Γκάνος Στέφανος, Α.Μ. 4043

Μπαλής Κωνσταντίνος, Α.Μ. 4117

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Μεταφραστές, Εαρινό Εξάμηνο 2021

ΑΝΑΦΟΡΑ

ΜΑΤΑΦΡΑΣΤΗ ΓΛΩΣΣΑΣ

C-imple

Πίνακας περιεχομένων

1. Εισαγωγή

Σκοπός αυτής της εργασίας είναι να κατασκευαστεί μεταφραστής που θα δέχεται ως είσοδο ένα πρόγραμμα C-imple και θα δίνει ως έξοδο να δημιουργήσουμε κώδικα για τον επεξεργαστή MIPS. Στην αναφορά θα παρουσιάσουμε τα βήματα που ακολουθήσαμε για να μεταφράσουμε ένα προγράμμα C-imple.

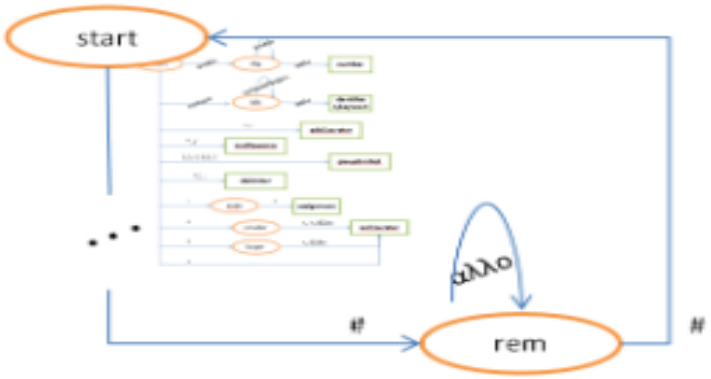
Να σημειωθεί πως φωτογραφίες χρησιμοποιήθηκαν από τις διαφάνειες του μαθήματος ενώ τμήματα κώδικα είναι από το πρόγραμμα που υποβάλλαμε.

2. Λεκτικός Αναλυτής

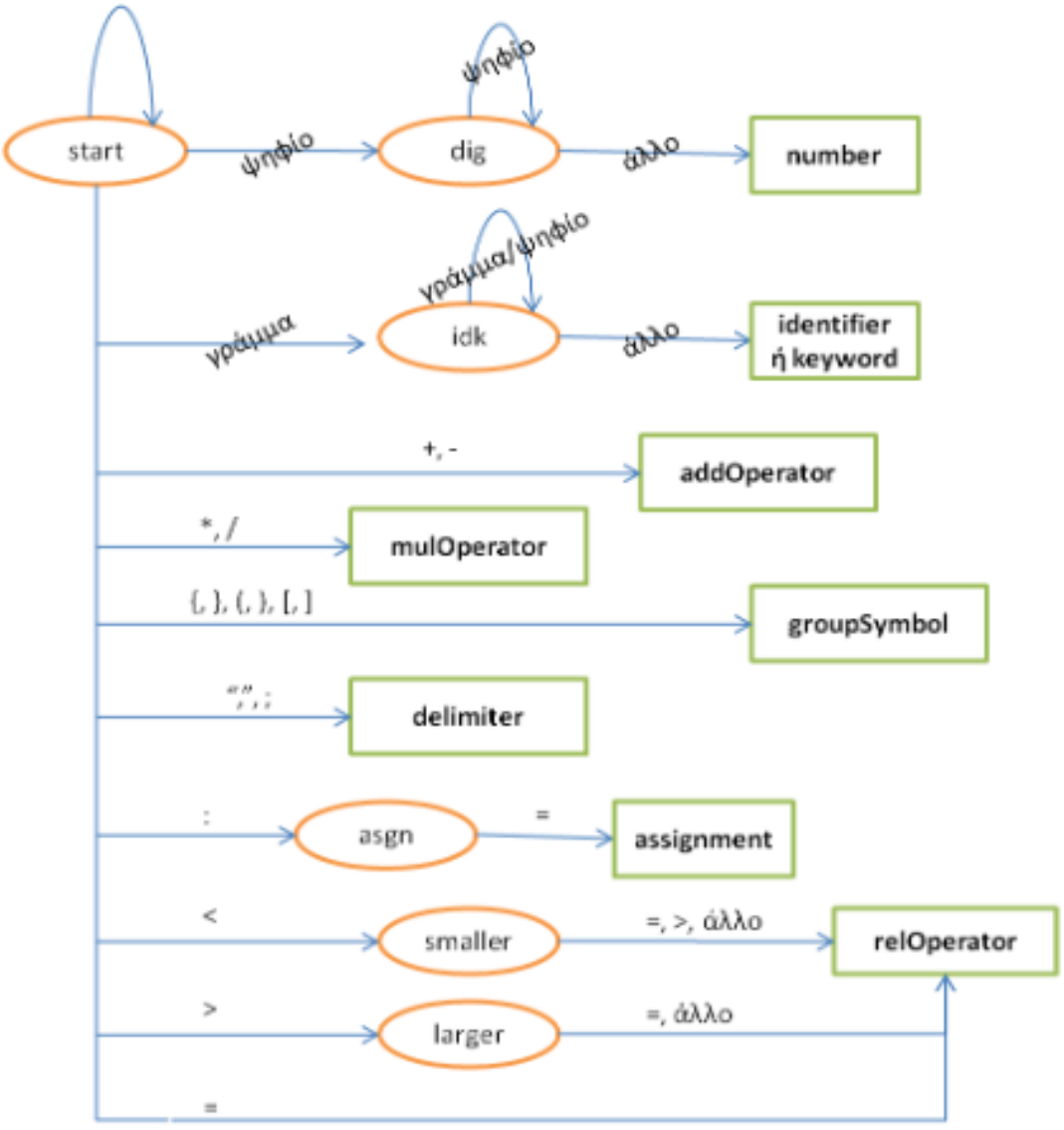
Ο ρόλος του λεκτικού αναλυτή είναι να διαβάζει το πρόγραμμα γράμμα-γράμμα, να υλοποιήσει ένα αυτόματο καταστάσεων και να επιστρέφει λεκτικές μονάδες που θα χρησιμοποιηθούν από το συντακτικό αναλυτή. Αρχικά την υλοποίηση του αυτόματου κατασκευάσαμε ένα πίνακα καταστάσεων που ήταν ένας από τους τρόπους που παρουσιάστηκαν στο μάθημα, διάστασης 7 × 23 όπου κάθε γραμμή συμβολίζει κάθε μία από τις έγκυρες καταστάσεις (start state, idk state, dig state, asgn state, smaller state, larger state και rem state) που μπορεί να βρίσκεται το αυτόματο κάθε στιγμή, ενώ οι 23 στήλες τους χαρακτήρες της γραμματικής της C-imple, συμπεριλαμβανομένων των γραμμάτων του αγγλικού αλφάβητου και των αριθμητικών ψηφίων ενώ η τελευταία στήλη τους χαρακτήρες που δεν ανήκουν στη γραμματική.

Σε αυτό το σημείο να σημειώσουμε πως το κενό (“ ”), το tab (‘‘\t’’) και την αλλαγή γραμμής (“\n”), για την οποία επιπλέον κρατάμε και μια μεταβλητή σχετικά με τη γραμμή που βρισκόμαστε στο πρόγραμμα την οποία αργότερα θα επιστρέψουμε, τα θεωρούμε όλα λευκούς χαρακτήρες οπότε στο πίνακα καταστάσεων αντιστοιχούν όλα στην ίδια στήλη (και συγκεκριμένα στην πρώτη). Ακόμη να σημειώσουμε πως χαρακτήρες που δεν ανήκουν στην γραμματική προφανώς δε μπορούν να χρησιμοποιηθούν στον εκτελέσιμο κώδικα, επιτρέπουμε ωστόσο να χρησιμοποιηθούν στα σχόλια αφού όπως φαίνεται και στην (Εικόνα 2.1) δηλαδή οτιδήποτε και αν δούμε εφόσον είμαστε σε κατάσταση σχόλια παραμένουμε σε αυτή και από τη στιγμή που δε δηλώθηκε κάτι αντίθετο από την εκφώνηση το επιτρέψουμε. Δεδομένης της έγκυρης κατάστασης που βρισκόμαστε και σε συνδυασμό με το χαρακτήρα που διαβάζουμε μεταφερόμαστε στην αντίστοιχη θέση του πίνακα που και κατασκευάστηκε σύμφωνα με το αυτόματο της (Εικόνας 2.2) και θα μας υποδείξει ποια θα είναι η επόμενη κατάσταση. Ακόμη κρατάμε σε μορφή πίνακα κάποιες δεσμευμένες λέξεις (keywords) που έχουν ιδική σημασία

στο πρόγραμμα και να τις ξεχωρίζουμε από άλλες μεταβλητές (identifiers).



Εικόνα 2.1



Εικόνα 2.2

Αυτή η επανάληψη που διαβάζουμε χαρακτήρα συνεχίζεται ως ότου μας υποδείξει ο πίνακας μεταβάσεων ότι έχει αναγνωρίσει μια λεκτική μονάδα και εφόσον αυτή δε ξεπερνά τους 30 χαρακτήρες και αν πρόκειται για αριθμό να μη ξεπερνά το προκαθορισμένο εύρος τιμών που μας έχει δοθεί την επιστρέφουμε. Αντίθετα σε αυτές τις περιπτώσεις όπως και στις περιπτώσεις που αναγνωρίσει κάποιο σφάλμα, δηλαδή το πρόγραμμα εισόδου δε ακολουθεί τους κανόνες της C-imple όπου και θα προκύψει σφάλμα από τον πίνακα μεταβάσεων, τότε εμφανίζουμε αντίστοιχα μηνύματα λάθους και σταματάμε την εκτέλεση του προγράμματος. Τέλος αν όλα έχουν γίνει κανονικά επιστρέφουμε στο συντακτικό αναλυτή μια τριάδα τιμών σε μορφή πίνακα με την πρώτη στήλη να είναι ο τύπος της λεκτικής μονάδας που επιστρέφεται, στη δεύτερη το όνομά της και στη τρίτη ο αριθμός γραμμής για να βοηθήσει το συντακτικό αναλυτή να δίνει ακριβή μηνύματα σε περίπτωση λάθους. Στις παρακάτω εικόνες δίνονται ένα ενδεικτικό παράδειγμα από το τι επιστρέφει ο λεκτικός αναλυτής (Εικόνα 2.3) και ένα ενδεικτικό μήνυμα από λανθασμένη είσοδο (Εικόνα 2.4).

```
keyword             program             1
identifier          test3               1
keyword             declare             2
identifier          x                   2
comma               ,                   2
identifier          y                   2
comma               ,                   2
identifier          Output              2
semicolon           ;                   2
opencurlybracket    {                   3
keyword             input               4
openbracket         (                   4
identifier          x                   4
closebracket        )                   4
semicolon           ;                   4
keyword             while               5
openbracket         (                   5
identifier          x                   5
notequal            <>                  5
number              0                   5
closebracket        )                   5
opencurlybracket    {                   5
identifier          y                   6
assignment          :=                  6
identifier          y                   6
addition            +                   6
identifier          x                   6
semicolon           ;                   6
keyword             print               7
openbracket         (                   7
identifier          x                   7
closebracket        )                   7
semicolon           ;                   7
keyword             input               8
openbracket         (                   8
identifier          x                   8
closebracket        )                   8
semicolon           ;                   8
closecurlybracket   }                   9
semicolon           ;                   9
closecurlybracket   }                   10
dot                 .                   10
```

Εικόνα 2.3

```
keyword             program             1
identifier          test3               1
keyword             declare             2
identifier          x                   2
comma               ,                   2
identifier          y                   2
comma               ,                   2
Error! There is a string in line : (2) that's at least 30 characters long (the m
aximum length).
```

Εικόνα 2.4

3. Συντακτικός Αναλυτής

Ο ρόλος του συντακτικού αναλυτή είναι να αποκτά μια ακολουθία λεκτικών μονάδων από το λεκτικό αναλυτή και να επαληθεύσει ότι η ακολουθία των λεκτικών μονάδων μπορεί να παραχθεί από τη γραμματική της γλώσσας που έχει δοθεί το πρόγραμμα στην είσοδο. Όπως αναφέρθηκε στην στη διάλεξη για κάθε κανόνα δημιουργούμε ένα αντίστοιχο υποπρόγραμμα.

3.1 Υποπρόγραμμα program()

Η γραμματική της C-imple ξεκινά (βλ. Εικόνα 3.1) με την κωδική λέξη program γι' αυτό και ξεκινάμε το συντακτικό αναλυτή καλώντας την πρώτο υποπρόγραμμα program() με όρισμα την πρώτη λεκτική μονάδα του επιστρέφει ο λεκτικός αναλυτής ελέγχοντας αν όντως το όνομά της (δηλαδή η δεύτερη στήλη) είναι program και αντίστοιχα ξανακαλούμε τον λεκτικό αναλυτή για να προσδιορίσουμε το αναγνωριστικό του προγράμματος (ID) και καλούμε το επόμενο υποπρόγραμμα block(). Σε αντίθετη περίπτωση εμφανίζουμε σχετικά μηνύματα λάθους. Τέλος όταν τελειώσουν όλες οι διαδικασίες, που θα αναλυθούν παρακάτω, επιστρέφουμε ξανά στην program() όπου ελέγχουμε αν υπάρχει τελεία ''.'' που σημαίνει ότι φτάσαμε στο τέλος του προγράμματος και αντίστοιχα ενημερώνουμε το χρήστη με μήνυμα λάθους άμα δεν υπάρχει.

```
program         :     program ID block .
```

Εικόνα 3.1

3.2 Υποπρόγραμμα block()

Το υποπρόγραμμα block() δεν επιτελεί κάποια σύνθετη διαδικασία παρά μόνο καλεί τα τρία βασικά blocks (Εικόνα 3.2) του προγράμματος declarations(), subprograms(), statements() και επιστρέφει, κάτι που προκύπτει από τον τρόπο υλοποίησής μας, στο program() τη τελευταία λεκτική μονάδα που όπως ήδη αναφέρθηκε πρέπει να είναι η τελεία.

```
block        :    declarations subprograms statements
```

Εικόνα 3.2

3.3 Υποπρόγραμμα declarations()

Το υποπρόγραμμα είναι υπεύθυνο για να ξεκινήσουν οι δηλώσεις των μεταβλητών και να τελειώσουν με το ‘’;’’. Οι δηλώσεις ξεκινούν με το keyword declare και στη συνέχεια καλείται το υποπρόγραμμα varlist που είναι υπεύθυνο για τις μεταβλητές, ενώ μπορούμε να έχουμε πολλαπλές τέτοιες δηλώσεις μεταβλητών πριν τη δηλώσεις υποπρογραμμάτων (subprograms) ή και καθόλου (Εικόνα 3.3.2).

```
declarations  :    ( declare varlist ; )*
```

Εικόνα 3.3.1

3.4 Υποπρόγραμμα varlist()

Το υποπρόγραμμα είναι υπεύθυνο για τις δηλώσεις των μεταβλητών. Από τον λεκτικό αναλυτή αναμένουμε να επιστρέφει ένα τουλάχιστον αλφαριθμητικό ενώ για περισσότερα πρέπει να είναι διαχωρισμένα μεταξύ τους με ‘’,’’(Εικόνα 3.4).

```
varlist       :    ID  ( , ID )*
              |    ε
```

Εικόνα 3.4

3.5 Υποπρόγραμμα subprograms()

Να ξεκινήσουμε λέγοντας πως η subprograms και η subprogram υλοποιήθηκαν στο ίδιο υποπρόγραμμα . Αφού τελειώσουμε με τις δηλώσεις των μεταβλητών μετά έχουμε τις δηλώσεις της μίας η περισσοτέρων συναρτήσεων ή/και διαδικασιών εφόσον αυτές υπάρχουν και καλούμε τη subprogram() (Εικόνα 3.5.1). Για τις συναρτήσεις και τις διαδικασίες αναμένουμε (Εικόνα 3.5.2) να βρούμε στην αρχή της δήλωσης τους τα keywords ‘function’ και ‘procedure’ ακολουθούμενα από τα ονόματά τους. Στη συνέχεια παρενθέσεις για τα ορίσματα όπου

καλούμε την formalparlist(). Μετά και από αυτή τη δήλωση καλούμε την block() όπου βρίσκεται το σώμα της συνάρτησης.

```
subprograms    :    ( subprogram )*
```

Εικόνα 3.5.1

```
subprogram    :    function ID ( formalparlist ) block
              |    procedure ID ( formalparlist ) block
```

Εικόνα 3.5.2

3.6 Υποπρόγραμμα formalparlist()

Η formalparlist είναι ακόμη μια συνάρτηση απλή που απλά χωρίζει τα ορίσματα των συναρτήσεων και των διαδικασιών. Σημειώνεται πως μπορεί να μην έχει ορίσματα ενώ αν έχει πολλαπλά όπως είπαμε τα χωρίζουμε με κόμμα (Εικόνα 3.7).

```
formalparlist :    formalparitem  ( , formalparitem )*
              |    ε
```

Εικόνα 3.6

3.7 Υποπρόγραμμα formalparitem()

Η formalparitem είναι υπεύθυνη για τα ορίσματα συναρτήσεων και των διαδικασιών. Με in δηλώνονται οι μεταβλητές που περνιούνται με τιμή, ενώ με inout οι μεταβλητές που περνιούνται με αναφορά (Εικόνα 3.7).

```
formalparitem :    in ID
              |    inout ID
```

Εικόνα 3.7

3.7 Υποπρόγραμμα statements()

Όσον αφορά το τρίτο βασικό block τώρα που μπορεί να έχει δύο μορφές. Μπορεί να είναι ένα απλό statement ή πολλά statements που στη συγκεκριμένη περίπτωση πρέπει να περικλείσουμε σε αγκύλες. Και

στις δύο περιπτώσεις καλούμε το υποπρόγραμμα statement() στη δεύτερη περίπτωση επαναληπτικά διαχωρίζοντάς τα με ‘’;’’ (Εικόνα 3.8).

```
statements    :    statement ;
              |    { statement  ( ; statement )* }
```

Εικόνα 3.8

3.9 Υποπρόγραμμα statement()

Όσον αφορά τώρα τις εντολές του κυρίως προγράμματος η C-imple υποστηρίζει 9 τέτοιες εντολές και την επιλογή του κενού statement ώστε να τελειώνει κάποια στιγμή η επαναληπτική κλήση από τη statements(). Αυτό βέβαια αφήνει την δυνατότητα να μην εκτελέσει κανένα statement στη υλοποίησή μας. Μπορεί βέβαια πρόγραμμα χωρίς statement να μην έχει νόημα απλά θεωρήσαμε σωστό να αναφέρουμε πως τέτοιο πρόγραμμα θα περάσει από τον compiler. Για τα 9 υπόλοιπα statements :

- Ανάθεσης : Αν από το λεκτικό δούμε identifier τότε στην επόμενη κλήση αναμένουμε σύμβολο ανάθεσης διαφορετικά εμφανίζουμε μήνυμα λάθους. Άμα δούμε το σύμβολο ανάθεσης καλούμε την expression(), που αναλύεται στη συνέχεια, για να ελέγξουμε την εγκυρότητα της έκφρασης που πρέπει να αναθέσουμε στο identifier (Εικόνα 3.9.1)

```
assignStat    :    ID := expression
```

Εικόνα 3.9.1

- If : Αν ο λεκτικός αναλυτής επιστρέψει “if” τότε αναμένουμε παρένθεση και καλούμε την condition, που αναλύεται στη συνέχεια. Με την ολοκλήρωση της condition αναμένουμε το κλείσιμο της παρένθεσης. Αν όλα ολοκληρωθούν με ομαλό τρόπο τότε καλούμε τις statements που ακολουθούν. Το else δεν αποτελεί υποχρεωτίκο τμήμα της εντολής και τα statements που βρίσκονται μέσα στην else εκτελούνται μόνο εφόσον δεν ισχύει η συνθήκη condition. Σε περίπτωση μη ύπαρξης παρενθέσεων στην

“if” είτε στην αρχή είτε στο τέλος μας εμφανίζεται μήνυμα σφάλματος (Εικόνα 3.9.2).

```
ifStat      :   if ( condition ) statements elsepart

elsepart    :   else statements
            |   ε
```

Εικόνα 3.9.2

- While: Αν ο λεκτικός αναλυτής επιστρέψει “while” τότε αναμένουμε ακριβώς τα ίδια με την “if” που αναλύθηκε πιο πάνω, δηλαδή αναμένουμε παρένθεση και καλούμε την condition. Όσο η συνθήκη condition ισχύει καλούμε την statements. Στο τέλος είναι απαραίτητη η condition να κλείνει με παρένθεση. Στην περίπτωση απουσίας οποιασδήποτε παρένθεσης παρουσιάζεται μήνυμα λάθους (Εικόνα 3.9.3).

```
whileStat   :   while ( condition ) statements
```

Εικόνα 3.9.3

- Switchcase: Η switchcase ελέγχει τις conditions που βρίσκονται μετά τα case. Μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες statements. Είναι απαραίτητη η ύπαρξη παρενθέσεων στην αρχή και στο τέλος. Οποιαδήποτε έλλειψη τους έχει ως αποτέλεσμα την εμφάνιση σφάλματος . Μετά ο έλεγχος μεταβαίνει έξω από την switchcase. Αν κατά το πέρασμα, καμία από τις case δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην default και εκτελούνται οι statements. Αν σε αυτή την περίπτωση δεν εμφανιστεί το default όπως αναμένεται τότε έχουμε μήνυμα λάθους. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την switchcase (Εικόνα 3.9.4).

```
switchcaseStat:   switchcase
                    ( case ( condition ) statements )*
                    default statements
```

Εικόνα 3.9.4

- Forcase: Η forcase ελέγχει τις condition που βρίσκονται μετά τα case. Μόλις βρεθεί κάποια από αυτές αληθής, τότε εκτελούνται οι αντίστοιχα οι statements. Έπειτα ο έλεγχος μεταβαίνει στην αρχή της forcase. Στην περίπτωση που κανένα από τα case δεν ισχύει, τότε μεταβαίνουμε στην default και εκτελούνται τα αντίστοιχα statemens. Στην συνέχεια ο έλεγχος βγαίνει εκτός της forcase. Παρουσία σφαλμάτων στην forcase έχουμε αντίστοιχα όπως και στην switchcase, δηλαδή όταν απουσιάζουν οι παρενθέσεις είτε στην αρχή είτε στο τέλος ή όταν απουσιάζει το default στην περίπτωση που το αναφέραμε (Εικόνα 3.9.5).

```
forcaseStat    :    forcase
                        ( case ( condition ) statements )*
                        default statements
```

Εικόνα 3.9.5

- Incase: Η incase ελέγχει τις condition που βρίσκονται μετά το case,εξετάζοντας τες κατά σειρά. Για κάθε μία για τις οποίες η αντίστοιχη condition ισχύει εκτελούνται οι statemens που την ακολουθούν. Θα εκτελεστούν όλες οι statements των condition που ισχύουν,αφότου πρώτα εξεταστούν με τη σειρά όλες οι condition. Όταν ολοκληρωθεί η εξέταση όλων των case, τότε ο έλεγχος μεταβαίνει έξω από τη δομή incase, σε περίπτωση που καμία από τις statements δεν έχει εκτελεστεί. Σε κάθε άλλη περίπτωση, αν έστω και μία statement εκτελεστεί τότε ο έλεγχος μεταβαίνει στην αρχή της inacase. Η παρουσία σφαλμάτων οφείλεται στις ίδιες περιπτώσεις που αναφέρθηκαν και πιο πάνω στις “switchcase” και “forcase” (Εικόνα 3.9.6).

```
incaseStat    :    incase
                        ( case ( condition ) statements )*
```

Εικόνα 3.9.6

- Call: Η call καλεί μία διαδικασία. Αν υπάρχει identifier και έχουν χρησιμοποιθεί οι παρενθέσεις με τον

αναμενόμενο τρόπο, δηλαδή έχουν ανοίξει και κλείσει στα σημεία που πρέπει, τότε καλούμε την actualparlist. Στην περίπτωση της call σφάλματα μπορούμε να έχουμε για δύο λόγους, είτε για την απουσία κάποιας παρένθεσης στην αρχή ή στο τέλος, είτε για την απουσία του identifier, το οποίο αναμένονταν να δούμε (Εικόνα 3.9.7).

```
callStat      :     call ID( actualparlist )
```

Εικόνα 3.9.7

- Return And Print: Οι δύο διαδικασίες αυτές είναι αρκετά όμοιες γι'αυτό και αποφασίσαμε για να μη επαναλαμβάνουμε ίδια κομμάτια κώδικα να τις χειριστούμε ταυτόχρονα.
  Η return χρησιμοποιείται μέσα σε συναρτήσεις για να επιστραφεί το αποτέλεσμα της συνάρτησης, το οποίο είναι το αποτέλεσμα της αποτίμησης του expression. Η print εμφανίζει στην οθόνη το αποτέλσμα της αποτίμησης του expression. Και στα δύο έχουμε για τους ίδιους λόγους εμφάνιση σφαλμάτων, όταν δηλαδή απουσιάζουν οι παρενθέσεις στην αρχή ή στο τέλος (Εικόνα 3.9.8).

```
returnStat    :    return( expression )
printStat     :    print( expression )
```

Εικόνα 3.9.8

- Input: Η input ζητάει από τον χρήστη να δώσει μία τιμή μέσα από το πληκτρολόγιο. Η τιμή που θα δώσει θα μεταφερθεί στην μεταβλητή ID. Σφάλματα εδώ παρουσιάζονται όταν η δήλωση εισαγωγής δεν έχει κλείσει ή δεν έχει ανοίξει στην αρχή (Εικόνα 3.9.9).

```
inputStat     :    input( ID )
```

Εικόνα 3.9.9

3.10 Υποπρόγραμμα actualparlist()

Η actualpartlist είναι μια συνάρτηση με λίστα πραγματικών παραμέτρων στην οποία με in δηλώνονται οι παράμετροι που περνιούνται με τιμή, ενώ με inout οι παράμετροι που περνιούνται με αναφορά. Στην περίπτωση του inout, δηλαδή όταν οι παράμετροι περνιούνται με αναφορά προκύπτει σφάλμα όταν μετά από το inout δεν ακολουθεί identifier (Εικόνα 3.10). Στην αρχή της parlist έχει υλοποηθεί

και η actualparitem.

```
actualparlist :    actualparitem ( , actualparitem )*
              |    ε
```

Εικόνα 3.10

3.11 Υποπρόγραμμα condition()

Η condition είναι μια συνάρτηση η οποία είναι υπεύθυνη για την διαδική έκφραση. Ο λεκτικός αναλυτής επιστρέφει ένα ή και περισσότερους διαδικούς όρους, ανάμεσα στους οποίους βρίσκεται η λέξη “or” (Εικόνα 3.11).

```
condition     :    boolterm ( or boolterm )*
```

Εικόνα 3.11

3.12 Υποπρόγραμμα boolterm

Η boolterm είναι υπεύθυνη για τους διαδικούς όρους. Ενώ ο λεκτικός αναλυτής επιστρέφει
έναν ή και περισσότερους διαδικούς παράγοντες, οι οποίοι χωρίζονται από την λέξη “and” ανάμεσα (Εικόνα 3.12).

```
boolterm      :    boolfactor ( and boolfactor )*
```

Εικόνα 3.12

3.13 Υποπρόγραμμα boolfactor

Η boolfactor είναι άλλη μία συνάρτηση η οποία είναι υπεύθυνη για την μη ύπαρξη συνθηκών(conditions).Σε διάφορες περιπτώσεις παρουσιάζεται σφάλμα, όταν δεν υπάρχει αγκύλη μετά από την συνθήκη αλλά και στην περίπτωση που δεν υπάρχει κάποιος αναμενόμενος σχεσιακός τελεστής (Εικόνα 3.13).

```
boolfactor    :    not [ condition ]
              |    [ condition ]
              |    expression REL_OP expression
```

Εικόνα 3.13

3.14 Υποπρόγραμμα expression

Η expression είναι υπεύθυνη για την αριθμητική έκφραση των αριθμητικών συμβόλων “+” και “-”. Όπου όταν ένα από τα δύο εμφανίζεται τότε καλείται ο λεκτικός αναλυτής (Εικόνα 3.14).Να

σημειώσουμε πως η optinalSign έχει υλοποιηθεί μέσα στην expression.

```
expression    :    optionalSign term ( ADD_OP term )*
```

Εικόνα 3.14

3.15 Υποπρόγραμμα term

Η term είναι μία συνάρτηση υπεύθυνη για όρους στην αριθμητική έκφραση. Συγκριμένα για τα αριθμητικά σύμβολα “*” και “/” (Εικόνα 3.15).

```
term          :    factor  ( MUL_OP factor )*
```

Εικόνα 3.15

3.16 Υποπρόγραμμα factor

Η factor είναι παράγοντας στην αριθμητκή έκφραση στην οποία έχουμε ακέραιους αριθμούς , αριθμητικές εκφράσεις και αναγνωριστικό. Εμφανίζεται σφάλμα σε δύο περιπτώσεις, όταν δεν υπάρχει παρένθεση “)” μετά από αριθμητικές εκφράσεις και όταν δεν υπάρχει ορθή σύνταξη. Όσον αφορά τους αριθμούς καταφέραμε να φτιάξουμε την τετριμμένη λύση που απλά αναγνωρίζει θετικούς αριθμούς ώστε να χρήση μιας global μεταβλητής να κρατάμε το πρόσημο αν υπάρχει και να το προσθέτουμε στον αριθμό. Κάτι που δε καταφέραμε να κάνουμε και θεωρήσαμε σωστό να αναφέρουμε είναι να βάζει πρόσημο αν υπάρχει σε αριθμητικές παραστάσεις (Εικόνα 3.16).

```
factor        :    INTEGER
              |    ( expression )
              |    ID idtail
```

Εικόνα 3.16

3.17 Υποπρόγραμμα idtail

Η idtail είναι η τελευταία συνάρτηση του συντακτικού αναλυτή και είναι υπεύθυνη για την ύπαρξη των παρενθέσεων όταν υπάρχει λίστα πραγματικών παραμέτρων(είναι δυνατό να μην έχουμε καθόλου παραμέτρους)και για την σωστή λειτουργία της διαδικασίας (Εικόνα 3.17).

```
idtail        :    ( actualparlist )
              |    ε
```

Εικόνα 3.17

4. Ενδιάμεσος Κώδικας

Ο ενδιάμεσος κώδικας αποτελείται από ένα σύνολο τετράδων (ένα τελεστή στην πρώτη θέση της τετράδας και οι υπόλοιπες τρεις είναι τα τελούμενα) με όλα τα τελούμενα να μπορούν να είναι μεταβλητές και τα πρώτα δύο να μπορούν να είναι και αριθμητικές σταθερές, συνοδευόμενα από ένα μοναδικό αριθμό τετράδας. Ο ενδιάμεσος κώδικας έχει 7 βασικές υπορουτίνες :

- **nextquad() :** Ο ρόλος της nextquad() είναι μονάχα να επιστρέφει τον αριθμό τετράδας που πρόκειται να παραχθεί. Η μεταβλητή που κρατά αυτόν τον αριθμό καθώς και τον αυξάνει μετά από κάθε κλήση βρίσκεται στην genquad() όπως θα δούμε στη συνέχεια.
  Η κλήση αυτής της υπορουτίνας γίνεται αφενός μεν στην genquad() στο σημείο που παράγονται οι τετράδες (πεντάδες ουσιαστικά με τον αριθμό τετράδας), σαν δεύτερο όρισμα στις backpatch() που διακρίνουμε δύο περιπτώσεις : (α) είτε για να ‘’φύγουμε’’ από δομές απόφασης (π.χ. if, while, switchcase κ.τ.λ.) αν πληρούμε κάποια συνθήκη είτε (β) να αποθηκεύσουμε τον αριθμό τετράδας που πρέπει να επιστρέψουμε αν δεν συμβαίνει αυτό (π.χ. forcase, incase). Τέλος η nextquad() χρησιμοποιείται στη δημιουργία λίστας τετράδων makelist().

- **genquad() :** Η genquad() είναι ουσιαστικά η υπορουτίνα που δημιουργεί τις τετράδες. Δέχεται τέσσερα ορίσματα που όπως ήδη είπαμε είναι ένας τελεστής και τρία τελούμενα και τα αποθηκεύει σε μορφή πίνακα μαζί με τον αριθμό της τετράδας στη μορφή [quadnum, operation, operator1, operator2, operator3]. Στη συνέχεια όπως είπαμε και στην nextquad() αυξάνει μια global μεταβλητή που κρατά των αριθμό για την επόμενη τετράδα και σε ένα global πίνακα προσθέτει στο τέλος την τετράδα ώστε να

μπορούμε να έχουμε σε κάποιο σημείο αποθηκευμένες όλες τις τετράδες του προγράμματος. Η κλήση της υπορουτίνας γίνεται σε όλα τα σήμεια που αναφέρονται στις διαφάνειες του μαθήματος (Δομές ενδιάμεσου κώδικα και incase-forcase) με όποιους τελεστές δε χρησιμοποιούνται ή δε γνωρίζουμε σε εκείνο το σημείο τι να βάλουμε (θα προστεθεί στη συνέχεια με την backpatch()) σημειώνονται με '_'. Τέλος μια ιδιαιτερότητα η οποία είναι η κλήση της halt που συμβαίνει μόνο όταν τελειώσουν τα statements του κυρίως προγράμματος (begin_block και end_block συναντάμε και όταν καλούμε την block() από τη subprograms()) πετυχαίνεται με τη προσθήκη ενός ακόμη ορίσματος στο τέλος της λεκτικής μονάδας 'True' όταν αυτή προέρχεται από την program(), 'False' διαφορετικά.

- **newtemp() :** Η newtemp() καλείται όπου χρειάζεται να δημιουργήσουμε μεταβλητές της μορφής 'T_x' όπου x ένας αριθμός, για να αποθηκεύσουμε μια προσωρινή τιμή. Η κλήση της γίνεται στις expression(), term(), idtail() για να κρατάμε προσωρινά το αποτέλεσμα και στην incase όπου δημιουργούμε προσωρινή μεταβλητή πριν ελέγξουμε πότε δε πληρείται μια συνθήκη.

- **emptylist() :** Η emptylist() που όπως λέει και το όνομα της δημιουργεί μια κενή λίστα χρησιμοποιείται μόνο σε ένα σημείο του κώδικα στην 'switchcase' όπου γίνεται merge αν ικανοποιείται κάποια συνθήκη αλλιώς από τη default απλά μας πάει στην επόμενη τετράδα.

- **makelist() :** Η makelist() δέχεται σαν όρισμα τον αριθμό της επόμενης τετράδας και δημιουργεί μια μη συμπληρωμένη λίστα που είτε συμπληρώνεται από την backpatch όπως η iflist στη statement, είτε γίνεται merge όπως στην 'switchcase' και την boolterm.

- **merge() :** Η merge() δέχεται ως είσοδο δύο πίνακες και δημιουργεί μια λίστα ετικετών τετράδων. Χρησιμοποιείται για την συμπλήρωση τετράδων στις condition() και boolterm() στην ‘switchcase’.

- **backpatch() :** Η backpatch() δέχεται σαν όρισμα ένα πίνακα και άμα βρει στη λίστα των τετράδων κενή τη τελευταία θέση τότε στη συμπληρώνει με το δεύτερο όρισμα που είναι είτε ο επόμενος αριθμός τετράδας που πρέπει να επισκεφτεί είτε ένας αριθμός τετράδας στον οποίο πρέπει να επιστρέψει. Στο κώδικα η backpatch() βρίσκεται σε πολλαπλά σημεία του κώδικα που υλοποιήθηκαν σύμφωνα με τις διαφάνειες του μαθήματος (Δομές ενδιάμεσου κώδικα και incase-forcase).

5. Αρχεία εξόδου

5.1 Αρχείο καταγραφής Quads

Για το αρχείο καταγραφής τετράδων απλά επισκεπτόμαστε τη δομή που έχουμε κρατήσει τις τετράδες και τις τυπώνουμε σε ένα αρχείο (test.int). Για εποπτικούς λόγους χρησιμοποιήσαμε το .ljust() στα ορίσματα. Ενδεικτικό παράδειγμα στην (Εικόνα 5.1)

```
1       begin_block             test3                   _                       _
2       inp                     x                       _                       _
3       <>                      x                       0                       5
4       jump                    _                       _                       10
5       +                       y                       x                       T_1
6       :=                      T_1                     _                       y
7       out                     x                       _                       _
8       inp                     x                       _                       _
9       jump                    _                       _                       3
10      halt                    _                       _                       _
11      end_block               test3                   _                       _
```

Εικόνα 5.1

5.2 Ισοδύναμο σε C#

Για το ισοδύναμο σε C# βασική προϋπόθεση είναι να μην υπάρχουν συναρτήσεις και διαδικασίες. Για αυτό το λόγο στο υποπρόγραμμα subprograms άμα βρούμε κάποια συνάρτηση ή διαδικασία τότε ενημερώνουμε μια global μεταβλητή που λειτουργεί στο πρόγραμμα σαν flag και άμα βρίσκεται στη μονάδα δε δημιουργούμε το ισοδύναμο σε C#.

Στο αρχείο C# αρχικά εισάγουμε μόνο την κεφαλίδα <stdio.h> που είναι απαραίτητη για να μεταγλωττιστεί και να εκτελεστούν αυτές οι βασικές εντολές που αποτελούν το πρόγραμμα από τη στιγμή που δεν έχουμε πολύπλοκες διαδικασίες στην C-imple. Στη συνέχεια φτιάχνουμε την main() του προγράμματος. Τροποποιούμε λίγο την varlist και τη βάζουμε να αποθηκεύει σε μια δομή πίνακα τους identifiers δηλαδή τα ονόματα των μεταβλητών που δηλώνονται. Αντίστοιχα στη newtemp() αποθηκεύουμε και τις τοπικές μεταβλητές που αποτελούν μέρος του προγράμματος και βάζοντας τον τύπο τους ‘int’ στο αρχείο .c τις τυπώνουμε. Έπειτα για κάθε μια από τις τετράδες τυπώνουμε ‘’L_i’’ όπου i είναι ο αριθμός της τετράδας που θα τυπωθεί και ανανεώνεται +1 σε κάθε επανάληψη. Ανάλογα με το operation της τετράδας βρίσκουμε ποια είναι η αντίστοιχη εντολή σε C# και την τυπώνουμε. Τέλος όπως μας ζητήθηκε βάζουμε και την τετράδα σε σχόλια. Ενδεικτικό παράδειγμα ισοδύναμου σε C# στην (Εικόνα 5.2).

```c
#include <stdio.h>
int main(void){
    int x, y, Output;
    int T_1;

    L_1 :        /* ( begin_block test3 _ _ ) */
    L_2 : scanf("%d", &x);       /* ( inp x _ _ ) */
    L_3 : if (x <> 0) goto L_5;      /* ( <> x 0 5 ) */
    L_4 : goto L_10;          /* ( jump _ _ 10 ) */
    L_5 : T_1 = y + x;        /* ( + y x T_1 ) */
    L_6 : y = T_1;        /* ( := T_1 _ y ) */
    L_7 : printf("x = %d", x);       /* ( out x _ _ ) */
    L_8 : scanf("%d", &x);       /* ( inp x _ _ ) */
    L_9 : goto L_3;       /* ( jump _ _ 3 ) */
    L_10 : {};        /* ( halt _ _ _ ) */
/* ( end_block test3 _ _ ) */
}
```

Εικόνα 5.2

6. Πίνακας Συμβόλων

Ο πίνακας συμβόλων αποτελείται από Scopes, Entities και Arguments. Περιέχει πληροφορίες για τη συνάρτηση σχετικά με την εκτέλεση και τερματισμό αλλά και για τις μεταβλητές που χρησιμοποιούνται.

Σύμφωνα με τις υποδείξεις που έγιναν στη διάλεξη αναφέρθηκε πως καλό θα ήταν να γίνουν με κλάσεις για καλύτερη διαχείριση τους. Οπότε ορίστηκαν οι παρακάτω κλάσεις :

- **Record_Entiny** : Αποθηκεύει οντότητες όπως οι μεταβλητές, οι συναρτήσεις, σταθερές, παραμέτρους και προσωρινές μεταβλητές. Τα κοινά όλων των οντοτήτων είναι ότι έχουν κάποιο όνομα (αρχικοποίηση στο name) και σε ποια υποκλάση(αρχικοποίηση στο type). Από εκεί και πέρα για κάθε οντότητα δημιουργούμε μια υποκλάση για τα διαφορετικά τους χαρακτηριστικά.
    - Για τις **μεταβλητές (Variable)** αποθηκεύουμε ακόμη τον τύπο τους (στο type που στη C-imple είναι μόνο Integers γι’αυτό και το αρχικοποιούμε μέσα στη κλάση) και την

απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης (offset).

- Για τις **συναρτήσεις (που ουσιαστικά αναφερόμαστε στα υποπρογράμματα Subprogram)** αποθηκεύουμε ακόμη τον τύπο την ετικέτα της πρώτης τετράδας στον κώδικα της (startQuad), μια λίστα παραμέτρων (Argument_List) και το μήκος του εγγραφήματος δραστηριοποίησης (framelength).
- Για τις **παραμέτρους (Parameter)** αποθηκεύουμε ακόμη τον τρόπο περάσματος (στο Mode) και την απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης (offset).
- Τέλος για τις **προσωρινές μεταβλητές (TempVar)** ισχύει ό,τι ακριβώς ισχύει και για τις μεταβλητές δηλαδή αποθηκεύουμε τον τύπο τους (στο type που στη C-imple είναι μόνο Integers γι'αυτό και το αρχικοποιούμε μέσα στη κλάση) και την απόσταση από την αρχή του εγγραφήματος δραστηριοποίησης (offset).

- **Record_Scope() :** Αντίστοιχα για το scope καταγράφουμε το όνομα του scope, κρατάμε μια λίστα με τα entities (List_Entity), το βάθος φωλιάσματος (nestinglevel) και το Scope που περικλείεται από το παρών (enclosingScope) [το βρήκαμε από τις διαφάνειες στο ecourse].

- **Record_Argument :** Τέλος για το argument κρατάμε το όνομα (name), τον τρόπο περάσματος (parMode) και τον τύπο που όπως εξηγήσαμε είναι πάντα Integer στην C-imple.

Όσον αφορά τις ενέργειες στον πίνακα συμβόλων. Αρχικά ορίσαμε μια global μεταβλητή Top_Scope (που μας βοηθάει να κρατάμε πληροφορίες για το Scope που βρισκόμαστε).

- Για το **New_Scope()** το δημιουργούμε με το που εισέρθουμε στην block() δηλαδή ξεκινάμε τη μετάφραση της νέας συνάρτησης. Καταγράφουμε το όνομα, την ορίζουμε ως enclosing Scope και της αναθέτουμε βάθος φωλιάσματος +1 από εκεί που βρισκόμαστε, 0 αν είναι το πρώτο που βλέπουμε.
- Το **delete_Scope**() καλείται μετά τη statements() στη block() και διαγράφει το παρών Top_Scope και στη θέση του βάζει το Scope που περικλείονταν.
- Για το **New_Entity()** κάθε φορά που βλέπουμε κάποια μεταβλητή, συνάρτηση, τοπική μεταβλητή ή κάποια παράμετρο δημιουργούμε καινούρια οντότητα περνάμε τις κατάλληλες παραμέτρους για να την ορίσουμε και τη προσθέτουμε στη λίστα με τις οντότητες.
- Για το **New_Argument()** κάθε φορά που βλέπουμε μια μεταβλητή κάνουμε καταγραφή του argument, περνάμε τα ορίσματά της (όνομα και τύπο (CV ή REF)) και περνάμε το νέο argument σα παράμετρο στη τελευταία συνάρτηση στη Top_Scope (δηλαδή την πιο πρόσφατη που είδαμε αφού προσθέτουμε με append τα νέα Entities στη List_Entity).

Για τον υπολογισμό του offset στην **offset()** έχουμε μια τοπική μεταβλητή για να κρατάμε πόσες μεταβλητές, προσωρινές μεταβλητές και παραμέτρους έχουμε δει. Ο υπολογισμός προκύπτει από το ότι κάθε τέτοιο Entity χρειαζόμαστε 4 bytes στη μνήμη συν 12 που είναι δεσμευμένα.

Η **startQuad()** όπως αναφέραμε και στη Record_Entity στις συναρτήσεις είναι η ετικέτα της πρώτης τετράδας του κώδικα της συνάρτησης, οπότε αφού ελέγξουμε ότι δε βρισκόμαστε στη program καλούμε την nextquad() και τη βάζουμε στην startQuad της οντότητας.

Παρόμοια αν δε βρισκόμαστε στη program() τότε είμαστε σε συνάρτηση που έχει μήκος εγγραφήματος δραστηριοποίησης. Άρα αφού ελέγξουμε

ότι όντως προερχόμαστε από function ή procedure καλούμε την **framelength()** που υπολογίζει το offset() και αυτό είναι το μήκος εγγραφήματος δραστηριοποίησής της.

Ακόμη έχουμε τα μετατρέψουμε τις παραμέτρους των συναρτήσεων και των διαδικασιών σε οντότητες. Γι'αυτό δημιουργούμε την **addParameters()** που για κάθε παράμετρο στην argument[] δημιουργεί οντότητα τύπου 'Parameter'.

Τέλος κάθε φορά που πριν διαγράψουμε ένα Scope τυπώνουμε όλα τα Scopes με κάθε entity που έχουμε βρει μέχρι εκείνη τη στιγμή σε ένα αρχείο (SymbolTable.txt). Ακολουθεί στην (Εικόνα 6.1) μια ενδεικτική έξοδος για ένα απλό παράδειγμα.

```
Nesting Level: 0
      Scope: test3
            Entities:
                  x Variable Integer 12
                  y Variable Integer 16
                  Output Variable Integer 20
                  T_1 Temporary Variable Integer 0
```

Εικόνα 6.1

7. Τελικός Κώδικας

Τέλος πρέπει να αντιστοιχήσουμε τις εντλές του ενδιάμεσου κώδικα, σε εντολές για τον επεξεργαστή MIPS.

gnvlcode

Αρχικά όπως μας παρουσιάστηκαν στις διαλέξεις φτιάξαμε την βοηθητική συνάρτηση gnvlcode, η οποία μεταφέρει στο καταχωρητή $t0 την διεύθυνση μιας μη τοπικής μεταβλητής. Επίσης θα χρειαστούμε το επίπεδο στο οποίο είναι φωλιασμένη αυτή η μεταβλητή(nestiglevel), γι' αυτό δημιουργούμε μια βοηθητική συνάρτηση ancestor_stack , που ψάχνει που βρίσκεται η μεταβλητή(στοίβα προγόνων) . Στη συνέχεια τυπώνουμε τις εντολές που δίνονται στις διαφάνειες, σε ένα αρχείο.asm .

```
lw $t0,-4($sp)
όσες φορές χρειαστεί:
        lw $t0,-4($t0)
addi $t0,$t0,-offset
```

εικόνα 7.1

Loadvr

Η loadvr δέχεται δύο τιμές σαν ορίσματα την “v” που είναι μία μεταβλητή και την “r” που είναι ο αριθμός καταχωρητή που πρέπει να μεταφέρει δεδομένα η συνάρτηση . Για κάθε τύπο μεταβλητής τυπώνουμε στο αρχείο.asm τις ακόλουθες εντολές, αντικαθιστώντας τις τιμές v, r με τις τιμές που κλήθηκε η loadvr:

- Αν v είναι σταθερά:

```
li $tr,v
```

- Αν v είναι καθολική μεταβλητή ή καθολική προσωρινή μεταβλητή:

```
lw $tr,-offset($s0)
```

- Αν v είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή, ή προσωρινή μεταβλητή:

```
lw $tr,-offset($sp)
```

- Αν v είναι τυπική παράμετρος που περνάει με αναφορά:

```
lw $t0,-offset($sp)
lw $tr,($t0)
```

- Αν v έχει δηλωθεί σε κάποιο πρόγονο και είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή:

```
gnlvcode()
lw $tr,($t0)
```

- Αν v έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τυπική παράμετρος που περνάει με αναφορά:

```
gnlvcode()
lw $t0,($t0)
lw $tr,($t0)
```

Storerv

Η storerv επίσης δέχεται δύο τιμές σαν ορίσματα την “v” που είναι μία μεταβλητή και την “r” που είναι ο αριθμός καταχωρητή που πρέπει να μεταφέρει δεδομένα από τον καταχωρητή στη μνήμη . Για κάθε τύπο μεταβλητής τυπώνουμε στο αρχείο.asm τις ακόλουθες εντολές, αντικαθιστώντας τις τιμές v, r με τις τιμές που κλήθηκε η storerv:,

- Αν v είναι καθολική μεταβλητή:

```
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)
```

Τώρα θα εξηγήσουμε την βασική συνάρτηση του τελικού κώδικα την codegenerator, η οποία έχει ως σκοπό να αντικαταστήσει τις τετράδες σε εντολές MIPS. Η codegenerator καλείται στην υποσυνάρτηση block() του συντακτικού αναλυτή αμέσως μετά τη δήλωση των statements, ώστε να παράγει τελικό κώδικα για κάθε συνάρτηση, διαδικασία και για την program() του προγράμματος C-imple. Για κάθε λεκτική μονάδα ακολουθήσαμε τις εντολές που δίνονται στο pdf (παραγωγή τελικού κώδικα). Μερικές εντολές που πρέπει να σχολιάσουμε είναι:

- Πρέπει να γίνει διάκριση της begin_block της program() και σε όλες τις υπόλοιπες begin_block. Γι' αυτό το πρώτο που κάνουμε είναι να αναζητήσουμε την begin_block με βάθος φωλιάσματος ίσο με μηδέν και κάνουμε JUMP στην πρώτη ετικέτα του κυρίως προγράμματος.

```
j Lmain

addi $sp,$sp,framelength

move $s0,$sp
```

Ενώ για τις υπόλοιπες εκτελούμε:

```
sw $ra,($sp)
```

- Επίσης η halt και η end_block της program() (με βάθος φωλιάσματος δηλαδή ίσο με μηδέν)δεν εκτελούμε καμία εντολή και κάνουμε “pass”.

- Τέλος δεν καταφέραμε να υλοποιήσουμε τελικό κώδικα για πέρασμα παραμέτρων και γι' αυτό τον λογό κάνουμε “pass” όταν βλέπουμε στις τετράδες “par”,”CV”,”REF”, και ”call”. Αυτό έχει ως αποτέλεσμα το αρχείο.asm να μη λειτουργεί σωστά όταν υπάρχουν κλήσεις συναρτήσεων και διαδικιών(procedure). Ένα δικό μας λάθος δε το κάνει φανερό αυτό εξαρχής , καθώς ξεχάσαμε και δεν βγάλαμε από σχόλια ένα “print” που θα ενημέρωνε τον χρήστη ότι το αρχείο.asm δεν λειτουργεί σωστά. Σκεφτήκαμε σε αυτή τη περίπτωση να το διαγράψουμε, όμως για την διαγραφή χρειαζόταν “path”, το οποίο δεν μπορούμε να το γνωρίζουμε για κάθε εκτέλεση.