικέτας της τετράδας της. Επίσης, υπάρχει η συνάρτηση appendArgument(self,argument) η οποία προσθέται ενα αντικείμενο *argument* στη λίστα arguments[] του entity που την κάλεσε, και η συνάρτηση setNext(self,entity), η οποία ενημερώνει τον δείκτη nextEntity. Τέλος, χρησιμοποιήσαμε συναρτήσεις getters και setters για την προσπέλαση των πεδίων κάθε entity.

Η δεύτερη οντότητα που χρειαστήκαμε, ήταν για αναπαραστήσουμε ομαδοποιήσεις απο entities (π.χ. enitites της ίδιας συνάρτησης) και βάθη φωλιάσματος (το χρησιμοποιούμε για τον εντοπισμό ενός **εγγραφήματος δραστηριοποίησης** , εξηγούμε παρακάτω). Δημιουργήσαμε λοιπόν την κλάση **Scope.** Η κλάση scope έχει τα εξής πεδία:

- scopeName
- offset=12
- entitylist=[]
- enclosingScope=""
- startQuad=""
- nestingLevel=0

Κάθε φορά που δημιουργούμε αντικείμενο scope, ο constructor αρχικοποιεί το πεδίο scopeName με το όνομα του scope, το πεδίο startQuad με ό,τι επιστρέφει η

συνάρτηση nextquad() (την έχουμε περιγράψει στο μέρος Ενδιάμεσος Κώδικας), το nestingLevel γίνεται ίσο με το scopelevel (καθολική μεταβλητή που κρατά το βάθος του συνολικού scope, μετά το αυξάνουμε και κατα ένα) και δημιουργούμε μια λίστα **entitylist[]**. Όπως προαναφέραμε, ο σκοπός του scope είναι να κρατήσει ομαδοποιημένα entities. Αρα θα πρέπει να δημιουργήσουμε μια συνάρτηση η οποία θα προσθέτει ενα entity στη λιστα entitylist[] του scope που την κάλεσε. Δημιουργούμε λοιπόν τη συνάρτηση addEntity(self,entity), η οποία προσθέτει το entity στη λίστα και αυξάνει το offset κατα 4. Στη συνέχεια, η συνάρτηση setStartQuad(self,startQuad) χρησιμοποιείται για να πάρουμε τον αριθμό ετικέτας, καλούμε όμως την nextquad() νωρίτερα, αρα θα πρέπει να προσθέσουμε self.startQuad=str(int_startQuad-1) ώστε να πάρουμε το τρέχον.Όταν θα χρειαστεί να μάθουμε το frame length μιας συνάρτησης, θα κληθεί η getFrameLength(self) η οποία υλοποιείται μεσα στην κλάση scope, και επιστρέφει το offset του συγκεκριμένου scope. Έχουμε επίσης υλοποιήσει τους setters και getters για την προσπέλαση των πεδίων κάθε αντικειμένου scope.

Η τρίτη οντότητα που χρειαστήκαμε, ήταν για να αναπαραστήσουμε τις παραμέτρους των συναρτήσεων. Συγκεκριμένα ορίσαμε την κλάση **argument** η οποία θα περιγράφει τον τρόπο περάσματος της παραμέτρου (cv, ref) καθώς και τον τύπο της παραμέτρου (π.χ. int). Η κλάση έχει τα εξής πεδία:

- parMode="" (τρόπος περάσματος)
- nextArg="" (δείκτης στο επόμενο argument)

Έχουμε επίσης υλοποιήσει και πάλι τους setters και getters για την προσπέλαση των πεδίων κάθε αντικειμένου argument.

Αφού εξηγήσαμε το πως δομήθηκαν σε κλάσεις οι 3 οντότητές μας, ας περιγράψουμε το πότε θα τροποποιείται απο την συντακτική ανάλυση ο πίνακας συμβόλων. Όταν ξεκινάμε τη μετάφραση μιας συνάρτησης θα κάνουμε *προσθήκη ενός scope.* Όταν τελειώνουμε τη μετάφραση μια συνάρτησης, *διαγράφουμε την εγγραφή του scope* της και μαζι με αυτό διαγράφονται και όλες τις λίστες με τα Entity και τα Argument που εξαρτώνται από αυτήν. Όταν ο συντακτικός μας αναλυτής συναντήσει μια δήλωση μεταβλητής ή δημιουργία προσωρινής μεταβλητής (newTemp() ) ή δήλωση νέας συνάρτησης ή δήλωση τυπικής παραμέτρου συνάρτησης, *δημιουργούμε ενα entity.* Αν συναντήσουμε δήλωση τυπικής παραμέτρου συνάρτησης κάνουμε *προσθήκη ενος argument.* Η τελευταία ενέργεια που κάνουμε στον πίνακα συμβόλων είναι η *αναζήτηση.* Μπορεί να αναζητηθεί κάποιο entity με βάση το όνομά του.Η αναζήτηση ενός entity γίνεται ξεκινώντας από την αρχή του πίνακα και την πρώτη του γραμμή.Αν δε βρεθεί πηγαίνουμε στην επόμενη γραμμή έως ότου βρεθεί το entity ή τελειώσουν όλα τα entities οπότε επιστρέφουμε και μήνυμα λάθους. Αν με το ζητούμενο όνομα υπάρχει πάνω από ένα entity τότε επιστρέφουμε το πρώτο που θα

συναντήσουμε.Το ίδιο κάνουμε και στις λίστες με arguments. Οι συναρτήσεις που είναι υπεύθυνες για τα πραπάνω είναι η new_variable_entity(entityname), η new_function_entity(entityname), η new_scope(scopename), η delete_scope(), η searchEntity(entityName) και η searchParametes(funcName,parList).

Όλα όσα περγράψαμε σε αυτό το κομμάτι της εργασίας, έγιναν με σκοπό τη διαχείρηση του **εγγραφήματος δραστηριοποίησης.** Το εγγράφημα δραστηριοποίησης δημιουργείται για κάθε συνάρτηση από αυτήν που την καλεί. Όταν αρχίζει η εκτέλεση της συνάρτησης ο δείκτης στοίβας μεταφέρεται στην αρχή του εγγρασήματος δραστηριοποίησης. Περιέχει πληροφορίες που χρησιμέυουν για την εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί. Όταν τερματίζεται η συνάρτηση ο χώρος που καταλαμβάνει το εγγράφημα δραστηριοποίησης επιστρέφεται στο σύστημα. Η η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης είναι η διεύθυνση επιστροφής. Ένα αλλο στοιχείο που θα χρησιμοποιήσουμε και στο τελευταίο κομμάτη του project, την **παραγωγή του τελικού κώδικα** είναι ο σύνδεσμος προσπέλασης. Ο σύνδεσμος προσπέλασης δείχνει στο εγγράφημα δραστηριοποίσης που πρέπει να αναζητηθούν μεταβλητές οι οποίες δεν είναι τοπικές αλλά η συνάρτηση έχει δικαίωμα να χρησιμοποιήσει. Ενώ η διεύθυνση στην οποία θα γραφεί το αποτέλεσμα της συνάρτησης όταν αυτό υπολογιστεί ονομάζεται επιστροφή τιμής. Στο εγγράφημα δραστηριοποίησης βρίσκεται και ο χώρος αποθήκευσης παραμέτρων συνάρτησης, στον οποίον αποθηκεύεται η τιμή, αν πρόκειται πέρασμα με τιμή, ή η διεύθυνση, αν πρόκειται για πέρασμα με αναφορά. Εκεί βρίσκεται επίσης και ο χώρος των προσωρινών και των τοπικών μεταβλητών.

## ΠΑΡΑΓΩΓΗ ΤΕΛΙΚΟΥ ΚΩΔΙΚΑ

Έχοντας φτάσει στο τελευταίο κομμάτι της μεταγλώττισής μας, το πρόγραμμά μας έχει ελέγξει για συντακτικά και σημασιολογικά λάθη, έχει συμπληρωθεί ο πίνακας συμβόλων και έχουν παραχθεί οι τετράδες του ενδιάμεσου κώδικα. Στο σημείο που βρισκόμαστε καλούμαστε να παράξουμε κώδικα assembly για την αρχιτεκτονική του MIPS. Ο MIPS διαθέτει ένα σύνολο καταχωρητών στους οποίους αποθηκεύει προσωρινά τις πράξεις που κάνει στο εσωτερικό του. Ορισμένοι απο τους πιο βασικούς είναι οι καταχωριτές προσωρινών τιμών ($t0 έως $t7), καταχωριτές των οποίων οι τιμές διατηρούντε ανάμεσα σε κλήσεις συναρτήσεων ($s0 εως $s7), καταχωρητές ορισμάτων ($a0 εως $a3), καταχωριτές επιστροφής τιμών ($v0 και $v1), τον stack pointer $sp, τον frame pointer $fp και τον καταχωρητή που κρατάει τη διεύθυνση που θα επιστρέψει ο έλεγχος μετά το πέρας μια συνάρτησης.

Ενδεικτικά, μερικές εντολές του MIPS, ορίζονται ως εξής:

- για πρόσθεση και αποθήκευση στον καταχωρητή $t0 με την τιμή του $t1
  add $t0,$t1,$t2
- για αφαίρεση και αποθήκευση στον καταχωρητή $t0 με την τιμή του $t1
  sub $t0,$t1,$t2
- για πολλαπλασιασμό και αποθήκευση στον καταχωρητή $t0 με την τιμή του $t1
  mul $t0,$t1,$t2
- για διαίρεση και αποθήκευση στον καταχωρητή $t0 με την τιμή του $t1
  div $t0,$t1,$t2

Εντολές που σχετίζονται με αλλαγή ροής του προγράμματος (διακλαδώσεις):

- beq $t1,$t2,label  jump to label if $t1=$t2
- blt $t1,$t2,label  jump to label if $t1<$t2
- bgt $t1,$t2,label  jump to label if $t1>$t2
- ble $t1,$t2,label  jump to label if $t1<=$t2
- bge $t1,$t2,label  jump to label if $t1>=$t2
- bne $t1,$t2,label  jump to label if $t1<>$t2

Εντολές που σχετίζονται με την κλήση συναρτήσεων:

- j label  jump to label
- jal label  κλήση συνάρτησης
- jr $ra  άλμα στη διεύθυνση που έχει ο καταχωρητής στο παράδειγμα είναι ο $ra που έχει την διεύθυνση επιστροφής συνάρτησης

Αφού περιγράψαμε το στόχο του σημείου που βρισκόμαστε, δηλαδή την παραγωγή της assembly βάσει των 4αδων ενδιάμεσου κώδικα που έχουν παραχθεί, καθώς και τις βασικές εντολές του MIPS,θα συνεχίσουμε με την επισκόπιση των συναρτήσεων που υλοποιήσαμε για την παραγωγή του τελικού κώδικα. Η βασική μας συνάρτηση που παράγει τον κώδικα assembly είναι η finalCodeGeneration(). Όπως είναι λογικό η μέθοδος αυτη κάνει χρήση άλλων βοηθητικών μεθόδων. Θα ξεκινήσουμε απο την ανάλυση των μεθόδων αυτών και στη συνέχεια θα καταλήξουμε στην finalCodeGeneration().

Η πρώτη βοηθητική συνάρτηση είναι η **gnlvcode()**. Η gnlvcode() μεταφέρει στον $t0 την διεύθυνση μιας *μη τοπικής μεταβλητής*. Από τον πίνακα συμβόλων βρίσκει πόσα επίπεδα επάνω βρίσκεται μια τοπική μεταβλητή (την παίρνει ως όρισμα οταν συναντήσει μια τέτοια) και μέσα από τον σύνδεσμο προσπέλασης την εντοπίζει.

Ψάχνει δηλαδή σε ολα τα entities για κάθε scope. Αφού την εντοπίσει, χρησιμοποιώντας getter της κλάσης entity (σύμφωνα δηλαδή με το όνομά της),

ανακτά το offset της, κρατάει σε μια μεταβλητή το πόσα scopes <<πάνω>> βρέθηκε (έστω i) και γράφει στο αρχείο assembly:

lw $t0,-4($sp)  // στοίβα του γονέα

όσες φορές χρειαστεί: (μέχρι το i)

lw $t0,-4($t0)  // στοίβα του προγόνου που έχει τη μεταβλητή

fileASM.write("\taddi $t0,$t0,-"+str(offset)+"\n") // διεύθυνση της μη τοπικής μεταβλητής

Επίσης υπάρχει μια παραλλαγή της gnlvcode(), η **gnlvcodecp(),** η οποία επιστρέφει ακριβώς αυτό που θα έγραφε η gnlvcode() σε ένα string ώστε να τυπωθεί σε κάποιο άλλο σημείο, όταν χρειαστεί (εξηγούμε πιο κάτω).

Η δεύτερη βοηθητική συνάρτηση που υλοποιήσαμε είναι η **loadvr().** Η συνάρτηση αυτή παίρνει ως όρισμα μια μεταβλητή και έναν αριθμό καταχωρητή που θα χρησιμοποιηθεί και κάνει μεταφορά δεδομένων απο τη μεταβλητή στον καταχωρητή. Η λειτουργία της συνάρτησης αυτής εξαρτάται απο το τι είδος μεταβλητή δόθηκε ως όρισμα. Δηλαδή εαν η μεταβλητή είναι *σταθερά, σφαιρική μεταβλητή, τοπική μεταβλητή, παράμετρος περασμένη με αναφορά ή τιμή ή αντιγραφή,* ή μεταβλητή που ανήκει σε κάποιον πρόγονο , την οποία έχει πάρει (ο πρόγονος) ως όρισμα απο αλλού με τιμή, αντιγραφή ή με αναφορα (στον κώδικά μας αναφέρεται ως prog_cp_parameter για αντιγραφή, prog_cv_parameter για τιμή και prog_ref_parameter για αναφορά). Με σκοπό να διακρίνουμε τις παραπάνω περιπτώσεις, μόλις ξεκινησει η εκτέλεση της loadvr(), καλείται μια άλλη συνάρτηση, η **checkVariable(var)**, η οποία επιστρέφει ένα string που περιέχει το τι είναι η μεταβλητή. Για παράδειγμα εαν η μεταβλητή είναι στοιχείο που έχει πάρει ως όρισμα κάποια πρόγονη συνάρτηση (της τρέχουσας) με αναφορά, τότε τη checkVariable(var) θα επιστρέψει “prog_ref_parameter”.Η checkVariable() κάνει μια απλή προσπέλαση των scopes και των entities στον πίνακα συμβόλων, χρησιμοποιώντας τους αντίστοιχους getters, και αφού βρεί τον τύπο της μεταβλητής κάνει break και τον επιστρέφει.

Ας επιστρέψουμε τώρα στην συνάρτηση loadvr(). Αφού βρεθεί το τι τύπος μεταβλητής είναι το όρισμά της, θα πρέπει να γίνει αναζήτηση στον πίνακα συμβόλων ώστε να βρούμε το offset της. Αν η μεταβλητή είναι σφαιρική, προφανώς θα πρέπει να την αναζητήσουμε στο entity list του scope της main. Έχουμε λοιπόν τις εξής περιπτώσεις:

- αν v είναι σταθερά
  li $tr,v
- αν v είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα
  lw $tr,-offset($s0)

- αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος

φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή
lw $tr,-offset($sp)

- αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο
με το τρέχον
lw $t0,-offset($sp)
lw $tr,($t0)

- αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος
φωλιάσματος μικρότερο από το τρέχον
gnlvcode()
lw $tr,($t0)
- αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος
μικρότερο από το τρέχον
gnlvcode()
lw $t0,($t0)
lw $tr,($t0)

Η τρίτη βοηθητική συνάρτηση που υλοποιήσαμε είναι η **storevr().** Η συνάρτηση αυτή παίρνει ως όρισμα τον αριθμό ενός καταχωρητή και μια μεταβλητή και κάνει μεταφορά δεδομένων από τον καταχωρητή στη μνήμη. Με την ίδια λογική που διακρίναμε περιπτώσεις στην loadvr(), έτσι και εδώ το τι κώδικα θα παράξουμε εξαρτάται, απο το τι τύπο μεταβλητής έχουμε συναντήσει.

- αν v είναι καθολική μεταβλητή – δηλαδή ανήκει στο κυρίως πρόγραμμα

  sw $tr,-offset($s0)

- αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος
φωλιάσματος ίσο με το τρέχον, ή προσωρινή μεταβλητή
sw $tr,-offset($sp)
- αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος ίσο
με το τρέχον
lw $t0,-offset($sp)
sw $tr,($t0)

- αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή και βάθος

φωλιάσματος μικρότερο από το τρέχον
gnlvcode(v)
sw $tr,($t0)

- αν v είναι τυπική παράμετρος που περνάει με αναφορά και βάθος φωλιάσματος
  μικρότερο από το τρέχον
  gnlvcode(v)
  lw $t0,($t0)
  sw $tr,($t0)

Προφανώς όπως και πρίν, βάθη φωλιάσματος και offsets βρίσκονται με προσπελάσεις του πίνακα συμβόλων. Επίσης υπάρχει μια παραλλαγή της storerv(),η **storerp(),** η οποία επιστρέφει ακριβώς αυτό που θα έγραφε η storerv() σε ένα string ώστε να τυπωθεί σε κάποιο άλλο σημείο, όταν χρειαστεί (εξηγούμε πιο κάτω).

Ήρθε η ώρα λοιπόν να περιγράψουμε τη βασική συνάρτηση που παράγει τον τελικό κώδικα και χρησιμοποιεί όλες τις παραπάνω, την **finalCodeGeneration(framelength,startQuad).**Η συνάρτηση αυτή καλείται απο την program() ή την subprogram(), τις συναρτήσεις του συντακτικού αναλυτή, όταν ο λεκτικός αναλυτής (lex()) συναντήσει το αλφαριθμητικό “endprogram” ή “endfunction”.Καλείται δηλαδή όταν φτάσουμε στο τέλος του πηγαίου προγράμματος starlet ή στο τέλος μιας συνάρτησης. Στο σημείο αυτο, στον σφαιρικό πίνακα quadList, έχουμε όλα τα αντικείμενα quad, όλες δηλαδή τις τετράδες που έχουν παραχθεί. Συνεπώς το μόνο που μένει είναι για κάθε τετράδα να παράξουμε τις αντίστοιχες εντολές μηχανής assembly. Η συνάρτηση παίρνει ως πρώτο όρισμα το frame length του μπλοκ που την κάλεσε και την πρώτη τετράδα του μπλοκ. Αυτό που μένει είναι με κατάλληλες προσπελάσεις πεδίων των αντικειμένων που εξετάζουμε (π.χ. με quad.getOp1() παίρνουμε το 2$^{ο}$ στοιχείο της τετράδας), να διακρίνουμε σε τι περίπτωση είμαστε (π.χ. έχουμε ενα jump, _, _, label ή begin_block, name, _, _) και να γράψουμε τις καταλληλες εντολές μηχανής σύμφωνα με τα παρακάτω:

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

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

- op x,y,z

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

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

- retv “_”, “_”, x
  loadvr(x,1)
  lw $t0,-8($sp)
  sw $t1,($t0)
- par,x,CV, _
  loadvr(x,0)
  sw $t0, -(12+4i)($fp)
  όπου i ο αύξων αριθμός της παραμέτρου

- par,x,REF, _
  αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η
  παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος
  που έχει περαστεί με τιμή
  add $t0,$sp,-offset
  sw $t0,-(12+4i)($fp)

- par,x,REF, _
  αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η
  παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με
  αναφορά
  lw $t0,-offset($sp)
  sw $t0,-(12+4i)($fp)

- par,x,REF, _

αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή

gnlvcode(x)

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

- par,x,REF, _

αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά

gnlvcode(x)

lw $t0,($t0)

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

- par,x,RET, _

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

add $t0,$sp,-offset

sw $t0,-8($fp)

- call, _, _, f

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

1. αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο

γονέα

lw $t0,-4($sp)

sw $t0,-4($fp)

2. αν καλούσα και κληθείσα έχουν διαφορετικό βάθος φωλιάσματος, τότε η

καλούσα είναι ο γονέας της κληθείσας

sw $sp,-4($fp)

στη συνέχεια μεταφέρουμε τον δείκτη στοίβας στην κληθείσα

add $sp,$sp,framelength

καλούμε τη συνάρτηση

jal f

και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα

add $sp,$sp,-framelength

<u>ΚΛΗΣΗ ΣΥΝΑΡΤΗΣΗΣ</u>

μέσα στην κληθείσα

- στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος

δραστηριοποίησης την διεύθυνση επιστροφής της την οποία έχει τοποθετήσει

στον $ra η jal

sw $ra,($sp)

- στην τέλος κάθε συνάρτησης κάνουμε το αντίστροφο, παίρνουμε από την πρώτη

θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της

συνάρτησης και την βάζουμε πάλι στον $ra. Μέσω του $ra επιστρέφουμε στην

καλούσα

lw $ra,($sp)

jr $ra

Αρχή Προγράμματος και Κυρίως Πρόγραμμα

- το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται, οπότε στην αρχή

του προγράμματος χρειάζεται ένα άλμα που να οδηγεί στην πρώτη ετικέτα του κυρίως

προγράμματος

j Lmain

- στη συνέχεια πρέπει να κατεβάσουμε τον $sp κατά framelength της main

add $sp,$sp,framelength

- και να σημειώσουμε στον $s0 το εγγράφημα δραστηριοποίησης της main ώστε να

έχουμε εύκολη πρόσβαση στις global μεταβλητές

move $s0,$sp