Εργασία Μεταφραστές

Κουφόπουλος Μύρων 4398

Τριανταφυλλόπουλος Ανδρέας 4504

Σε αυτό το project δημιουργούμε έναν μεταφραστή για την γλώσσα C-imple. Η υλοποίηση χωρίστηκε σε 5 στάδια για να είναι σωστά δομημένος και να ελέγχεται πως επιτελεί όλες τις λειτουργίες του σωστά ο μεταφραστής. Επίσης με αυτό το τρόπο ο κώδικας είναι πιο εύκολα διατηρήσιμος , αναγνώσιμος και επεκτάσιμος ώστε να μπορεί να υποστηρίξει στο μέλλον διαφορετικές γλώσσες προς μετάφραση πέραν της C-imple που χρησιμοποιούμε.

1) Λεκτικός Αναλυτής

2) Συντακτικός Αναλυτής

3) Ενδιάμεσος Κώδικας

4) Πίνακας Συμβόλων

5) Τελικός Κώδικας

Λεκτικός Αναλυτής

Η λεκτική ανάλυση αποτελεί την πρώτη φάση της μεταγλώττισης. Σε αυτή τη φάση το πρόγραμμα προς μετάφραση θα διαβάζεται χαρακτήρα-χαρακτήρα από τον λεκτικό αναλυτή με σκοπό να παραχθούν λεκτικές μονάδες. Οι λεκτικές μονάδες είναι αναγνωριστικά με βάση τα οποία κατατάσσονται οι ομάδες χαρακτήρων που διαβάζονται ώστε να χρησιμοποιηθούν αργότερα από το συντακτικό αναλυτή. Σε περίπτωση που μια ομάδα χαρακτήρων δεν ταιριάζει σε καμία από τις λεκτικές μονάδες θα παραχθεί το αντίστοιχο μήνυμα σφάλματος. Όταν ο λεκτικός αναλυτής αναγνωρίσει μια λεκτική μονάδα, τότε την επιστρέφει στον συντακτικό αναλυτή.

Τα χαρακτηριστικά μίας λεκτικής μονάδας είναι η συμβολοσειρά την οποία αναγνώρισε (recognized_string), η κατηγορία στην οποία ανήκει (family) και ο αριθμός γραμμής στην οποία αναγνωρίστηκε

(line_number). Για την καλύτερη διαχείριση των λεκτικών μονάδων δημιουργούμε μια κλάση Token που περιέχει αυτά τα 3 πεδία. Οι κατηγορίες στις οποίες μπορεί να ανήκει μια λεκτική μονάδα είναι:

identifier: ανήκουν τα ονόματα των μεταβλητών, των συναρτήσεων και ότι άλλο μπορεί να έχει ονομασία σε ένα πρόγραμμα. Θα αναφερόμαστε σε αυτά ως 'αναγνωριστικά'

number: ανήκουν οι αριθμοί

keyword: ανήκουν οι δεσμευμένες λέξεις της εκάστοτε γλώσσας που μεταφράζεται

addOperator: ανήκουν οι προσθετικοί αριθμητικοί τελεστές: (+,-)

mulOperator: ανήκουν οι πολλαπλασιαστικοί αριθμητικοί τελεστές: (*,/)

relOperator: ανήκουν οι λογικοί τελεστές: (==,>=,<=,<,>,<>)

assignment: ο τελεστής εκχώρησης: (:=)

delimeter: σύμβολα διαχωριστών: (, . ;)

groupSymbol: σύμβολα ομαδοποίησης: (()),{},[])

Για να μπορέσουμε να αναγνωρίζουμε εύκολα και γρήγορα σε ποια από τις παραπάνω κατηγορίες ανήκουν οι λεκτικές μονάδες, έχουμε αναθέσει έναν ακέραιο αριθμό 1-9. Για να γίνει κατάταξη ενός συνόλου χαρακτήρων στις κατηγορίες που αναφέρθηκαν παραπάνω, έχει σχεδιαστεί ένα αυτόματο που αντικατοπτρίζει τις λειτουργίες του λεκτικού μας αναλυτή. Οι τελικές καταστάσεις είναι οι 9 κατηγορίες που αναφέραμε παραπάνω. Στο αυτόματο αυτό έχουμε μια αρχική κατάσταση (start), όπου εκεί αναγνωρίζουμε το πρώτο χαρακτήρα κάθε λεκτικής μονάδας και αναλόγως περνάμε είτε σε κάποια ενδιάμεση κατάσταση είτε σε κάποια τελική κατάσταση. Αρχικοποιούμε με ακεραίους τα ονόματα της αρχικής κατάστασης και των ενδιάμεσων καταστάσεων ώστε όσο ο λεκτικός αναλυτής βρίσκεται σε ενδιάμεση κατάσταση να συνεχίζει να διαβάζει χαρακτήρες μέχρι να περάσει σε κάποια τελική κατάσταση.

Αν στη κατάσταση αυτή αν αναγνωρίσουμε κάποιο λευκό χαρακτήρα (π.χ. κενό, αλλαγή γραμμής, tab) τότε παραμένουμε στην ίδια κατάσταση καθώς θέλουμε να αγνοήσουμε αυτούς τους χαρακτήρες (δεν θέλουμε να δημιουργηθεί κάποια λεκτική μονάδα) και ταυτόχρονα

θέλουμε ο επόμενος χαρακτήρας που θα διαβάσουμε να βρίσκεται στην αρχική μας κατάσταση. Λευκός χαρακτήρας είναι και η αλλαγή γραμμής η οποία όταν αναγνωρίζεται θα πρέπει να αυξάνουμε τη μεταβλητή την οποία κρατάει σε ποια γραμμή βρισκόμαστε στο πρόγραμμα.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι ψηφίο τότε περνάμε στην ενδιάμεση κατάσταση dig και όσο συνεχίζουμε να διαβάζουμε ψηφία παραμένουμε σε αυτή. Όταν διαβάσουμε οποιοδήποτε άλλο σύμβολο εκτός από γράμμα τότε περνάμε στην τελική κατάσταση number και έχουμε δημιουργήσει μια λεκτική μονάδα number.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι γράμμα τότε περνάμε στην ενδιάμεση κατάσταση idk και όσο διαβάζουμε γράμματα ή αριθμούς παραμένουμε στην ίδια κατάσταση. Αυτό συμβαίνει διότι αναμένουμε να ολοκληρωθεί το όνομα της μεταβλητής ή μιας συνάρτησης ή μιας δεσμευμένης λέξης. Οποιοδήποτε άλλο χαρακτήρα διαβάσουμε μετά, περνάμε στην τελική κατάσταση identifier ή keyword. Για να επιλέξουμε σε ποια από τις δύο τελικές καταστάσεις ανήκει δημιουργούμε μια λίστα η οποία περιέχει όλες τις δεσμευμένες λέξεις της C-imple και εξετάζουμε αν η λεκτική μονάδα περιέχεται σε αυτή.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο + ή – τότε θα περάσουμε κατευθείαν στη τελική κατάσταση addOperator.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο * ή / τότε θα περάσουμε κατευθείαν στη τελική κατάσταση muldOperator.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο { ή } ή (ή) ή [ή] τότε θα περάσουμε κατευθείαν στη τελική κατάσταση groypSymbol.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο , ή ; ή . τότε θα περάσουμε κατευθείαν στη τελική κατάσταση delimeter.

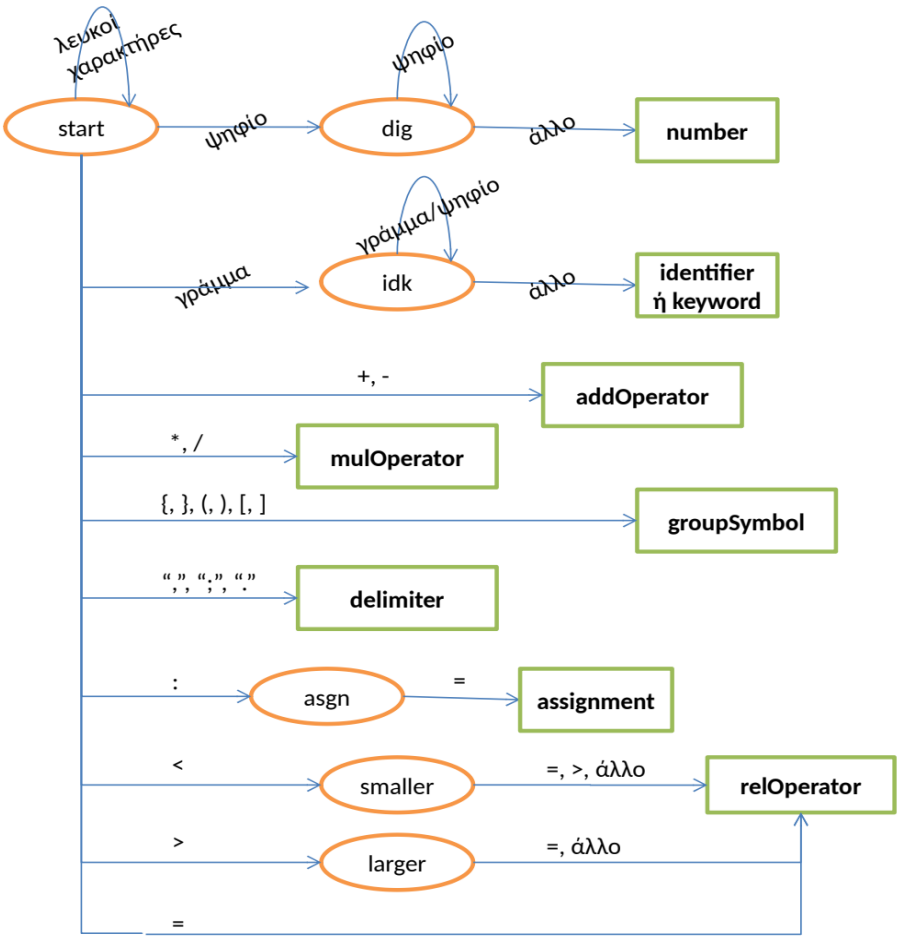
Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο : τότε θα περάσουμε στην ενδιάμεση κατάσταση asgn και θα πρέπει το ακριβώς επόμενο σύμβολο να είναι το = για να περάσουμε στην τελική κατάσταση assignment. Αν μετά το σύμβολο : διαβάσουμε οποιοδήποτε άλλο χαρακτήρα τότε έχουμε error.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο < τότε θα περάσουμε στην ενδιάμεση κατάσταση smaller. Αν ο επόμενος χαρακτήρας είναι = ή > τότε περνάμε στην τελική κατάσταση relOperator και δημιουργούμε την λεκτική μονάδα <= ή <>. Αν διαβάσουμε οποιοδήποτε άλλο χαρακτήρα τότε περνάμε στην τελική κατάσταση relOperator και δημιουργούμε την λεκτική μονάδα <.

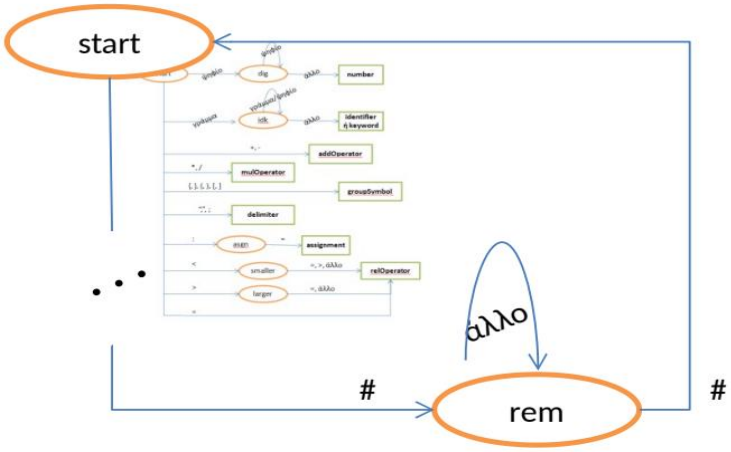
Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο > τότε θα περάσουμε στην ενδιάμεση κατάσταση larger. Αν ο επόμενος χαρακτήρας είναι = τότε περνάμε στην τελική κατάσταση relOperator και δημιουργούμε την λεκτική μονάδα >=. Αν διαβάσουμε οποιοδήποτε άλλο χαρακτήρα τότε περνάμε στην τελική κατάσταση relOperator και δημιουργούμε την λεκτική μονάδα >.

Αν τελικά από την ανάγνωση ενός συνόλου χαρακτήρων έχει αποφασίσει το αυτόματο ότι έχουμε αναγνωριστικό (μεταβλητή ή όνομα συνάρτησης) ή νούμερο ή δεσμευμένη λέξη ή το σύμβολο < ή το σύμβολο > τότε θα πρέπει να γυρίσουμε πίσω κατά ένα χαρακτήρα και να συνεχίσουμε από εκεί την ανάγνωση. Αυτό συμβαίνει διότι όταν βρισκόμαστε στις ενδιάμεσες καταστάσεις των παραπάνω, περνάμε στις τελικές τους καταστάσεις διαβάζοντας τον επόμενο χαρακτήρα (στο αυτόματο στοίβας : ‘άλλο’). Αυτός ο χαρακτήρας δεν αναγνωρίζεται από το αυτόματο οπότε είναι απαραίτητο να γυρίσουμε μία θέση πίσω και να ξαναδιαβάσουμε αυτό το χαρακτήρα.

Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο = τότε θα περάσουμε κατευθείαν στη τελική κατάσταση relOperator.



Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι το σύμβολο # τότε περνάμε στην ενδιάμεση κατάσταση rem όσο βλέπουμε οποιοδήποτε χαρακτήρα παραμένουμε σε αυτή τη κατάσταση, εκτός αν δούμε EOF που τότε έχει τερματίσει το πρόγραμμα και έχουμε error. Όταν ξαναδούμε το σύμβολο # τότε περνάμε πάλι στην αρχική κατάσταση start καθώς ότι βρίσκεται ενδιάμεσα στα σύμβολα αυτά αποτελεί σχόλια και δεν θέλουμε να περάσουν στον συντακτικό αναλυτή άρα δεν θέλουμε να μπουν σε κάποια τελική κατάσταση. Επίσης όσο βρισκόμαστε εντός των σχολίων αν βρεθεί κάποια αλλαγή γραμμής ενημερώνουμε το current_line. Με αυτό το τρόπο ο λεκτικός αναλυτής θα συνεχίσει να διαβάζει τον επόμενο χαρακτήρα στη σωστή γραμμή.

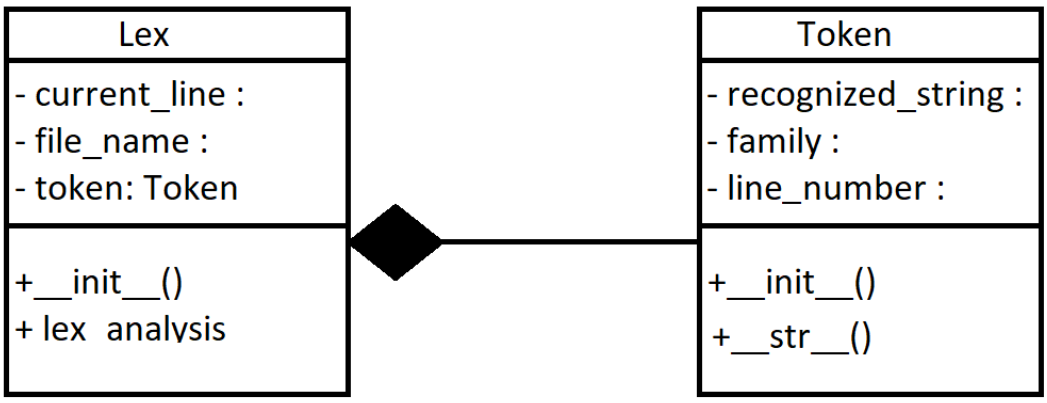


Αν στη κατάσταση start ο επόμενος χαρακτήρας είναι σύμβολο το οποίο δεν σχολιάσαμε παραπάνω τότε το σύμβολο αυτό δεν ανήκει στη γλώσσα C-imple και ο λεκτικός αναλυτής θα το αναγνωρίσει σαν error.

Δύο ακόμα περιορισμοί που χρειάστηκε να ελέγξουμε στο λεκτικό αναλυτή είναι ότι το μήκος του identifier πως δεν είναι μεγαλύτερο από 30 χαρακτήρες και πως κάθε ακέραια αριθμητική σταθερά βρίσκεται εντός του επιτρεπτού εύρους τιμών (-(2^32-1) –(2^32-1)).

Για να παίρνουμε την επόμενη κατάσταση του αυτόματου, δημιουργήσαμε έναν δισδιάστατο πίνακα. Η μία συντεταγμένη είναι η κατάσταση στην οποία βρισκόμαστε τώρα και η άλλη συντεταγμένη είναι ο χαρακτήρας που μόλις διαβάσαμε.

Παρακάτω παρατίθεται το διάγραμμα κλάσεων για τις κλάσεις Token που δημιουργεί αντικείμενα που θα αναπαριστούν λεκτικές μονάδες και Lex που θα δημιουργεί το αντικείμενο του λεκτικού αναλυτή.



Συντακτικός Αναλυτής

Η συντακτική ανάλυση αποτελεί την δεύτερη φάση της μεταγλώττισης. Σε αυτή τη φάση παίρνουμε σαν είσοδο τις λεκτικές μονάδες που παράγει ο λεκτικός αναλυτής. Σαν έξοδο έχουμε ένα συντακτικό δέντρο το οποίο αποτελεί το 'καλούπι' για την επόμενη φάση του μεταφραστή ενώ παράλληλα γίνεται ο έλεγχος για πιθανά συντακτικά σφάλματα από το χρήστη. Εκμεταλλευόμαστε τη γλώσσα C-imple διότι είναι μια γραμματική χωρίς συμφραζόμενα και κατάλληλη για να υλοποιήσουμε την τεχνική της προβλεπούσας αναδρομικής κατάβασης. Αυτό σημαίνει πως όταν υπάρχουν εναλλακτικές, η απόφαση για το ποιος κανόνας θα ενεργοποιηθεί κατά την συντακτική ανάλυση, λαμβάνεται με βάση την επόμενη διαθέσιμη λεκτική μονάδα της εισόδου. Για την υλοποίηση της αναδρομικής κατάβασης ακολουθούμε τον παρακάτω αλγόριθμο.

- Για κάθε κανόνα της γραμματικής υλοποιούμε και μία συνάρτηση η οποία παίρνει το ίδιο όνομα με τον κανόνα. Αυτό γίνεται για διευκόλυνση του προγραμματιστή στην αποσφαλμάτωση και για την εύκολη κατανόηση του κώδικα.

- Για κάθε κανόνα της γλώσσας συμπληρώνουμε το δεξί μέλος.

α) Πρώτη περίπτωση είναι όταν έχουμε μη τερματικά σύμβολα όπου καλούμε τις αντίστοιχες συναρτήσεις. Με αυτό το τρόπο η απόφαση για την ορθότητα θα ανατεθεί στον κατάλληλο κανόνα του προγράμματος. Όταν αυτή αναγνωριστεί, το πρόγραμμα μέσω των αναδρομών(return) θα συνεχίζει την ανάγνωση από τον κανόνα στον οποίον βρισκόταν χωρίς να επηρεαστεί.

β) Δεύτερη περίπτωση είναι όταν συναντάμε τερματικό σύμβολο όπου ελέγχουμε αν συμπίπτει ή όχι με την επόμενη λεκτική μονάδα. Αν το τερματικό σύμβολο της γραμματικής δεν είναι ίδιο με την επόμενη λεκτική μονάδα της εισόδου και στις επιλογές του κανόνα υπάρχει το κενό συνεχίζεται η ροή του προγράμματος διότι μπορεί να αναγνωριστεί από επόμενο κανόνα. Διαφορετικά τυπώνεται το σφάλμα μαζί με την γραμμή που εντοπίστηκε, πληροφορώντας τον χρήστη ότι το τερματικό σύμβολο δεν ανήκει στον συγκεκριμένο κανόνα και τερματίζεται η μετάφραση. Αν το τερματικό σύμβολο της γραμματικής συμπίπτει με την επόμενη λεκτική μονάδα της εισόδου, τότε καταναλώνουμε την λεκτική μονάδα, δηλαδή την αναγνωρίζουμε και διαβάζουμε την επόμενη.

Αρχικά για την υλοποίηση του συντακτικού αναλυτή δημιουργούμε μια ξεχωριστή κλάση(Parser) η οποία θα σχετίζεται με τον λεκτικό αναλυτή και με την κλάση Token. Θέλουμε κάθε κλήση του λεκτικού αναλυτή να επιστρέφει στον συντακτικό αναλυτή ένα αντικείμενο της κλάσης Token. Δημιουργούμε μία μέθοδο get_token() μέσα στη κλάση του συντακτικού αναλυτή η οποία καλεί τον λεκτικό αναλυτή και θα καλεί την επόμενη λεκτική μονάδα. Η συνάρτηση αυτή θα καλείται πριν την εκκίνηση του αρχικού κανόνα του συντακτικού αναλυτή για να έχουμε διαθέσιμη την πρώτη λεκτική μονάδα του προγράμματος. Όταν αναγνωρίσει ο συντακτικός αναλυτής μια λεκτική μονάδα τότε αυτή καταναλώνεται, δηλαδή καλούμε την συνάρτηση get_token() για να μας δώσει την επόμενη. Η εκάστοτε λεκτική μονάδα που διαβάζεται θέλουμε να είναι φανερή σε όλη την κλάση. Η πρώτη μέθοδος που θα καλείται από την κλάση Parser θα είναι η syntax_analyzer() ώστε να παίρνουμε την πρώτη διαθέσιμη λεκτική μονάδα και έπειτα να ξεκινάει ο αλγόριθμος με την συνάρτηση program(). Αν ο συντακτικός αναλυτής αναδρομικά ξαναφτάσει στην κλήση της συνάρτησης program() τότε ο συντακτική ανάλυση θα έχει ολοκληρωθεί επιτυχώς.

Η program() είναι η συνάρτηση που θα ελέγξει αν η πρώτη λεκτική μονάδα(token.recognized_string) στο πρόγραμμα μας είναι η δεσμευμένη λέξη ‘program’, αν ισχύει τότε την καταναλώνουμε και περνάμε στον επόμενο έλεγχο. Στη συνέχεια αναμένουμε να υπάρχει μια μεταβλητή(ID) η οποία θα αποτελεί το όνομα του προγράμματος, την καταναλώνουμε, και έπειτα καλείται η μέθοδος block(). Αν δεν έχουμε κάποιο συντακτικό λάθος και επιστρέψουμε αναδρομικά, τότε ελέγχουμε πως η επόμενη λεκτική μονάδα είναι η ‘.’ Και την καταναλώνουμε. Στο τέλος του προγράμματος περιμένουμε να υπάρχει το EOF και το καταναλώνουμε. Για κάθε έναν από αυτούς τους ελέγχους αν δεν έχουμε την αναμενόμενη λεκτική μονάδα τυπώνουμε το αντίστοιχο μήνυμα σφάλματος και την γραμμή στην οποία βρέθηκε.

```
program        →   program ID
                   block
                   .
```

Αφού έχουμε αναγνωρίσει το όνομα προγράμματος, περνάμε έχουμε το μη τερματικό σύμβολο block οπότε καλείται η μέθοδος block(). Σε αυτή τη μέθοδο η λεκτική μονάδα που θα δει αρχικά είναι η αριστερή αγκύλη ‘{’η οποία σηματοδοτεί πως ό,τι κώδικα διαβάσουμε βρίσκεται στη

main, εφόσον την αναγνωρίσουμε την καταναλώνουμε. Στο σημείο αυτό να σημειώσουμε πως ο κανόνας block εξετάζεται και από άλλους κανόνες για να δηλώσει την εμβέλεια που θα έχει μία συνάρτηση μέσω των αγκυλών της. Στη συνέχεια, ακολουθεί το μη τερματικό σύμβολο declarations για να δηλωθούν οι μεταβλητές της συνάρτησης. Έπειτα αναγνωρίζουμε τις εσωτερικές συναρτήσεις(παιδία) που μπορεί να έχει και το τελευταίο μη τερματικό σύμβολο που θα ακολουθήσουμε είναι το blockstatements για να διαβάσουμε το κώδικα της συνάρτησης. Για να ολοκληρωθεί σωστά ο κανόνας block σαν τελευταία λεκτική μονάδα αναμένουμε την δεξιά αγκύλη '}' που θα υποδείξει το τέλος της εμβέλειας της συνάρτησης, εφόσον την αναγνωρίσουμε την καταναλώνουμε.

```
block        →   {
                     declarations
                     subprograms
                     blockstatements
                 }
```

Στη αρχή κάθε συνάρτησης θα υπάρχουν οι δηλώσεις των μεταβλητών. Σαν πρώτη λεκτική μονάδα περιμένουμε την δεσμευμένη λέξη declare, αν την αναγνωρίσουμε την καταναλώνουμε. Στην συνέχεια για την δήλωση των μεταβλητών αυτών καλείται η μέθοδος varlist(). Μετά την επιστροφή από την μέθοδο αυτή είναι απαραίτητο να αναγνωρίσουμε το ';' το οποίο θα σημάνει το τέλος της δήλωσης, εφόσον το αναγνωρίσουμε το καταναλώνουμε. Οι δηλώσεις των μεταβλητών μπορεί να βρίσκονται και σαν ξεχωριστές αρχικοποιήσεις δηλαδή σε διαφορετικές γραμμές οπότε αν μετά το σύμβολο ';' έχουμε πάλι την λεκτική μονάδα declare, τότε συνεχίζουμε τα έχουμε δηλώσεις. Ωστόσο, είναι πιθανό να μην αναγνωρίσουμε σαν πρώτη λεκτική μονάδα το declare, αυτό σημαίνει πως δεν θα έχουμε δηλώσεις νέων μεταβλητών και πως δεν θα πρέπει να καταναλώσουμε την λεκτική μονάδα αυτή αλλά να επιστρέψουμε στην συνάρτηση που κάλεσε τη μέθοδο declarations().

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

Στο σημείο αυτό, ο κανόνας αναγνωρίζει τις μεταβλητές που έχουν δηλωθεί και οι οποίες αν είναι περισσότερες από μια, θα πρέπει να είναι χωρισμένες με την λεκτική μονάδα ','. Κάθε φορά που συναντάμε είτε το όνομα της μεταβλητής είτε το σύμβολο ',' θα την

καταναλώνουμε. Όμως, μπορεί να μην έχουμε καμία δήλωση μεταβλητής τότε δεν θα πρέπει να καταναλώσουμε, ούτε να βγάλουμε κάποιο σφάλμα στο χρήστη αλλά να επιστρέψουμε στην μέθοδο που κάλεσε το varlist.

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

Ο κανόνας subprograms που αναγνωρίζει τις συναρτήσεις παιδία μπορεί να έχει από καμία μέχρι άπειρες συναρτήσεις ορισμένες εσωτερικά. Κάθε μια που θα βρίσκουμε θα το μη τερματικό σύμβολο subprogram χωρίς να καταναλώνει αφού δεν έχει κάποιο συγκεκριμένο σύμβολο να ανγνωρίσει.

```
subprograms   →    ( subprogram )*
```

Στον κανόνα subprogram για να έχουμε περάσει αυτό σημαίνει πως ήδη η subprograms που την κάλεσε έχει αναγνωρίσει πως υπάρχει συνάρτηση άρα έχει αναγνωρίσει μία από τις δύο δεσμευμένες λέξεις ‘function’ ή ‘procedure’ οπότε καταναλώνουμε. Στις δύο αυτές περιπτώσεις ακολουθούμε ίδια βήματα, δηλαδή περιμένουμε το όνομα της συνάρτησης και καταναλώνουμε, έπειτα την λεκτική μονάδα ‘(’ για να δηλωθούν οι παράμετροι της συνάρτησης, μετά καλούμε τη μέθοδο που διαβάζει τις παραμέτρους και τέλος το κλείσιμο των παραμέτρων με το σύμβολο ‘)’. Σε όλες αυτές τις αναγνωρίσεις καταναλώνουμε. Στη συνέχεια μετά τις παραμέτρους καλούμε την μέθοδο block() για να διαβάσει εσωτερικά της συνάρτησής μας χωρίς να καταναλώσουμε.

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

Σε αυτό το κανόνα περιμένουμε να έχουμε είτε καμία παράμετρο στη συνάρτηση, είτε να έχουμε μία παράμετρο, είτε να έχουμε πάνω από μία παραμέτρους και να χωρίζονται μεταξύ τους με το σύμβολο ‘,’. Σημειώνουμε πως μόνο όταν αναγνωρίζουμε το ‘,’ καταναλώνουμε καθώς στις άλλες περιπτώσεις είτε καλούμε τη μέθοδο formalparitem() είτε επιστρέφουμε στην από την οποία καλέστηκε η formalparlist().

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

Στο σημείο αυτό περιμένουμε να αναγνωρίσουμε μία από τις δύο δεσμευμένες λέξεις ‘in’ και ‘inout’ για να δούμε αν η παράμετρος έχει περάσει με αναφορά ή με τιμή και έπειτα καταναλώνουμε. Στην συνέχεια διαβάζουμε το όνομα της παραμέτρου.

```
formalparitem →    in ID
               |   inout ID
```

Ο κανόνας αυτός διαβάζει μία-μία τις εντολές του κορμού της συνάρτησης και καλεί την μέθοδο statement() για να τις αναγνωρίσει. Στο τέλος κάθε εντολής θα υπάρχει τουλάχιστον μία φορά το σύμβολο ‘;’ που θα το αναγνωρίζουμε και θα καταναλώνουμε, εκτός από την τελευταία εντολή που θα το έχει προαιρετικά.

```
blockstatements→   statement
                   ( ; statement )*
```

Με την σειρά της η statement θα αναγνωρίσει αν η λεκτική μονάδα είναι μεταβλητή ώστε να περάσει στη μέθοδο με την ανάθεση ή αν είναι κάποια δεσμευμένη λέξη για να καλέσει την αντίστοιχη σε αυτή μέθοδο. Αν δεν ισχύει καμία από τις παραπάνω τότε επιστρέφουμε στην συνάρτηση που κάλεσε την μέθοδο αυτή χωρίς να καταναλώσουμε.

```
statement      →   assignStat
               |   ifStat
               |   whileStat
               |   switchcaseStat
               |   forcaseStat
               |   incaseStat
               |   callStat
               |   returnStat
               |   inputStat
               |   printStat
               |   ε
```

Στο κανόνα αυτό εξετάζονται οι εντολές του προγράμματός (π.χ. while,if) για το πόσες εντολές έχουν εσωτερικά. Αν έχουν μία εντολή τότε είναι προαιρετικό να περικλείονται από αγκύλες. Όμως, αν έχουν δύο και παραπάνω εντολές τότε πρέπει να έχουν πριν την πρώτη εντολή

αριστερή αγκύλη ‘{’, ανάμεσα σε κάθε εντολή τουλάχιστον μία φορά το σύμβολο ‘;’ (εκτός από την τελευταία εντολή που θα το έχει προαιρετικά) και μετά την τελευταία εντολή την δεξιά αγκύλη ‘}’.

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

Στο κανόνα assignStat η πρώτη λεκτική μονάδα είναι η μεταβλητή που θα αποθηκεύσει, την καταναλώνει και περιμένει να δει το σύμβολο της ανάθεσης ‘:=’ το οποίο καταναλώνει. Στη συνέχεια καλεί τη μέθοδο expression() για να εντοπίσει τι θα αποθηκευτεί στην μεταβλητή.

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

Ο κανόνας ifstat γνωρίζει ότι η έκφραση που ακολουθεί αποτελεί έλεγχο και αφού καταναλώσει την λεκτική μονάδα ‘if’ περιμένει να δει την αριστερή παρένθεση ‘(‘. Αν την δει την καταναλώνει και καλεί το μη τερματικό σύμβολο condition για διαβάσει τον έλεγχο και στη συνέχεια την δεξιά παρένθεση ‘)’. Αφού την δει καταναλώνει και αυτή την λεκτική μονάδα και καλεί το μη τερματικό σύμβολο statements. Τέλος καλεί την μέθοδο elsepart().

```
ifStat        →    if ( condition )
                       statements
                   elsepart
```

Στο σημείο αυτό προαιρετικά περιμένουμε να αναγνωρίσουμε την δεσμευμένη λέξη else και να την καταναλώσουμε. Έπειτα καλούμε την statements για να βρούμε τις εντολές που περιέχει. Αν δεν βρούμε την else τότε χωρίς να καταναλώσουμε επιστρέφει στην συνάρτηση από την οποία καλέστηκε.

```
elsepart      →    else
                       statements
                |  ε
```

Έχοντας καλέσει το κανόνα αυτό περιμένουμε η πρώτη λεκτική μονάδα που θα βρούμε να είναι η δεσμευμένη λέξη while και να την καταναλώσουμε. Στην συνέχεια εντοπίζουμε την αριστερή παρένθεση ‘(’ και καταναλώνουμε. Μετά θα καλεστεί η condition() για να

διαβάσουμε τους ελέγχους που κάνει και μετά θα πρέπει να βρεθεί η δεξιά παρένθεση ‘)’ και να καταναλωθεί για καταλάβουμε πως αναγνωρίστηκαν όλοι οι έλεγχοι. Τέλος, καλεστεί μη συμβολική σταθερά statements() για να διαβαστούν οι εντολές εσωτερικά της while.

```
whileStat    →    while ( condition )
                       statements
```

Ο κανόνας switchcaseStat περιμένει σαν πρώτη λεκτική μονάδα να βρει την δεσμευμένη λέξη ‘switchcase’ και όταν την βρει την καταναλώνει. Στη συνέχεια ψάχνει την δεσμευμένη λεκτική μονάδα την ‘case’ η οποία μπορεί να εμφανίζεται καμία ή άπειρες φορές. Αν υπάρχει πρέπει να την καταναλώσουμε και μετά να διαβάσουμε την αριστερή παρένθεση ‘(‘ και να την καταναλώσουμε και αυτή. Έπειτα καλείται η μη τερματική λεκτική μονάδα condition για να διαβάσουμε τους ελέγχους που έχει εσωτερικά. Επόμενη λεκτική μονάδα θα είναι η δεξιά παρένθεση ‘)’ και θα καταναλωθεί για να κλείσουν οι έλεγχοι του case. Ελέγχουμε το επόμενο μη τερματικό σύμβολο statements για να διαβάσουμε τον κορμό της εντολής case. Οι παραπάνω έλεγχοι γίνονται όσες φορές εμφανίζεται και η λεκτική μονάδα case. Όταν εμφανιστεί η δεσμευμένη λέξη default η οποία θα πρέπει να υπάρχει σε κάθε κανόνα του switchcaseStat τότε δεν θα έχουμε άλλα case. Αφού την αναγνωρίσαμε την καταναλώνουμε και καλούμε την μέθοδο statements() ώστε να μας δώσει το κώδικα που θα εκτελείται όταν κανένας από τους παραπάνω ελέγχους των case δεν επιβεβαιώνεται.

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

Η forcaseStat έχει την ίδια εξήγηση με την switchcaseStat.

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

Η incaseStat έχει την ίδια εξήγηση με την switchcaseStat με μόνη διαφορά πως δεν υπάρχει η δεσμευμένη λέξη default δηλαδή αν δεν ισχύει κάποιο από τα case τότε επιστρέφουμε στην μέθοδο που μας κάλεσε χωρίς να κάνουμε κάποια λειτουργία.

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

Στο σημείο αυτό, η λεκτική μονάδα που περιμένουμε να υπάρχει είναι η δεσμευμένη λέξη return, όταν την δει την καταναλώνει. Έπειτα αναγνωρίζει την αριστερή παρένθεση ‘(’ και την καταναλώνει και καλεί την μέθοδο expression() η οποία αναγνωρίζει την μεταβλητή που θα επιστρέψει η εντολή return. Στο τέλος περιμένει να υπάρχει η δεξιά παρένθεση για να κλείσει σωστά και να καταναλώσει ώστε να επιστρέψει στην μέθοδο από την οποία καλέστηκε.

```
returnStat    →    return( expression )
```

Έχουμε τον κανόνα callStat ο οποίος αναμένει σαν πρώτη λεκτική μονάδα να εντοπίσει την δεσμευμένη λέξη call και να την καταναλώσει. Μετά θα υπάρχει το όνομα της συνάρτησης που θέλει να διαβάσει και θα το καταναλώσει. Στην συνέχεια θα βρει την αριστερή παρένθεση ‘(’και θα την καταναλώσει ώστε να μπει μέσα στις παραμέτρους με τις οποίες θα καλεστεί η συνάρτηση. Για την ανάγνωση αυτών θα καλέσει την μέθοδο actualparlist(). Επόμενη λεκτική μονάδα θα πρέπει να είναι η δεξιά παρένθεση ‘)’ ώστε να κλείσουν οι παράμετροι με τις οποίες θα καλεστεί και να καταναλώσουμε την ‘)’.

```
callStat      →    call ID( actualparlist )
```

Ο κανόνας printStat χρησιμοποιείται για τα σημεία που το πρόγραμμα προς μετάφραση θέλει να τυπώσει κάτι στην οθόνη. Πρέπει αρχικά να διαβάσει την δεσμευμένη λεκτική μονάδα ‘print’, αφού την καταναλώσει, να βρει την αριστερή παρένθεση ‘(‘ και να την καταναλώσει. Στη συνέχεια ελέγχεται το μη τερματικό σύμβολο expression για να αναγνωρίσει τι ζητείται να εκτυπωθεί. Τέλος περιμένει την λεκτική μονάδα ‘)’, την οποία όταν την βρει την καταναλώνει.

```
printStat     →    print( expression )
```

Ο κανόνας inputStat χρησιμοποιείται για τα σημεία που το πρόγραμμα προς μετάφραση περιμένει να δώσει είσοδο ο χρήστης. Στην αρχή θα δει την δεσμευμένη λεκτική μονάδα ‘input’, την καταναλώνει και στη συνέχεια περιμένει να δει την αριστερή παρένθεση ‘(‘, την οποία καταναλώνει. Το επόμενο βήμα είναι να καλεστεί το μη τερματικό

σύμβολο ID για να διαβάσει την είσοδο από τον χρήστη. Όταν επιστρέψει από την συνάρτηση αυτή θα πρέπει να υπάρχει η δεξιά παρένθεση για να την καταναλώσει και να καταλάβουμε πως ο χρήστης έδωσε την είσοδο.

```
inputStat      →    input( ID )
```

Οι κανόνες actualparlist και actualparitem ακολουθούν την ίδια δομή με τα formalparlist και formalparitem αντίστοιχα. Η διαφορά τους είναι στο σημείο που χρησιμοποιείται η κάθε μία, για να διαβάζουμε τις παράμετρούς όταν καλούμε μία συνάρτηση ενώ οι δεύτεροι όταν ορίζεται μια συνάρτηση.

```
actualparlist →    actualparitem
                   ( , actualparitem )*
               |    ε
actualparitem →    in expression
               |   inout ID
```

Οι κανόνες condition και boolterm ακολουθούν την ιδιά δομή με τον κανόνα blockstatements. Οι διάφορες είναι πως αντί για το ‘;’ ο κανόνας condition έχει το ‘or’ και για τον boolterm έχει το ‘and’. Οι δύο αυτές συναρτήσεις διαβάζουν ελέγχους των εντολών.

```
condition      →    boolterm
                    ( or boolterm )*

# term in boolean expression
boolterm       →    boolfactor
                    ( and boolfactor )*
```

Σε αυτό το κανόνα αναγνωρίζουμε αν στις συνθήκες έχουμε την δεσμευμένη λέξη not ή την αριστερή αγκύλη ‘[’ ή αν θα καλέσουμε κατευθείαν την μέθοδο expression(). Στην πρώτη περίπτωση της άρνησης, καταναλώνουμε τη λεκτική μονάδα του not και στην συνέχεια θα έχουμε αριστερή αγκύλη την οποία θα καταναλώσουμε, για να καλέσουμε την μέθοδο condition(), η οποία θα ορίσει την συνθήκη και τέλος θα κλείσει η συνθήκη με την δεξιά αγκύλη την οποία θα καταναλώσουμε. Στην δεύτερη περίπτωση που θα έχουμε μόνο την συνθήκη θα καταναλώσουμε την λεκτική μονάδα ‘[’ στην συνέχεια θα καλέσουμε την μέθοδο condition() και τέλος θα κλείσει η συνθήκη με

την δεξιά αγκύλη την οποία θα καταναλώσουμε. Στη τελευταία περίπτωση θα έχουμε κάποιον έλεγχο για αυτό το λόγο καλούμε με την σειρά τις μεθόδους expression(), REL_OP() και expression(). Αυτά θα μας δώσουν το δεξί μέλος του ελέγχου, τον έλεγχο που θέλουμε και τέλος το αριστερό μέλος του ελέγχου. Παρατηρούμε πως σε κανένα σημείο δεν καταναλώνουμε αφού δεν έχουμε αναγνωρίσει κάποιο σύμβολο.

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

Ο κανόνας expression χρησιμοποιείται για να αναγνωρίσει αριθμητικούς όρους της πρόσθεσης και της αφαίρεσης. Πρώτα καλεί το μη τερματικό σύμβολο opotionalSign(), το οποίο θα δώσει το πρόσιμο και έπειτα από αυτό θα καλέσουμε την μέθοδο term(). Στη συνέχεια, ελέγχει επαναληπτικά αν υπάρχει το σύμβολο της πρόσθεσης ή της αφαίρεσης για να καλέσει την ADD_OP και αν υπάρχει έπειτα από αυτό καλεί την term() όσες φορές εμφανίζεται κάποιο mul_op.

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

Ο κανόνας term χρησιμοποιείται για να αναγνωρίσει αριθμητικούς όρους του πολλαπλασιασμού και της διαίρεσης. Πρώτα καλεί το μη τερματικό σύμβολο factor. Στη συνέχεια, ελέγχει επαναληπτικά αν υπάρχει το σύμβολο του πολλαπλασιασμού ή της διαίρεσης για να καλέσει την mul_op και αν υπάρχει έπειτα από αυτό καλεί την factor() όσες φορές εμφανίζεται κάποιο mul_op.

```
term          →    factor
                   ( MUL_OP factor )*
```

Ο κανόνας factor κάνει τρεις ελέγχους. Πρώτα εξετάζει αν η λεκτική μονάδα που έχουμε ανήκει στην κατηγορία των αριθμών. Αν αυτός ο έλεγχος δεν αποδώσει, εξετάζεται αν έχουμε αριστερή παρένθεση ‘(‘, καταναλώνεται αυτή η λεκτική μονάδα και καλείται η μέθοδος expression() για να βρει το πρόσημο με τον αριθμό, όταν επιστρέψει αναδρομικά θα βρει την δεξιά παρένθεση ‘)’ και θα την καταναλώσει. Στο τελευταίο έλεγχο, η λεκτική μονάδα θα είναι μια μεταβλητή, την

οποία θα καταναλώσουμε και στην συνέχεια θα καλέσουμε την μέθοδο idtail().

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

Ο κανόνας idtail εκτελείται κάθε φορά μετά από μια συνάρτηση και διαβάζει τις παραμέτρους. Συνεπώς μετά από το όνομα μιας συνάρτησης περιμένει την λεκτική μονάδα '(' την οποία και καταναλώνει. Στη συνέχεια εκτελεί το μη τερματικό σύμβολο actualparlist, επιστρέφει και διαβάζει την λεκτική μονάδα ')', την οποία καταναλώνει. Αν η πρώτη λεκτική μονάδα δεν είναι παρένθεση τότε επιστρέφουμε στην συνάρτηση από την οποία καλέστηκε η idtail() χωρίς να καταναλώσουμε.

```
idtail      →    ( actualparlist )
             |   ε
```

Για τον κανόνα optionalSign όπου μας δίνει το πρόσημο καλείται η μέθοδος ADD_OP() για να αναγνωριστεί το σύμβολο της πρόσθεσης ή της αφαίρεσης. Σε αυτόν τον κανόνα δεν γίνεται κάποια κατανάλωση λεκτικής μονάδας. Αν δεν υπάρχει κανένα από τα δύο σύμβολα τότε επιστρέφουμε αναδρομικά στην συνάρτηση η οποία κάλεσε το optionalSign.

```
optionalSign  →    ADD_OP
               |   ε
```

Για τον κανόνα σύγκρισης λεκτικών μονάδων πρέπει να γίνει έλεγχος για όλους τους τελεστές σύγκρισης που δέχεται η C-imple. Γίνεται έλεγχος για την ισότητα '=', για την διαφορά '<>', για μικρότερο ή μεγαλύτερο '<' και '>' και για μικρότερο ή ίσο και μεγαλύτερο ή ίσο ελέγχουμε τα '<=' και '>=' αντίστοιχα. Τέλος μόλις βρεθεί ένας από τους παραπάνω τελεστές σύγκρισης καταναλώνεται η λεκτική μονάδα και συνεχίζουμε με την μετάφραση.

```
REL_OP      →    = | <= | >=  | > | < | <>
```

Για τον κανόνα αναγνώρισης συμβολών πρόσθεσης γίνεται έλεγχος αν υπάρχει το σύμβολο της πρόσθεσης '+' ή της αφαίρεσης '-' και στη συνέχεια γίνεται κατανάλωση της λεκτικής μονάδας.

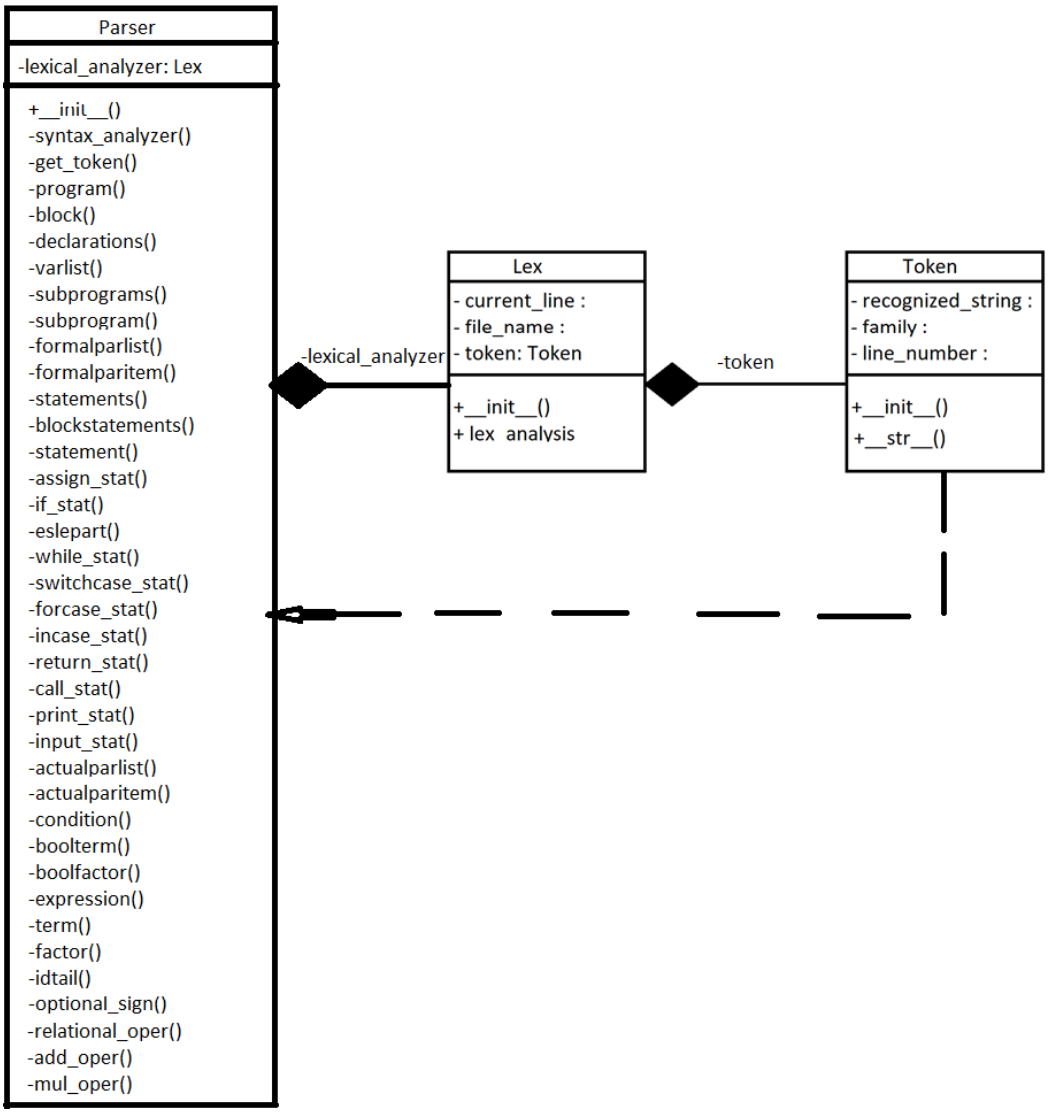
```
ADD_OP          →     + | -
```

Για τον κανόνα αναγνώρισης συμβολών πολλαπλασιασμού γίνεται έλεγχος αν υπάρχει το σύμβολο του πολλαπλασιασμού ‘*’ ή της διαίρεσης ‘/‘ και στη συνέχεια γίνεται κατανάλωση της λεκτικής μονάδας.

```
MUL_OP          →     * | /
```

Σε οποιαδήποτε από τις παραπάνω περιπτώσεις δεν εντοπίσουμε το αναμενόμενο σύμβολο όπως αναφέραμε θα έχουμε κατάλληλο μήνυμα σφάλματος. Θα τυπώνεται στο χρήστη το σημείο του σφάλματος, πληροφορία για το κομμάτι κώδικα στο οποίο ανήκει και σε περιπτώσεις τον κώδικα που ο μεταφραστής περίμενε στο σημείο.

Το UML της κλάσης Parser για τον συντακτικό αναλυτή:



Ενδιάμεσος Κώδικας

Η σχεδίαση του μεταγλωττιστή χωρίζεται σε δύο ανεξάρτητες φάσεις, το εμπρόσθιο τμήμα(front end) και το οπίσθιο τμήμα(back end). Ο ενδιάμεσος κώδικας αποτελεί το στρώμα επικοινωνίας ανάμεσα στα δύο τμήματα. Το στάδιο του ενδιάμεσου κώδικα αφορά την παραγωγή δομών οι οποίες δεν είναι τόσο σύνθετες όσο η αρχική γλώσσα(C-imple) και η συντακτική της ανάλυση είναι τετριμμένη. Το στάδιο αυτό θα γίνεται ταυτόχρονα με τον συντακτικό αναλυτή. Η ενδιάμεση γλώσσα αποτελείται από τετράδες οι οποίες είναι αριθμημένες σε αύξουσα σειρά και ο αριθμός αυτός χρησιμοποιείται σαν ετικέτα. Θα κρατάμε τον αριθμό αυτό στην αρχή κάθε τετράδας άρα θα έχουμε πέντε πεδία

να αποθηκεύουμε. Κάθε τετράδα αποτελείται από έναν τελεστή και τρία τελούμενα, ο τελεστής καθορίζει την ενέργεια που πρόκειται να γίνει και τα τελούμενα είναι εκείνα στα οποία θα εφαρμοστεί η ενέργεια. Λόγο του παραπάνω, η κάθε γραμμή κώδικα του αρχικού προγράμματος μεταφράζεται σε τουλάχιστον μία ή περισσότερες τετράδες. Δημιουργούμε μια λίστα όπου αποθηκεύουμε τις τετράδες και θα είναι το μέσο επικοινωνίας της φάσης της παραγωγής ενδιάμεσου κώδικα και της παραγωγής τελικού κώδικα.

Πιο συγκεκριμένα παρακάτω θα αναλύσουμε τις τετράδες που θα χρησιμοποιήσουμε για την ενδιάμεση αναπαράσταση:

Α) Αναπαράσταση τετράδων

Με τις δυο αυτές εντολές οριοθετούμε την αρχή(begin_block) και το τέλος(end_block) μιας συνάρτησης. Είναι σημαντικό ο ενδιάμεσος κώδικας να έχει κωδικοποιημένη την πληροφορία για την ορατότητα μιας συνάρτησης, διαδικασίας ή του κυρίως προγράμματος. Γι' αυτό και στη δεύτερη θέση της τετράδας αποθηκεύεται το όνομα(name) για την διαδικασία στην οποία αναφερόμαστε.

Ομαδοποίηση κώδικα:

```
begin_block, name, _, _
end_block, name, _, _
```

Όταν αναγνωρίζουμε την εκχώρηση τότε σαν τελεστή θα συμπληρώνουμε το σύμβολο ':='. Στην θέση του πρώτου τελούμενου(source) θα συμπληρώνουμε μια μεταβλητή, μια αριθμητική ή συμβολική σταθερά και στην θέση του τρίτου τελούμενου (target) μία μεταβλητή.

Εκχώρηση:

```
:= , source, _, target
```

Ο τελεστής op χρησιμοποιείται όταν συναντάμε αριθμητικές πράξεις. Στην θέση του πρώτου και δεύτερου τελούμενου σημειώνουμε τις μεταβλητές στις οποίες θα εφαρμοστεί η πράξη, ενώ στο target θα κρατήσουμε από το αποτέλεσμα της πράξης.

Αριθμητική πράξη:

```
op, operand1, operand2, target
```

Η τετράδα αυτή χρησιμοποιείται για να αλλάξει η ροή εκτέλεσης του προγράμματος δίνοντας εντολή να εκτελεστεί η τετράδα με ετικέτα label, η οποία θα αποθηκευτεί στο τρίτο τελούμενο.

Εντολή άλματος:

```
jump, _, _, label
```

Ο τελεστής relop χρησιμοποιείται όταν συναντάμε πράξεις σύγκρισης και αντικαθίσταται από το εκάστοτε σύμβολο(<,>,<>,=,<=,>=). Το πρώτο(a) και δεύτερο(b) τελούμενο είναι οι συγκρινόμενες τιμές ενώ το label είναι η ετικέτα που θα αναλάβει την εκτέλεση σε περίπτωση που η σύγκριση βγει αληθής, θα αποθηκευτεί στο τρίτο τελούμενο.

Εντολή λογικού άλματος:

```
relop , a, b, label
```

Ο τελεστής call χρησιμοποιείται κατά την κλήση συναρτήσεων ή διαδικασιών βάζοντας στην τετράδα ως πρώτο τελούμενο το όνομα(name) τους.

Κλήση συνάρτησης ή διαδικασίας:

```
call, name, _, _
```

Ο τελεστής par χρησιμοποιείται κατά την κλήση μιας συναρτησης ή διαδικασίας. Περνάμε στην τετράδα σαν πρώτο τελούμενο το όνομα της παραμέτρου(name) και σαν δεύτερο τελούμενο το mode τον τρόπο περάσματος(cv,ref,ret) της μεταβλητής στην συνάρτηση. Αυτός μπορεί να είναι: με τιμή (cv) , με αναφορά (ref) και σαν επιστροφή τιμής συνάρτησης (ret). Στην περίπτωση της ret σαν όνομα ο χρήστης δεν έχει ορίσει κάποια μεταβλητή να κάνει return. Για αυτό το λόγο ορίζουμε τις προσωρινές μεταβλητές οι οποίες είναι μοναδικές και δεν διαφέρουν από τις μεταβλητές που έχουν δηλωθεί μέσα στο πρόγραμμα. Για να μπορούμε να τις ξεχωρίσουμε δηλώνουμε πως ξεκινάνε με το γράμμα T. Στη συνέχεια για να εξασφαλίσουμε πως δεν έχουν δηλωθεί από τον προγραμματιστή, εκμεταλλευόμαστε το σύμβολο '_' το οποίο δεν ανήκει στο αλφάβητο της C-imple, άρα θα έχει γίνει της μορφής T_. Τέλος για να μην ξαναχρησιμοποιούμε τις ίδιες προσωρινές μεταβλητές κρατάμε έναν μετρητή ο οποίος αυξάνεται σε κάθε δήλωση μίας προσωρινής μεταβλητής, άρα θα έχει γίνει της μορφής T_1,T_2. Όμως χρειαζόμαστε ακόμα μια εντολή για την επιστροφή τιμής η οποία θα

αντιστοιχίζεται στο return και θα είναι η εξής: `ret, source, _, _` η οποία θα έχει το τελεστή ret και σαν μόνο τελούμενο την μεταβλητή source.

Πέρασμα πραγματικής παραμέτρου:

```
par, name, mode, _
```

Ο τελεστης in χρησιμοποιείται όταν εισάγει από το πληκτρολογιο ο χρήστης. Σαν πρώτο τελούμενο μπαίνει το όνομα της μεταβλητής στο οποίο θα γίνει η αποθήκευση της εισόδου. Αντίστοιχα ο τελεστής out χρησιμοποιείται όταν το πρόγραμμα πρέπει να τυπώσει, σαν πρώτο τελούμενο μπαίνει πάλι η μεταβλητή που πρέπει να τυπωθεί.

Είσοδος-έξοδος:

```
in, x, _, _
out, x, _, _
```

Όταν έχουμε τον τελεστή halt σημαίνει πως τερματίζει το πρόγραμμα δηλαδή μόλις διαβάστηκε όλη η main. Επόμενη τετράδα μετά από το halt θα είναι αυτή του end_block για να τερματίσει το κυρίως πρόγραμμα.

Τερματισμός προγράμματος:

```
halt, _, _, _
```

B) Βοηθητικές συναρτήσεις

Κατά την σχεδίαση του ενδιάμεσου κώδικα παρατηρείται πως κάποιες ενέργειες επαναλαμβάνονται αρκετά συχνά για αυτό θα τις υλοποιήσουμε σαν συναρτήσεις για διευκόλυνση.

genQuad(operator, operand1, operand2, operand3):

Η συνάρτηση αυτή θα δημιουργεί μια νέα τετράδα(με πέντε στοιχεία διότι κρατάμε στη πρώτη θέση, την ετικέτα της νέας τετράδας). Η καινούργια τετράδα θα προκύπτει αυτόματα από την ετικέτα της τελευταίας τετράδας καθώς θα είναι η επόμενή της. Θα κρατάμε στη λίστα την ετικέτα, τον τελεστή(operator) και τα τρία τελούμενα (operand1, operand2, operand3).

nextQuad():

Η συνάρτηση αυτή, επιστρέφει την ετικέτα της επόμενης τετράδας που θέλουμε να δημιουργηθεί.

newTemp():

Η συνάρτηση αυτή, δημιουργεί την επόμενη νέα προσωρινή μεταβλητή, όπως εξηγήσαμε παραπάνω προσθέτει 1, αλλά και την επιστρέφει στο πρόγραμμα από το οποίο καλέστηκε.(υπενθυμίζουμε η προσωρινές μεταβλητές είναι της μορφής T_1,T_2)

emptyList():

Η συνάρτηση αυτή, χρησιμοποιείται για να δημιουργηθεί και να επιστραφεί μία κενή λίστα ώστε να μπορούν να τοποθετηθούν καινούργιες ετικέτες τετράδων.

makeList(label)

Η makeList δημιουργεί μια λίστα ετικετών τετράδων, με μοναδικό στοιχείο την ετικέτα της τετράδας.

mergeList(list1, list2)

Η mergeList έχει ως ρόλο την δημιουργία μιας λίστας ετικετών από τετράδες. Η λίστα αυτή θα περιέχει τις λίστες list1 και list2.

backpatch(list, label)

Η συνάρτηση backpatch ρόλο έχει την συμπλήρωση του τελευταίου στοιχείου των τετράδων που πρέπει να δηλωθεί το επόμενο άλμα σε περίπτωση ελέγχου. Παίρνει από το όρισμα list τις ετικέτες των τετράδων που χρήζουν συμπλήρωσης και τοποθετεί στο τέλος την ετικέτα label.

Γ) Αριθμητικές παραστάσεις

Στις αριθμητικές παραστάσεις κάθε κανόνα των χειριζόμαστε ανεξάρτητα από τους υπόλοιπους, ώστε όταν εκτελεστεί ο συγκεκριμένος κώδικας να τοποθετηθεί σε συγκεκριμένη μεταβλητή. Οι συμβολισμοί {P1} και {P2} δηλώνουν σε ποιο σημείο του κανόνα θα γίνει σημασιολογική ενέργεια και χρησιμοποιούνται για την θεωρητική κατανόηση. Επίσης, τα T1 και T2 που θα χρησιμοποιηθούν δεν αποτελούν διαφορετικό κανόνα, απλά διαφορετική εμφάνιση του κανόνα στην γραμματική και διευκολύνει να διαχωρίζουμε τις εμφανίσεις τους. Στο μεταγλωττιστή μας η βασική τους χρήση θα είναι σαν κλήσεις συναρτήσεων, δηλαδή θα έχουμε διαφορετικές κλήσεις της ίδιας συνάρτησης. Ακόμα, θα δίνουμε το όνομα place στις μεταβλητές αυτές που αναφέραμε παραπάνω (στον κώδικα συμπληρώνονται με το σύμβολο '_' μπροστά από την λέξη place).

$$E \rightarrow T^1 (+ T^2 \{P_1\})^* \{P_2\} \quad E \rightarrow T^1 (- T^2 \{P_1\})^* \{P_2\}$$

Ο πρώτος κανόνας που θα δούμε θα είναι για την πρόσθεση και την αφαίρεση. υλοποιήσουμε το κανόνα στην συνάρτηση expression() όπου και ελέγχονται οι πράξεις αυτές.

{P1} : Αρχικά θα καλούμε την βοηθητική συνάρτηση newTemp() για να μας δημιουργεί μια καινούργια προσωρινή μεταβλητή η οποία θα αποθηκεύσει το μέχρι στιγμής αποτέλεσμα. Η μεταβλητή αυτή χρησιμοποιείται κυρίως για όταν έχουμε παραπάνω από 1 πράξεις δηλαδή όταν στο θεωρητικό κομμάτι έχουμε το Kleene star. Στην συνέχεια, παράγουμε μια τετράδα που θα προσθέσει/αφαιρέσει τις εκάστοτε μεταβλητές T1.place και T2.place και το αποτέλεσμα θα το εκχωρήσει σε μια μεταβλητή η οποία δεν θα επηρεάσει το αποτέλεσμα του προγράμματος, δηλαδή σε μια προσωρινή μεταβλητή. Στο τέλος της ενέργειας αυτής θα τοποθετήσουμε το αποτέλεσμα αυτό στην μεταβλητή T1.place διότι μετά την πρώτη φορά η μεταβλητή αυτή δεν θα πάρει καινούργια τιμή, θα χρησιμοποιείται μόνο αν υπάρξει κάποιο επόμενο T2. Σημειώνουμε πως αφού ο κανόνας P1 βρίσκεται εσωτερικά του Kleene star δεν γνωρίζουμε πόσες φορές θα καλεστεί. Οπότε όσο αναγνωρίζουμε σύμβολα πρόσθεσης ή αφαίρεσης θα τον καλούμε επαναληπτικά.

{P2} : Στο σημείο αυτό θα περνάμε όταν δεν αναγνωρίζουμε άλλη μεταβλητή T2 και θέλουμε να περάσουμε την τελευταία προσωρινή μεταβλητή(κρατάει το αποτέλεσμα όλης της παράστασης) που δημιουργήθηκε στην μεταβλητή του κανόνα(E.place). Θα εκτελούμε την εντολή E.place = T1.place, η οποία θα εκτελείται εκτός της επανάληψης καθώς θέλουμε να κρατήσουμε μόνο το τελικό αποτέλεσμα της παράστασης.

$T \rightarrow F^1 (\times F^2 \{P_1\})^* \{P_2\}$ $T \rightarrow F^1 (/ F^2 \{P_1\})^* \{P_2\}$

Στον δεύτερο κανόνα έχουμε τον πολλαπλασιασμό και την διαίρεση. Θα τον υλοποιήσουμε το κανόνα στην συνάρτηση term() η οποία ελέγχει τις συγκεκριμένες πράξεις.
Παρατηρούμε πως η δομή αυτού του κανόνα είναι ίδια με το πρώτο κανόνα που είδαμε άρα θα ακολουθήσουμε τα ίδια βήματα με μόνη διαφορά πως οι προσωρινές μεταβλητές θα ονομάζονται F1.place και F2.place, ενώ το αποτέλεσμα θα περνιέται στην T.place.

$F \rightarrow (E) \{P_1\}$

Στον τρίτο κανόνα παρουσιάζεται η προτεραιότητα των πράξεων. Θα τον υλοποιήσουμε το κανόνα στην συνάρτηση factor() καθώς σε αυτή την συνάρτηση περιέχεται το αποτέλεσμα των πράξεων που έγιναν εσωτερικά των παρενθέσεων.

{P1} : Στο σημείο αυτό θα κάνει μεταφορά(αντιγραφεί) του αποτελέσματος από την μεταβλητή E.place στην F.place. Για να γίνει σωστή μεταφορά είναι απαραίτητο να έχουμε κρατήσει το αποτέλεσμα της συνάρτησης expression() (η οποία μας επιστρέφει το αποτέλεσμα των πράξεων εντός των παρενθέσεων όπως είδαμε στο συντακτικό αναλυτή) στην μεταβλητή E.place.

$F \rightarrow id \{P_1\}$

Στο τελευταίο κανόνα των αριθμητικών παραστάσεων εκτελούμε ανάλογη λογική με τον τρίτο κανόνα. Στην ίδια συνάρτηση (factor),

αποθηκεύοντας όμως το αποτέλεσμα της συνάρτησης idtail() η οποία θα μας επιστρέψει την μεταβλητή id. Η μεταβλητές που θα χρησιμοποιήσουμε είναι id.place και F.place.

Δ) Λογικές συνθήκες

Οι λογικές παραστάσεις έχουν κανόνες οι οποίοι είναι ανεξάρτητοι μεταξύ και ο κάθε κανόνας δημιουργεί ένα νέο αποτέλεσμα. Ωστόσο, τα αποτελέσματα αυτά δεν είναι μεταβλητές αλλά διαχείριση πληροφορίας, μόνο ένας κανόνας παράγει ενδιάμεσο κώδικα. Αυτό γίνεται διότι οι τετράδες δεν γνωρίζουμε πως θα συμπληρωθούν στο τρίτο τελούμενο δηλαδή σε ποιο σημείο του κώδικα πρέπει να κάνουν jump. Για αυτό λόγο κάθε κανόνας που καλείται λαμβάνει, αλλά και προετοιμάζει για τον κανόνα που τον κάλεσε δύο λίστες, την λίστα true και την λίστα false. Αυτές οι λίστες θα περιέχουν τετράδες που έχουν μείνει ασυμπλήρωτες και θα πρέπει να συμπληρωθούν με την ετικέτα εκείνης της τετράδας στην οποία θα πρέπει να μεταβεί ο έλεγχος, αν η συνθήκη είναι αληθής(λίστα true) και αν η συνθήκη είναι ψευδής(λίστα false).

$$B \rightarrow Q^1 \{P_1\} (\text{or } \{P_2\} Q^2 \{P_3\})^*$$

Ο κανόνας B θα τοποθετηθεί στην συνάρτηση condition() καθώς σε αυτή αναγνωρίζει ο συντακτικός αναλυτής το σύμβολο or. Ο κανόνας αυτός θα διαχειριστεί τις ασυμπλήρωτες τετράδες που έρχονται στις λίστες.

{P1} : Μετά την ενεργοποιήσει του κανόνα Q1, θέλουμε να συμπληρώσουμε όσο περισσότερες τετράδες μπορούμε. Θα περάσουμε τις Q1.true και Q1.false(που λάβαμε) στις τοπικές λίστες B.true και B.false για να μπορούμε να τις διαχειριστούμε.

{P2} : Για να φτάσουμε στο συμβολισμό p2 σημαίνει πως έχουμε διαβάσει τουλάχιστον ένα σύμβολο or δηλαδή ότι ο προηγούμενος κανόνας απέτυχε. Γνωρίζουμε λοιπόν πως οι τετράδες από την τοπική λίστα B.false πρέπει να συμπληρωθούν με την τετράδα που θα δημιουργηθεί. Για αυτό το λόγο καλούμε την βοηθητική συνάρτηση backpatch με την λίστα B.false και σαν δεύτερο όρισμα την ετικέτα της

επόμενης τετράδας, ώστε να συμπληρωθεί στο τρίτο τελούμενο δηλαδή η θέση που θα γίνει το jump(πέμπτη θέση). Αυτό θα πρέπει να γίνεται επαναληπτικά διότι κάθε φορά μετά το or θα έχει αποτύχει ο προηγούμενος κανόνας (φαίνεται και από το kleen star).

{P3} : Η λίστα Q2.true περιέχει τετράδες οι οποίες θα κάνουν λογικό άλμα στο ίδιο σημείο με τις τετράδες στην τοπική λίστα B.true, αν έστω μια από αυτές αποτιμηθεί αληθής. Στην Q2.true αποθηκεύονται οι τετράδες από άλλο κανόνα οι οποίες μπορεί μεταξύ τους να χωρίζονται με το σύμβολο and. Για αυτό το λόγο, για αυτές τις δύο λίστες θα καλέσουμε την βοηθητική συνάρτηση mergelist, εκεί θα συσσωρευτούν οι τετράδες που θα μας οδηγήσουν εκτός της λογικής συνθήκης. Στη συνέχεια, την λίστα Q2.false που επιστράφηκε θα την αποθηκεύσουμε στην B.false, η οποία άδειασε λόγο του backpatch, γιατί αυτές θα θέλουμε να κάνουν άλμα στο σημείο του κώδικα όπου η συνθήκη του κανόνα αυτού δεν ισχύει.

$$Q \rightarrow R^1 \{P_1\} (\text{and } \{P_2\} R^2 \{P_3\})^*$$

Ο κανόνας Q θα τοποθετηθεί στην συνάρτηση boolterm() καθώς σε αυτή αναγνωρίζει ο συντακτικός αναλυτής το σύμβολο and. Ο κανόνας αυτός θα διαχειριστεί τις ασυμπλήρωτες τετράδες που έρχονται στις λίστες.
Απαραίτητο είναι να αναφέρουμε πως ο κανόνας B και ο κανόνας Q θα έχουν ομοιότητες, όπως καταλαβαίνουμε και από το σχήμα παραπάνω. Η μόνη διαφορά που παρατηρείται είναι πως είναι αντίθετοι στην αποτίμηση των συνθηκών. Πιο συγκεκριμένα, στον κανόνα Q όταν μία συνθήκη αποτυγχάνει και βρίσκεται αριστερά ενός and τότε μεταβαίνουμε έξω από την συνθήκη. Αυτό συμβαίνει διότι θέλουμε όλες οι συνθήκες να είναι αληθής. Όταν μια συνθήκη κρίνεται αληθής αριστερά από το and τότε ο έλεγχος μεταβαίνει δεξιά της συνθήκης για να ελέγξει και την επόμενη συνθήκη. Άρα παρατηρούμε πως όλη η σκέψη και οι εντολές θα είναι παρόμοιες αλλά όπου ο παραπάνω κανόνας(B) είχε Q.true ή B.true εδώ θα έχει Q.false ή B.false αντίστοιχα και όπου ο παραπάνω κανόνας(B) είχε Q.false ή B.false εδώ θα έχει Q.true ή B.true.

$R \rightarrow (B) \{P_1\}$ $R_2 \rightarrow \text{not} (B) \{P_1\}$

Οι δύο αυτοί κανόνες θα κάνουνε μεταφορά των τετράδων από τις λίστες του B(B.true, B.false) στις λίστες του R(B.true, B.false), δεν θα παραχθεί εδνιάμεσος κώδικας αφού οι έλεγχοι γίνονται στη συντακτική ανάλυση. Ο κανόνας R θα μεταφέρει από το true στο true και από το false στο false ενώ ο R2 το αντίστροφο δηλαδή από το true στο false και από το false στο true. Οι κανόνες αυτόι θα συμπληρωθούν στην συνάρτηση boolfactor() η οποία κάνει τους αντίστοιχους ελέγχους.

$R \rightarrow E^1 \text{ relop } E^2 \{P_1\}$

Ο κανόνας αυτός θα βρίσκεται στην ίδια συνάρτηση με τους δύο παραπάνω κανόνες (R και R2) για αυτό έχει το ίδιο όνομα. Ο συγκεκριμένος κανόνας είναι ο μόνος ο οποίος δημιουργεί ενδιάμεσο κώδικα και αυτό συμβαίνει γιατί κάνει σύγκριση δύο αριθμητικών εκφράσεων, ελέγχει μία συνθήκη. Η τετράδα που θα παραχθεί θα είναι αυτή που θα πραγματοποιεί το λογικό άλμα στην περίπτωση που η συνθήκη ισχύει.

{P1} : Στο κανόνα αυτό λόγο του ότι οι τετράδες των αλμάτων δημιουργούνται για αυτό δημιουργούνται και οι λίστες(B.true, B.false), δεν είναι όπως οι παραπάνω λίστες που εξηγήσαμε. Η βοηθητική συνάρτηση makelist() θα κληθεί για να μπορούμε να έχουμε ως παράμετρο το nextQuad(), δηλαδή την ετικέτα της επόμενης τετράδας. Με αυτό το τρόπο θα δημιουργηθεί μια νέα λίστα, με μοναδαικό στοιχείο την ετικέτα της επόμενης τετράδας, ταυπόλοιπα πεδία τα είναι άδεια. Στη συνέχεια θα καλεστεί η βοηθητική συνάρτηση genQuad() με τελεστή(πρώτο στοιχείο της τετράδας) το σύμβολο της σύγκρισης που θα έχουμε. Τα δύο πρώτα τελούμενα(θέσεις 2-3 στη τετράδα) που θα περαστούν θα είναι τα δύο expression() (π.χ. integers,strings) που θα συγκρίνουμε. Άρα αν ισχύει η συνθήκη θα μεταβεί ο έλεγχος στην ετικέτα που ορίζει η τετράδα. Αν όμως η συνθήκη αποτιμηθεί ψευδές τότε το άλμα με την εντολή του genQuad() δεν θα εκτελεστεί και θα μεταβούμε στην επόμενη εντολή. Τέλος θα υπάρχει η εντολή genQuad('jump','_' ,'_' ,'_') η οποία θα εκτελείται αν η συνθήκη δεν ισχύει. Συνοψίζοντας οι δύο πρώτες εντολές (B.true και genQuad) θα δημιουργήσουν τη μη συμπληρωμένη τετράδα και θα την εισάγουν στη

λίστα των μη συμπληρωμένων τετράδων για την αληθή αποτίμηση της relop, ενώ οι δύο υπόλοιπες για την ΜΗ αληθή αποτίμηση.

Εντολή return:

S -> return (E) {P1}

Ο κανόνας S θα συμπληρωθεί στη συνάρτηση return_stat() διότι σύμφωνα με το συντακτικό αναλυτή σε εκείνη την συνάρτηση αναγνωρίζουμε την δεσμευμένη λέξη return. Στη συνέχεια, όταν διαβάσουμε το expression() που θα μας δείξει ποια μεταβλητή θέλουμε να επιστρέψει η συνάρτηση θα την αποθηκεύσουμε στην μεταβλητή E.place όπως κάναμε παραπάνω. Θα κρατήσουμε αυτή την μεταβλητή διότι επόμενο βήμα είναι να δημιουργήσουμε τον κατάλληλο ενδιάμεσο κώδικα καλώντας την βοηθητική συνάρτηση genQuad(). Σαν τελεστή θα έχει την λέξη 'retv' και σαν πρώτο τελούμενο την μεταβλητή E.place που κρατήσαμε.

Εντολή εισόδου-εξόδου:

S -> input (id) {P1} S -> print (E) {P2}

Κατά αντίστοιχο τρόπο με το return θα συμπληρώσουμε τα print και input. Η μόνη διαφορά θα είναι οι συναρτήσεις στις οποίες θα τοποθετηθούν, οι οποίες είναι η print_stat() και input_stat() αντίστοιχα.

Εντολή εκχώρησης:

S -> id := E {P1};

Ο κανόνας αυτός θα συμπληρωθεί στη συνάρτηση assign_stat() διότι αυτή η συνάρτηση αναγνωρίζει το σύμβολο της εκχώρησης ':='. Κατά αντίστοιχο τρόπο με το κανόνα για το return θα κρατήσουμε την μεταβλητή E.place από το expression(). Η διαφορά στη συγκεκριμένη τετράδα που θα δημιουργήσουμε είναι πως θα πρέπει όταν καλούμε την βοηθητική συνάρτηση genQuad() εκτός από το τελεστή ':=' και την μεταβλητή e.place, θα τοποθετούμε στο τρίτο τελούμενο(πέμπτη θέση μέσα στην τετράδα) την μεταβλητή στην οποία θέλουμε να κρατήσει

την τιμή. Την μεταβλητή αυτή, ο συντακτικός αναλυτής την αναγνωρίζει πριν διαβάσει το σύμβολο της εκχώρησης, για αυτό το λόγο θα πρέπει να το αποθηκεύουμε πριν εντοπίσουμε την ανάθεση.

Κλήσεις συναρτήσεων και πέρασμα παραμέτρων:

call assign_v (in a, inout b)

Η κλήση συναρτήσεων πρέπει να μεταφραστεί σε ενδιάμεσο κώδικα μαζί με τις πιθανές παραμέτρους με τις οποίες καλείται. Βλέποντας την δεσμευμένη λέξη call στον συντακτικό αναλυτή πρέπει πρώτα να πάμε εντός των παρενθέσεων της κλήσης και να μεταφράσουμε σε ενδιάμεσο κώδικα τις παραμέτρους. Όταν μια παράμετρος περνιέται με το πρόθεμα in, αυτό σημαίνει ότι την περνάμε με τιμή, άρα μπορεί να είναι καθαρός αριθμός ή μεταβλητή. Η τετράδα που θα πρέπει να φτιαχτεί για την παράμετρο πρέπει στην πρώτη θέση(δεύτερη ουσιαστικά επειδή είναι πεντάδα με πρώτη την ετικέτα) να έχει τον τελεστή ‘par’. Αυτό δηλώνει σε όποιον διαβάσει την τετράδα ότι έχουμε μια παράμετρο. Στη δεύτερη θέση μπαίνει η πράξη ή το όνομα της μεταβλητής που έθεσε ο χρήστης και στην τρίτη το τελούμενο ’CV’ που δηλώνει σε όποιον διαβάζει τον ενδιάμεσο κώδικα ότι η παράμετρος έχει περαστεί με τιμή. Όταν μια παράμετρος περνιέται με το πρόθεμα inout, αυτό σημαίνει ότι την περνάμε με αναφορά, άρα μπορεί να είναι μόνο μεταβλητή αυτό που θα περαστεί μέσα. Η τετράδα που θα πρέπει να φτιαχτεί για την παράμετρο είναι παρόμοια με την τετράδα για το πέρασμα παραμέτρου με τιμή με διαφορά ότι στη δεύτερη θέση θα βρίσκεται το όνομα της παραμέτρου και στην τρίτη θέση το τελούμενο ‘REF’ που υποδηλώνει το πέρασμα με αναφορά. Αφού φτιαχτούν αυτές οι τετράδες για κάθε παράμετρο της κλήσης μιας συνάρτησης ή διαδικασίας πρέπει να φτιάξουμε και την τετράδα που λέει ποια συνάρτηση καλούμε τελικά με τις παραμέτρους που φτιάξαμε. Στην τετράδα αυτή θα βάλουμε στην πρώτη θέση τον τελεστή ‘call’ που υποδηλώνει σε όποιον διαβάζει τον ενδιάμεσο κώδικα ότι πρόκειται για κλήση συνάρτησης ή διαδικασίας, στη δεύτερη το όνομα της συνάρτησης και οι υπόλοιπες μένουν κενές.

Πέρασμα παραμέτρου με τιμή: **par, a, CV, _**

Όταν μια παράμετρος περνιέται με τιμή, τότε ότι αλλαγές γίνουν μέσα στη συνάρτηση, στη μεταβλητή που περάστηκε μένουν εκεί και δεν έχουν καμία επίπτωση στην ίδια. Όταν λοιπόν γυρίσουμε στην καλούσα και στην περίπτωση της μεταβλητής προσπαθήσουμε να την ξαναδιαβάσουμε θα έχει πάντα την τιμή που είχε πριν από την κλήση.

Πέρασμα παραμέτρου με αναφορά: **par, b, REF, _**

Όταν μια παράμετρος περνιέται με αναφορά, τότε ότι αλλαγές γίνουν μέσα στη συνάρτηση στην συγκεκριμένη μεταβλητή περνιόνται στη διεύθυνση μνήμης της και μπορούμε να τις δούμε όταν την διαβάσουμε ξανά στην καλούσα.

error = assign_v (in a, inout b)

Για την κλήση συναρτήσεων, πιο συγκεκριμένα χρειάζεται να υπάρχει και μια επιστροφή τιμής με return. Αυτή η τιμή μπορεί, αν την έχει ορίσει έτσι ο χρήστης, να αποθηκεύεται σε μια μεταβλητή. Βλέποντας μια ανάθεση σε μεταβλητή και μετά λεκτική μονάδα που ανήκει στα id κατά καταλαβαίνουμε ότι χρειάζεται με την genQuad να φτιάξουμε μια τετράδα στον ενδιάμεσο κώδικα που να κρατάει την μεταβλητή στην οποία θα πρέπει να επιστραφεί η τιμή. Αυτή η τετράδα δεν θα είναι για ανάθεση αλλά για επιστροφή τιμής. Στην πρώτη θέση της τετράδας θα βάλουμε τον τελεστή ‘par’, στην δεύτερη το όνομα της μεταβλητής και στην τρίτη το τελούμενο ‘RET’. Στη συνέχεια με την genQuad δημιουργείται η τετράδα για την κλήση της συνάρτησης με πρώτο τελεστή το ‘call’, στη δεύτερη θέση μπαίνει το όνομα της συνάρτησης και οι υπόλοιπες μένουν κενές.

Δομή while:

```
whileStat      →    while {p0} ( condition ) {p1}
                    statements {p2}
```

Η δομή της while αποτελείται από μια συνθήκη (condition) και τον κορμό της (statements). Η while είναι μια επαναληπτική δομή γι αυτό πρέπει να μπορεί να γίνεται ο έλεγχος της συνθήκης την πρώτη φορά και κάθε φορά που τελειώνει η εκτέλεση των statements. Για να το εξασφαλίσουμε αυτό πρέπει να κρατηθεί η ετικέτα της πρώτης εντολής της συνθήκης και να την αξιοποιήσουμε αργότερα.
{P1}: Για την περίπτωση που η συνθήκη ισχύει κάνουμε backpatch για την λίστα true και τοποθετούμε την ετικέτα της αρχής των statements ώστε να μπορούμε να τα εκτελέσουμε όταν το condition ισχύει.
{P2}: Αντίθετα για την περίπτωση που η συνθήκη δεν θα ισχύει θα πρέπει να γίνει ένα backpatch στη λίστα false και να συμπληρωθούν οι τετράδες με την ετικέτα της εντολής αμέσως μετά τα statements. Στο τέλος των statements και πριν το backpatch στη λίστα false, πρέπει να δημιουργηθεί με genQuad μια τετράδα που θα κάνει άλμα(jump) στην πρώτη τετράδα της συνθήκης που κρατήσαμε στην αρχή για να δουλεύει σωστά η επανάληψη και να ξαναγίνει ο έλεγχος.

Δομή if:

```
ifStat     →  if ( condition ) {p1} statements(1)  {p2}
              elsePart {p3}
```

Η δομή if αποτελείται από έναν έλεγχο συνθήκης(condition), τον κορμό-τον κώδικα (statements) και από το προαιρετικό else κομμάτι.

{P1}: Για την περίπτωση που η συνθήκη ισχύει, κάνουμε backpatch για την λίστα true και τοποθετούμε την ετικέτα της πρώτης τετράδας των statements για να εκτελεστούν οι εντολές μέσα στην if.

{P2}: Για την περίπτωση που η συνθήκη δεν ισχύει, κάνουμε backpatch για την λίστα false και τοποθετούμε την ετικέτα της πρώτης τετράδας μετά τα statements για να γίνει άλμα εκεί διότι αν δεν ισχύει η συνθήκη δεν θέλουμε να εκτελεστούν οι εντολές μέσα στην if.

{P3}: Αν υπάρχει else τότε πρέπει να γίνει κατάλληλο jump ώστε αν έχουν εκτελεστεί τα statements μέσα στην if να παρακαμφθούν αυτές

μέσα στο else. Πρέπει λοιπόν να δημιουργηθει, μετά το τέλος των statements της if και πριν το backpatch για τη λίστα false, μια τετραδα jump με το genQuad που θα πηγαίνει τη ροή εκτέλεσης μετά τα statements του else και ένα backpatch για τη jump που θα την συμπληρώσει με την ετικέτα της εντολής μετά τα statements της else. Αν δεν έχουν εκτελεστεί οι εντολές της if τότε η ροή εκτέλεσης θα συνεχίζεται κανονικά προς την else αν αυτή υπάρχει.

Δομή switchcase:

```
switchcaseStat  →   switchcase {p0}
                    ( case ( condition ) {p1}
                        statements(1)  {p2} )*
                    default statements(2)
                    {p3}
```

Η δομή switchcase αποτελείται ουσιαστικά από περιπτώσεις(cases) (δεν υπάρχει περιορισμός στον αριθμό) με συνθήκες(conditions) και κορμό(statements) για την περίπτωση που τα conditions ισχύουν. Αν κανένα από τα condition των cases δεν ισχύει τότε υπάρχουν τα default statements που δεν χρειάζονται κάποιο condition για να εκτελεστούν.

{P0}: Πριν το πρώτο case τοποθετούμε μια λίστα την exitlist με την emptylist στην οποία θα μπαίνουν όλες οι τετράδες jump ώστε στο τέλος της switchcase να συμπληρωθούν με την ετικέτα της πρώτης τετράδας μετά το τέλος της switchcase.

{P1}: Σε περίπτωση που το ένα condition είναι αληθές, ο έλεγχος πρέπει να μεταφερθεί στην πρώτη τετράδα των statements που ελέγχθηκαν. Γίνεται backpatch στη λίστα true τοποθετώντας την ετικέτα της πρώτης τετράδας των statements.
{P2}: Αν το condition αποτιμηθεί ως ψευδές τότε ο έλεγχος πρέπει να μεταβεί στο επόμενο condition. Αν δεν υπάρχει επόμενο condition τότε ο έλεγχος πρέπει να περάσει κατευθείαν στο default statements. Εδώ πριν από το backpatch πρέπει να δημιουργήσουμε μια ασυμπλήρωτη τετραδα jump με το genQuad για να μπορούμε να βγούμε από τη ροή του switchcase οποίο statement κ αν εκτελεστεί. Στην exitlist προσθέτουμε την τετράδα που πρέπει να συμπληρωθεί η ετικέτα για το jump. Τέλος για την περίπτωση που η συνθήκη είναι ψευδής πρέπει να κάνουμε ένα backpatch που θα μας βάζει στη λίστα false την ετικέτα της

τετράδας αμέσως μετά τα statements.
{P3}: Σε αυτό το σημείο πρέπει να γίνει backpatch στην λίστα exitlist για να συμπληρωθούν με την ετικέτα της τετράδας αμέσως μετά το τέλος της switchcase, οι τετράδες jump που έχουν μείνει κενές.

Δομή forcase:

```
forcaseStat  →   forcase  {p1}
                 ( case ( condition ) {p2}
                     statements(1)  {p3} )*
                 default statements(2)
```

Η δομή forcase αποτελείται από περιπτώσεις(cases) για τις οποίες αν ισχύει η συνθήκη(condition) εκτελείται το statement και ο ροή εκτέλεσης μεταβαίνει ξανά στην αρχή της δομής επαναληπτικά. Αν δεν ισχύει κανένα από τα conditions εκτελείται το default statement και γίνεται έξοδος από την δομή.

{P1): Σε αυτό το σημείο που μόλις έχει διαβαστεί η δεσμευμένη λέξη forcase και πρέπει να κρατηθεί το σημείο που αρχίζουν οι συνθήκες ώστε αν χρειαστεί να ξανά εκτελεστούν οι συνθήκες της δομής. Κρατάμε λοιπόν εδώ με την nextQuad την ετικέτα της τετράδας που ξεκινάνε οι έλεγχοι των συνθήκων.
{P2}: Σε αυτό το σημείο έχει αποτιμηθεί το condition για κάθε case και πρέπει να αποφασιστεί αν θα εκτελεστούν τα statements. Για την περίπτωση που ο κώδικας θα εκτελεστεί πρέπει να πάρουμε την λίστα true και να κάνουμε backpatch την ετικέτα του πρώτου statement του case.
{P3}: Σε αυτό το σημείο πρέπει να μπει αρχικά με genQuad η τετράδα που θα μας κάνει jump στην αρχή του forcase {P1} όταν θα έχουν εκτελεστεί τα statements ενός case. Στη συνέχεια πρέπει να γίνει το backpatch για την λίστα false στην οποία θα αντικατασταθεί το τελευταίο στοιχείο με την ετικέτα αμέσως μετά από τα statements του case που εξετάστηκε. Αυτό το κάνουμε ώστε αν το case που μόλις εξετάστηκε δεν ισχύει, να εξεταστεί το επόμενο αν υπάρχει αλλιώς να πάει στο default και να γίνει έξοδος από το forcase.

Δομή incase:

```
incaseStat  →   incase  {p1}
                ( case ( condition ) {p2}
                    statements(1)  {p3} )*
                default statements(2)
```

Η δομή incase χρησιμοποιεί περιπτώσεις(cases) για τις οποίες εξετάζεται η συνθήκη(condition) και για όσες ισχύει εκτελείται ο κορμός του case(statements). Αν έστω και ένα από τα conditions βγει αληθές τότε πάμε ξανά στην αρχή της δομής. Όταν εξεταστούν όλα και δεν βγει κανένα condition αληθές θα γίνει έξοδος από τη δομή.

{P1}: Σε αυτό το σημείο πρέπει να κρατήσουμε με nextQuad την ετικέτα της επομένης τετράδας ώστε κάθε φορά που χρειάζεται να ξανά εξεταστούν όλα τα conditions της incase, να πηγαίνουμε στο πρώτο case. Για να ξέρουμε αν έστω και ένα condition έχει βγει αληθές και έχουν εκτελεστεί τα statements του πρέπει να κατασκευάσουμε μια μεταβλητή που θα λειτουργεί ως flag και η τιμή της θα εξετάζεται σε επίπεδο εκτέλεσης και όχι μετάφρασης ακριβώς πριν εκτελεστούν τα default statements. Δημιουργούμε λοιπόν μια προσωρινή μεταβλητή(Temp) για τον σκοπό αυτό και αφού κρατήσουμε την ετικέτα της επομένης τετράδας που προαναφέραμε, πρέπει με genQuad να δημιουργήσουμε την τετράδα για τη μεταβλητή αναθέτοντας της την τιμή 0. Αυτό το κάνουμε ώστε όταν η τιμή του flag γίνει 1 και έχουν εκτελεστεί τα statements ενός case να το καταλάβουμε.
{P2}: Σε αυτό το σημείο πρέπει να γίνει backpatch στην λίστα true και να προστεθεί η ετικέτα του πρώτου statement για το condition που εξετάστηκε και βγήκε αληθές.
{P3}: Σε αυτό το σημείο ένα από τα conditions έχει βγει αληθές και έχουν εκτελεστεί τα statements του οπότε πρέπει το flag το οποίο φτιάξαμε παραπάνω και αρχικοποιήσαμε σε 0 να το κάνουμε 1. Αυτό το κάνουμε φτιάχνοντας μια τετράδα με το genQuad με τελεστή το ‘:=’, πρώτο τελούμενο το 1 για την αλλαγή της τιμής, δεύτερο τελούμενο κενό και τρίτο το όνομα της μεταβλητής. Σηματοδοτούμε έτσι ότι έχουν εκτελεστεί τα statements από τουλάχιστον ένα case. Στη συνέχεια πρέπει να υποδείξουμε το σημείο στο οποίο θα συνεχιστεί η εκτέλεση του κώδικα σε περίπτωση που το condition που εξετάστηκε δεν βγει αληθές. Αυτό το κάνουμε με backpatch στη λίστα false και συμπλήρωση

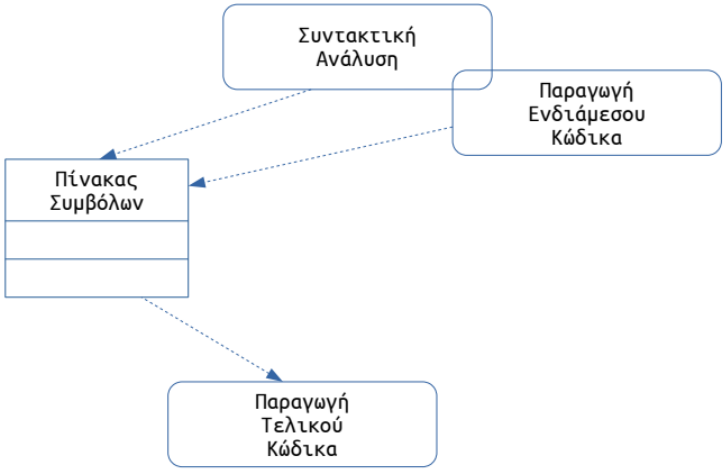
της τελευταίας θέσης με την ετικέτα αμέσως μετά το statements.
{P4}: Τέλος, πριν το default statements πρέπει να γίνει ένας έλεγχος για το αν το flag έχει γίνει 1 και έχει βγει κάποιο από τα παραπάνω conditions αληθές ώστε να ξανάπαμε στην αρχή της δομής αλλιώς να εκτελέσουμε τα default statements και να γίνει έξοδος από τη δομή. Αυτό γίνεται σε επίπεδο εκτέλεσης δημιουργώντας με genQuad μια τετράδα που εξετάζει αν η τιμή του flag είναι 1 και αν είναι αλλάζει τη ροή εκτέλεσης του κώδικα στην ετικέτα που έχουμε κρατήσει στην αρχή του incase. Πρώτος τελεστής στην τετράδα μπαίνει το '=', πρώτο τελούμενο μπαίνει το όνομα της μεταβλητής που θέλουμε να ελέγξουμε, δεύτερο τελούμενο μπαίνει το '1' και τρίτο η ετικέτα για να πάμε στην αρχή του incase.

Στο σημείο αυτό θα θέλαμε να σημειώσουμε πως εσωτερικά στο μεταγλωττιστή μας δημιουργούμε δύο αρχεία, ένα .c και ένα .int. Αυτά αποτελούν αναπαράσταση του ενδιάμεσου κώδικα το ένα στην γλώσσα C και το άλλο το δημιουργούμε εκμεταλλευόμενοι πως στην C-imple έχουμε μόνο integers. Με την εκτέλεση του κώδικα αυτά τα δύο αρχεία θα δημιουργηθούν.

Πίνακας Συμβόλων

Ο πίνακας συμβόλων αποτελεί την τέταρτη φάση της μεταγλώττισης. Είναι μια δυναμική δομή στην οποία αποθηκεύεται πληροφορία που θα χρησιμοποιηθεί στο υπό μεταγλώττιση πρόγραμμα. Η δομή αυτή παρακολουθεί τη μεταγλώττιση και μεταβάλλεται δυναμικά, με την προσθήκη ή αφαίρεση πληροφορίας. Αυτό γίνεται ώστε να περιέχει κάθε φορά μόνο τις εγγραφές εκείνες που οι κανόνες εμβέλειας της γλώσσας επιτρέπουν, δηλαδή αυτές που το υπό μεταγλώττιση πρόγραμμα έχει δικαίωμα να βλέπει. Πιο συγκεκριμένα, στον πίνακα συμβόλων αποθηκεύεται πληροφορία που σχετίζεται με τις μεταβλητές του προγράμματος, τις διαδικασίες, τις συναρτήσεις και τις παραμέτρους. Για κάθε ένα από τα παραπάνω υπάρχει και μία διαφορετική εγγραφή, όπου η πληροφορία αυτή είναι χρήσιμη για τον έλεγχο σφαλμάτων. Επίσης, ο πίνακας συμβόλων παρέχει την πληροφορία που απαιτείται για την σημασιολογική ανάλυση, ενώ μέρος της υλοποιείται μέσα στο πίνακα συμβόλων. Τέλος, όπως

φαίνεται και στο σχήμα παρακάτω ο πίνακας συμβόλων αντλεί πληροφορία από τις φάσεις της συντακτικής ανάλυσης και την παραγωγής ενδιάμεσου κώδικα. Η πληροφορία που διαθέτει αποτελεί την αρωγή στην παραγωγή του τελικού κώδικα. Για αυτό το λόγο η παραγωγή του τελικού κώδικα γίνεται παράλληλα με τις άλλες δύο φάσεις, θα συμπληρώνετε κάθε φορά πριν την διαγραφή ενός scope του πίνακα συμβόλων ώστε να έχουμε όλη την πληροφορία και τις εξαρτήσεις (γενεαλογικό δέντρο) που χρειαζόμαστε για την συμπλήρωση του τελικού κώδικα.



Εγγραφές στο πίνακα συμβόλων :

Ο πίνακας συμβόλων διατηρεί διαφορετική πληροφορία για κάθε είδος συμβολικού ονόματος. Σκοπός του είναι να συγκεντρώσει όλη την πληροφορία που θα απαιτηθεί μέχρι το τέλος της μεταγλώττισης. Η ανάλυση που θα γίνει παρακάτω αφορά τις ανάγκες της γλώσσας C-imple, για διαφορετική γλώσσα προγραμματισμού οι ανάγκες μπορεί να διαφέρουν. Πριν αναλύσουμε τους τύπους, θα ορίσουμε το εγγράφημα δραστηριοποίησης το οποίο είναι ο χώρος που δίνεται σε μια συνάρτηση ή διαδικασία για να τοποθετήσει τα δεδομένα της στην μνήμη. Θα ορίσουμε την κλάση Entity η οποία θα κρατήσει όλες τις διαφορετικές εγγραφές του πίνακα συμβόλων και θα τις ξεχωρίζει ανάλογα με το όνομα τους. Οποιαδήποτε άλλη πληροφορία χρειάζεται η κάθε εγγραφή θα αποθηκεύεται μέσα σε αυτή την κλάση.

Θα ξεκινήσουμε από τον πιο συχνά χρησιμοποιούμενο τύπο αυτόν της μεταβλητής(Variable). Κάθε μεταβλητή έχει το όνομα που την χαρακτηρίζει και τον τύπο της(θα είναι integer πάντα λόγο της C-imple).

Επίσης είναι σημαντικό για κάθε μεταβλητή να κρατάμε την θέση της στη μνήμη. Στο σημείο αυτό να σημειώσουμε πως είναι αδύνατο να προσδιορίσουμε με ακρίβεια τη θέση της στην φυσική μνήμη. Για αυτό το λόγο θα εκμεταλλευτούμε το πίνακα συμβόλων που ορίσαμε παραπάνω αλλά και το γεγονός ότι στην C-imple οι τύποι των μεταβλητών ορίζονται στην αρχή του προγράμματος ή ενός υπό προγράμματος και δεν μεταβάλλονται όσο διαρκεί η ζωή της μεταβλητής. Στον πίνακα συμβόλων θα αποθηκεύουμε την θέση της μεταβλητής μέσα στο εγγράφημα δραστηριοποίησης της συνάρτησης ή της διαδικασίας, την απόσταση δηλαδή από την αρχή του. Δηλαδή σαν entity εκτός από το όνομα(name) και το τύπο(datatype) της μεταβλητής θα κρατάμε και την απόσταση της μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης(offset).

Ένας άλλος τύπος εγγραφών στο πίνακα συμβόλων θα είναι η παράμετρος(Parameter). Θα είναι οι παράμετροι που περνάμε στις διαδικασίες και τις συναρτήσεις. Ο τρόπος αποθήκευσής τους στο πίνακα συμβόλων δεν έχει μεγάλες διαφορές σε σχέση με τις μεταβλητές, θα κρατάμε το όνομα, το τύπο, το offset και η διαφορά είναι πως θα κρατάμε και το τρόπο με τον οποίο περνάμε τις παραμέτρους. Οι τρόποι αυτοί για την C-imple είναι πέρασμα με τιμή, πέρασμα με αναφορά και πέρασμα για επιστροφή τιμής συνάρτησης, όπου θα αποθηκεύουμε τις τιμές CV, REF και RET αντίστοιχα στο πίνακα συμβόλων. Αυτή η πληροφορία θα μας χρειαστεί στο τελικό κώδικα. Για τα υπόλοιπα τρία πεδία ισχύει ότι αναφέραμε στις μεταβλητές.

Δύο άλλοι τύποι εγγραφής στο πίνακα συμβόλων είναι η συνάρτηση(function) και η διαδικασία(procedure). Αρχικά θα αποθηκεύουμε το όνομα των συναρτήσεων αυτών όπως αναφέραμε και στα παραπάνω. Επίσης, θέλουμε να κρατάμε τις τυπικές παραμέτρους που μπορεί να έχει η κάθε συνάρτηση ή διαδικασία, αλλά και την σειρά που εμφανίζονται, ώστε να γνωρίζουμε όταν καλούνται ποιες παραμέτρους έχουν το πεδίο αυτό θα ονομαστεί FormalParameters. Ένα ακόμη πεδίο που θα χρειαστεί είναι η ετικέτα της πρώτης εκτελέσιμης τετράδας της διαδικασίας ή συνάρτησης. Με αυτό το τρόπο θα ξέρουμε σε ποια τετράδα θα πρέπει να κάνει άλμα η

καλούσα συνάρτηση προκειμένου να εκκινήσει η εκτέλεση της κληθείσας. Το τελευταίο κοινό πεδίο θα είναι το framelength, το οποίο θα αποθηκεύει το μήκος (σε bytes) του εγγραφήματος δραστηριοποίησης της συνάρτησης ή της διαδικασίας. Πιο συγκεκριμένα, αυτό θα υπολογίζεται από το offset του τελευταίου entity που θα περιέχει το κάθε Scope προσθέτοντας 4 αφού η κάθε οντότητα αποθηκεύεται σε 4 byte. Η διαφοροποίηση της συνάρτησης είναι στο τελευταίο πεδίο, στο οποίο θα πρέπει να αποθηκεύουμε τον τύπο της συνάρτησης. Αυτό το πεδίο δεν είναι απαραίτητο καθώς λόγο της C-imple κάθε φορά ο τύπος δεδομένων της συνάρτησης θα έχει την τιμή integer, όμως θεωρούμε χρήσιμο να αναφερθεί αυτή η παρατήρηση και να συμπληρωθεί για την πληρότητα του μεταγλωττιστή.

Για το πεδίο formalParameter που αναφέραμε παραπάνω θα δημιουργήσουμε μια ξεχωριστή κλάση και θα την ονομάσουμε Argument() η οποία θα έχει σαν πεδία όλα αυτά που έχουν και οι παράμετροι. Η κλάση δημιουργείται ώστε για αυτές να κρατάμε τα πεδία της κάθε μίας παραμέτρου ξεχωριστά και να κρατάμε και την σειρά με την οποία διαβάστηκαν. Σημαντικό να αναφερθεί είναι το γεγονός πως ο πίνακας συμβόλων θα διαβάζει τις παραμέτρους αυτές στο scope που βρίσκεται ο πατέρας δηλαδή στην συνάρτηση που βρίσκεται έξω από αυτήν που περιέχει τις παραμέτρους αυτές. Αυτό συμβαίνει διότι η αριστερή αγκύλη ‘{‘ σε κάθε block βρίσκεται μετά το όρισμα των παραμέτρων της κάθε συνάρτησης. Για αυτό το λόγο συμπληρώνουμε τα πεδία αυτά στο επόμενο scope που θα έχουμε(το παιδί), δηλαδή μετά την αναγνώριση του συμβόλου της αριστερής αγκύλης ‘{’.

Τελευταία εγγραφή θα είναι η προσωρινή μεταβλητή (temporaryVariable), τα πεδία της θα είναι ίδια με αυτά της μεταβλητής (Variable). Η διαφοροποίησή τους γίνεται λόγο σχεδίασης και για λόγους πληρότητας.

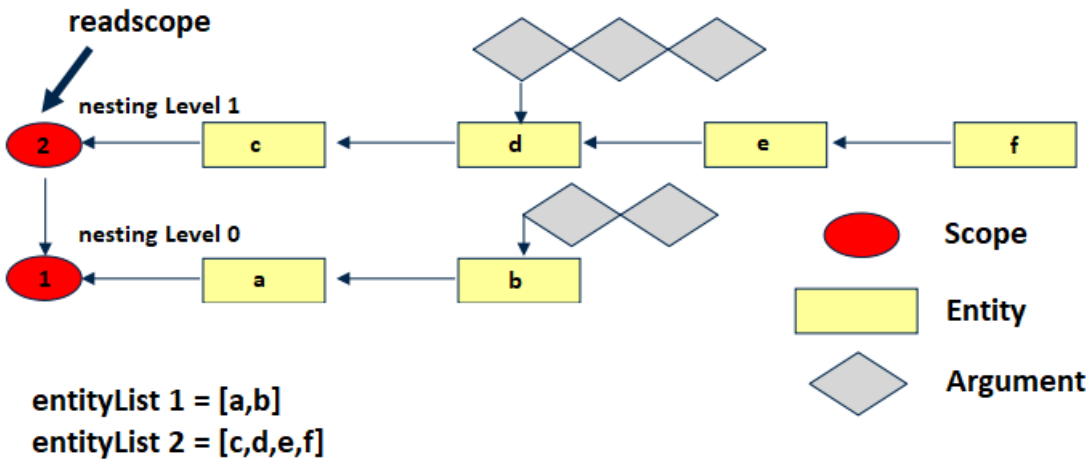
Για όλα τα παραπάνω πεδία(6 διαφορετικά πεδία σύνολο) έχουμε δημιουργήσει μία λίστα(array) η οποία θα υπάρχει σε κάθε κλάση Entity. Εκεί αναλόγως την εγγραφή που έχουμε θα συμπληρώνουμε μόνο τα πεδία που χρειαζόμαστε. Πιο συγκεκριμένα για το πεδίο

formalParameters έχουμε μια εσωτερική λίστα στην λίστα array για να αποθηκεύονται εσωτερικά.

Δομή του πίνακα συμβολών:

Ο πίνακας συμβολών αποτελείται από επίπεδα που ονομάζουμε Scopes. Κάθε Scope αντιστοιχεί στη μετάφραση μιας συνάρτησης και όταν ξεκινάμε να τη μεταφράζουμε φτιάχνουμε κατασκευάζουμε ένα τέτοιο επίπεδο. Όταν τελειώσει η μετάφραση της συνάρτησης το Scope αφαιρείται από τον πίνακα συμβολών. Επειδή στην C-imple μπορούμε να έχουμε και εμφωλιασμένες συναρτήσεις, μπορούμε στον πίνακα συμβολών να έχουμε και πολλαπλά επίπεδα για κάθε συνάρτηση και το 'παιδί' της που μεταφράζονται κάθε στιγμή.

Δημιουργούμε την κλάση Scope την οποία θα αναγνωρίζουμε μέσω του ονόματος που θα παίρνει το οποίο θα είναι μοναδικό, αυτό συμβαίνει διότι θα έχει το όνομα της κάθε συνάρτησης ή διαδικασίας. Επίσης για κάθε Scope είναι απαραίτητο να γνωρίζουμε τα entities τα οποία υπάρχουν-βλέπει, ώστε να ξέρουμε ποιες μεταβλητές, παράμετροι, αρχικοποιήσεις μεταβλητών είναι ορατές σε κάθε συνάρτηση. Τα Entities τα βάζουμε σε μια λίστα για να έχουμε πρόσβαση όποτε χρειαστεί. Επιπροσθέτως, για να γνωρίζουμε σωστά τις εμβέλειες, πρέπει να γνωρίζουμε το βάθος που βρίσκεται κάθε νέο επίπεδο(Scope) που δημιουργούμε, ορίζουμε πως για βάθος = 0 σημαίνει πως είναι ο μεγαλύτερος πρόγονος και όσο αυξάνεται το βάθος πηγαίνουμε στα παιδιά. Ορίσαμε το βάθος = 0 για τον μεγαλύτερο πρόγονο γιατί η main θα είναι η τελευταία που θα εκτελεστεί στο πρόγραμμα, άρα θα βγει από το πίνακα συμβόλων τελευταία. Τέλος, αποθηκεύουμε στην κλάση Scope το πατέρα κάθε επιπέδου(scope) για να γνωρίζουμε ποια συνάρτηση βρίσκεται εξωτερικά από αυτήν που εκτελείται. Με αυτό το τρόπο γνωρίζουμε ποια συνάρτηση θα εκτελεστεί μετά από αυτήν που βρισκόμαστε και ποιες μεταβλητές είναι ορατές από την συνάρτηση που εκτελείται.



Εγγράφημα Δραστηριοποίησης:

Ένα εγγράφημα δραστηριοποίησης δημιουργείται για κάθε συνάρτηση που πρόκειται να κληθεί και καταστρέφεται με την ολοκλήρωση της εκτέλεσης της συνάρτησης. Πιο συγκεκριμένα αποτελεί τον χώρο που δεσμεύει η συνάρτηση στη στοίβα για την αποθήκευση πληροφοριών χρήσιμων για την εκτέλεση της όπως πραγματικές παραμέτρους, τοπικές και προσωρινές μεταβλητές. Η δομή του εγγραφήματος δραστηριοποίησης έχει στην πρώτη θέση τη διεύθυνση επιστροφής της συνάρτησης. Το μέγεθος είναι τόσο ώστε να χωράει μια διεύθυνση στη μνήμη και επειδή στη C-imple χρησιμοποιούμε μόνο ακέραιους που η διεύθυνση τους στη μνήμη πιάνει σταθερά 4 bytes, μπορούμε να ξέρουμε και το μέγεθος της διεύθυνσης επιστροφής. Στη δεύτερη θέση του εγγραφήματος δραστηριοποίησης τοποθετούμε τον σύνδεσμο προσπέλασης που είναι ένας δείκτης που δείχνει σε ένα άλλο εγγράφημα δραστηριοποίησης. Σε εκείνο το εγγράφημα είναι που πρέπει να γίνει η αναζήτηση μιας μεταβλητής ή μιας παραμέτρου που δεν έχει αρχικοποιηθεί στη συνάρτηση που εξετάζουμε αλλά σύμφωνα με την εμβέλεια των συναρτήσεων πρέπει να έχουμε πρόσβαση. Αναδρομικά όσο δεν βρίσκουμε μια πληροφορία στο εγγράφημα δραστηριοποίησης το οποίο κοιτάμε, πάμε στην διεύθυνση που δείχνει ο σύνδεσμος προσπέλασης και ψάχνουμε το επόμενο αν υπάρχει. Στην τρίτη θέση του εγγραφήματος δραστηριοποίησης γίνεται δέσμευση χώρου για την επιστροφή τιμής της συνάρτησης. Εκεί θα γίνει η αποθήκευση της διεύθυνσης της μεταβλητής στην οποία θα επιστραφεί η τιμή της συνάρτησης. Στην περίπτωση που το εγγράφημα δραστηριοποίησης αφορά διαδικασία, η θέση μένει κενή. Οι τρεις αυτές βασικές θέσεις για ένα εγγράφημα δραστηριοποίησης σχηματίζουν ένα

κομμάτι μεγέθους 12 bytes που κάθε εγγράφημα δραστηριοποίησης θα έχει στην αρχή του. Στη συνέχεια του εγγραφήματος τοποθετούμε παραμέτρους και μεταβλητές που έχει η συνάρτηση. Κάθε παράμετρος χρειάζεται χώρο 4 bytes, εμφανίζεται στο εγγράφημα στη σειρά που έχει δηλωθεί και ανάλογα με το αν είναι περασμένη με τιμή ή με αναφορά, θα τοποθετηθεί εκεί η τιμή ή η διεύθυνση της. Η δέσμευση χώρου για τις τοπικές μεταβλητές και τις προσωρινές μεταβλητές δουλεύει με τον ίδιο τρόπο με τις παραμέτρους. Κάθε εγγράφημα δραστηριοποίησης χαρακτηρίζεται από το μέγεθος του σε Bytes (framelength). Το μέγεθος αυτό υπολογίζεται ευκολά και γρήγορα γιατί ξέρουμε το μέγεθος της κεφάλης του εγγραφήματος που είναι σταθερά 12 bytes, τον αριθμό των παραμέτρων, των μεταβλητών και των προσωρινών μεταβλητών μιας συνάρτησης ή διαδικασίας. Το επόμενο σημαντικό χαρακτηριστικό ενός εγγραφήματος δραστηριοποίησης είναι η απόσταση μιας παραμέτρου ή μεταβλητής από την αρχή του (offset). Η απόσταση μετριέται σε bytes και περιλαμβάνει τα 12 πρώτα bytes του εγγραφήματος.

Λειτουργίες του πίνακα συμβολών:

Η λειτουργία του πίνακα συμβολών βασίζεται στη δυνατότητα της προσθαφαίρεσης πληροφορίας. Αυτό σημαίνει ότι στον πρέπει να έχουμε πρόσβαση στην πληροφορία που χρησιμοποιούμε ανά δεδομένη χρονική στιγμή δηλαδή στις συναρτήσεις που εκτελούνται και στους προγόνους τους και όχι σε αυτές που η εκτέλεση τους έχει τερματιστεί και η πληροφορία που έχουν μας είναι πλέον άχρηστη.

Κάθε φορά που γίνεται δήλωση μιας μεταβλητής, ονόματος συνάρτησης ή προσωρινής μεταβλητής, πρέπει να φτιάχνουμε ένα entity με τα αντίστοιχα χαρακτηριστικά και να του περνάμε την πληροφορία του. Η πληροφορία αυτή, ανάλογα την οντότητα που αναγνωρίστηκε, θα περιέχει τον τύπο της (μόνο ακέραιους(int) έχουμε στην C-imple), την τιμή της, την απόσταση της από την αρχή του εγγραφήματος δραστηριοποίησης(offset), τον τρόπο με τον οποίο περάστηκε στις παραμέτρους(με τιμή CV ή με αναφορά REF), την ετικέτα της πρώτης τετράδας(για συναρτήσεις ή διαδικασίες), το μέγεθος του Scope(framelength για συναρτήσεις ή διαδικασίες) και τις τυπικές παραμέτρους (για συναρτήσεις ή διαδικασίες). Η καινούρια

οντότητα προστίθεται στην τελευταία θέση της λίστας με τα Entities του Scope το οποίο διαβάζουμε εκείνη τη στιγμή.

Κάθε φορά που στη ροή ανάγνωσης(μετάφρασης) του προγράμματος συναντάμε μια νέα συνάρτηση ή διαδικασία πρέπει να κάνουμε ένα νέο Scope που θα την αντιπροσωπεύει. Για αυτό το Scope γίνεται αποθήκευση του ονόματος του, του επιπέδου φωλιάσματος στο οποίο βρίσκεται και του άμεσου προγόνου του (πατέρας).

Όταν τελειώνει το block μιας συνάρτησης ή διαδικασίας και έχουμε ολοκληρώσει την μετάφραση της πρέπει να το αφαιρέσουμε από τη "στοίβα" με τα Scopes και να διαγράψουμε τη σχετική πληροφορία. Αυτό γίνεται κάνοντας τον πρόγονο του Scope που διαβάζουμε ανώτερο Scope και στη συνέχεια διαγράφοντας το προηγούμενο Scope μαζί με τις εγγραφές του.

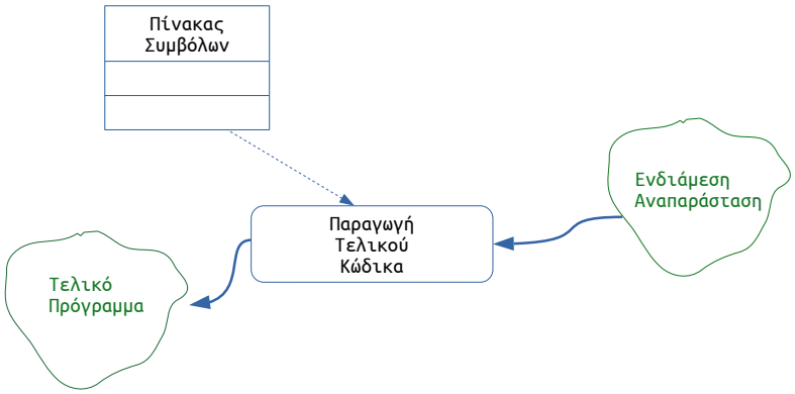
Όταν έχει μεταφραστεί ένα ολόκληρο block μιας συνάρτησης ή μιας διαδικασίας πρέπει να περάσουμε στο entity της το μέγεθος της σε bytes(framelength). Αυτό γίνεται βρίσκοντας το framelength της τελευταίας εγγραφής και προσθέτοντας 4 που είναι το default μέγεθος για την C-imple. Επίσης όταν έχει μεταφραστεί η πρώτη εκτελέσιμη εντολή του block μπορούμε να ξέρουμε και τον αριθμό της ετικέτας της πρώτης τετράδας μιας συνάρτησης ή διαδικασίας.

Κάθε φορά που συναντάμε μια τυπική παράμετρο(argument) πρέπει να δημιουργούμε ένα αντικείμενο τύπου Argument. Όταν ορίζεται μια συνάρτηση ή διαδικασία, τα ορίσματα που της περνάμε πρέπει να περαστούν σαν παράμετροι στο entity της συνάρτησης ή διαδικασίας που ανήκουν ώστε στο τέλος να μεταφερθούν στο Scope με βάθος 0(πρώτο). Το αντικείμενο αυτό θα περιέχει πληροφορίες για το argument όπως το όνομα και ο τρόπος περάσματος και στη συνέχεια θα αποθηκεύεται στη λίστα με τις τυπικές παραμέτρους στο entity.

Για τον πίνακα συμβόλων δημιουργούμε το αρχείο .symb στο οποίο αποθηκεύουμε τον πίνακα συμβόλων βήμα-βήμα, δηλαδή φαίνονται τα scope που θα έχουμε κάθε στιγμή και τα entities τους. Το επόμενο στιγμιότυπο θα είναι όταν διαγραφεί ένα scope.

Τελικός κώδικας

Ο τελικός κώδικας αποτελεί την Πέμπτη και τελευταία φάση του μεταγλωττιστή. Η φάση αυτή θα χρησιμοποιήσει σαν είσοδο τον ενδιάμεσο πίνακα και τον πίνακα συμβόλων. Πιο συγκεκριμένα, από κάθε εντολή ενδιάμεσου κώδικα προκύπτει μία σειρά εντολών του τελικού κώδικα, η οποία για να παραχθεί ανακτά πληροφορίες από τον πίνακα συμβόλων. Για την C-imple ο τελικός κώδικας που θα παράγουμε θα είναι σε συμβολική γλώσσα μηχανής (assembly code) του επεξεργαστή RISC-V.



Παραθέτουμε παρακάτω κάποιες από τις εντολές της assembly του RISC-V που χρησιμοποιούμε :

Ο επεξεργαστής RISC-V για τον οποίο γράφεται ο τελικός κώδικας, έχει 32 καταχωρητές για χρήση σε ακέραιους αριθμούς. Ξεκινάνε με x και ακολουθούνται από έναν αριθμό από το 0-32 πχ x0 κλπ. Κάποιοι λόγω των λειτουργιών που επιτελούν ξεφεύγουν από αυτό τον τρόπο ονοματοδοσίας για να παραπέμπουν εκεί. Ο καταχωρητής zero βρίσκεται μόνιμα στην τιμή 0 λόγω της συχνής χρήσης του αριθμού, και για ελέγχους. Ο καταχωρητής sp(stack pointer) χρησιμοποιείται για να κρατά τη διεύθυνση της στοίβας στην αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης ή της διαδικασίας που κάθε στιγμή εκτελείται. Ο καταχωρητής fp χρησιμοποιείται για να δείχνει την αρχή ενός εγγραφήματος δραστηριοποίησης που φτιάχνεται εκείνη τη στιγμή. Οι καταχωρητές t0-t6 χρησιμοποιούνται για οποιοδήποτε σκοπό συνήθως όμως τους χρησιμοποιούμε σαν προσωρινούς καταχωρητές, γι’ αυτό και χρησιμοποιούμε το t από το temporary στην αρχή του

ονόματος τους. Οι καταχωρητές s1-s11 χρησιμοποιούνται για σημαντικές τιμές που πρέπει να διατηρηθούν ανάμεσα σε κλήσεις συναρτήσεων και διαδικασιών. Οι καταχωρητές a0-a7 χρησιμοποιούνται για πέρασμα παραμέτρων ανάμεσα σε συναρτήσεις ή διαδικασίες και σε άλλες περιπτώσεις για επιστροφή τιμών. Ο καταχωρητής ra χρησιμοποιείται για αποθήκευση της διεύθυνσης στην οποία πρέπει να επιστρέψει ο έλεγχος μετά από κλήση συνάρτησης ή διαδικασίας. Ο καταχωρητής pc περιέχει τη διεύθυνση της εντολής που εκτελείται. Ο καταχωρητής gp χρησιμοποιείται για να κρατάει τη διεύθυνση της αρχής των μεταβλητών που είναι καθολικές σε ένα πρόγραμμα.

Εντολές στον RISC-V

Με την εντολή li περνάμε σε ένα καταχωρητή μια αριθμητική τιμή πχ li t0,2. Με την εντολή mv γίνεται μεταφορά της τιμής από έναν καταχωρητή σε έναν άλλο πχ mv reg1, reg2. Με την εντολή add γίνεται πρόσθεση των τιμών δυο καταχωρητών και το αποτέλεσμα μπαίνει σε έναν άλλο(μπορεί να είναι και ένας από τους 2). Αντίστοιχα αφαίρεση γίνεται με την εντολή sub, πολλαπλασιασμός με την εντολή mul και διαίρεση με την εντολή div. Για να κάνουμε πρόσθεση μιας ακέραιας τιμής στην τιμή ενός καταχωρητή χρησιμοποιούμε την εντολή addi αποθηκεύοντας την νέα τιμή σε έναν καταχωρητή.

Για πρόσβαση στη μνήμη χρησιμοποιούνται οι lw και sw με το l να συμβολίζει την ανάγνωση και το s την εγγραφή. Η πρόσβαση στη μνήμη γίνεται μέσω ενός καταχωρητή, συνήθως τον sp ή τον fp. Η έκφραση lw reg1, offset(reg2) παίρνει την τιμή που υπάρχει στη θέση μνήμης του reg2 + offset και την περνάει στον καταχωρητή reg1. Η έκφραση sw reg1, offset(reg2) πάει στη θέση μνήμης του reg2 + offset και γράφει την τιμή που έχει ο reg1. Όταν στη θέση του offset δεν υπάρχει τίποτα, εννοείται το μηδέν.

Για τις διακλαδώσεις χρησιμοποιείται το άλμα (jump) με διάφορους τρόπους. Ο πρώτος είναι το j label όπου label είναι μια υπάρχουσα ετικέτα. Αυτή η εντολή πάει τη ροή εκτέλεσης στην ετικέτα label. Αλλαγή της ροής του προγράμματος μπορούμε να έχουμε και στις εντολές άλματος υπό συνθήκη. Ο τρόπος που γράφονται είναι beq reg1, reg2, label όπου σ αυτή την περίπτωση γίνεται έλεγχος αν οι

τιμές στους reg1 και reg2 είναι ίσες. Αν η σύγκριση ισχύει τότε γίνεται άλμα στην ετικέτα label αλλιώς συνεχίζει η ροή του κώδικα κανονικά. Στη θέση του beq μπορούμε να έχουμε κ άλλους ελέγχους όπως το bne για έλεγχο ανισότητας, το blt για να ελέγξουμε αν το $1^{ο}$ είναι μικρότερο από το $2^{ο}$, αντίστοιχα το bgt για μεγαλύτερο, το ble για έλεγχο αν το $1^{ο}$ είναι μικρότερο ή ίσο από το $2^{ο}$ και το bge αντίστοιχα για έλεγχο αν είναι μεγαλύτερο ή ίσο. Άλμα γίνεται και με την εντολή jr reg όπου αν στον καταχωρητή reg είναι αποθηκευμένη μια διεύθυνση, κάνουμε άλμα στη διεύθυνση αυτή. Για την κλήση μιας συνάρτησης ή διαδικασίας πρέπει να γίνει ένα διαφορετικό άλμα καθώς πρέπει να τοποθετηθεί στον καταχωρητή ra η διεύθυνση της εντολής μετά το άλμα. Αυτό γίνεται με την εντολή jal L1 που γίνεται άλμα στην ετικέτα L1.

Για να γίνει είσοδος δεδομένων από το πληκτρολόγιο χρησιμοποιούνται οι καταχωρητές a0 και a7. Τοποθετούνται τα ορίσματα στους καταχωρητές και καλείται η εντολή ecall για να διαβαστεί η είσοδος από το πληκτρολόγιο. Βάζοντας στον a7 την τιμή 5 δηλώνουμε ότι θέλουμε να διαβαστεί ένας ακέραιος αριθμός και να μπει στον a0. Για εμφάνιση ενός ακέραιου στην οθόνη χρησιμοποιούμε τους ίδιους καταχωρητές και βάζουμε στον a7 την τιμή 1 και στον a0 αυτόν που θέλουμε να εμφανίσουμε αι καλούμε την ecall. Για να γίνει αλλαγή γραμμής στην τύπωση πρέπει να οριστεί στο asm αρχείο του τελικού κώδικα η κεφαλίδα .data και από κάτω να οριστεί ένα όνομα για την αλλαγή γραμμής που σε asciz είναι ο χαρακτήρας “\n” όπως φαίνεται παρακάτω .data

str_nl: .asciz “\n”

Στη συνέχεια τοποθετούμε στον καταχωρητή a7 την τιμή 4 και στον a0 το συμβολικό όνομα str_nl καλώντας στη συνέχεια την ecall.

Για να τερματίσουμε το πρόγραμμα χρησιμοποιούμε πάλι τους καταχωρητές a0 και a7, στον a7 βάζουμε την τιμή 93 και στον a0 αυτό που θέλουμε να επιστραφεί στο λειτουργικό ως αποτέλεσμα. Στη συνέχεια καλούμε την ecall.

Βοηθητικές συναρτήσεις :

Οι βοηθητικές συναρτήσεις στο τελικό κώδικα επιτελούν τον ίδιο σκοπό με αυτές στον ενδιάμεσο κώδικα.

Η βοηθητική συνάρτηση gnlvcode() παράγει τελικό κώδικα για την προσπέλαση μεταβλητών ή διευθύνσεων που είναι αποθηκευμένες σε κάποιο εγγράφημα δραστηριοποίησης προγόνου. Σαν ορίσματα παίρνει μια μεταβλητή(x) της οποίας την τιμή ή την διεύθυνση θέλουμε να προσπελάσουμε. Στόχος της συνάρτησης είναι η αναζήτηση στο πίνακα συμβόλων της μεταβλητής που παίρνει σαν παράμετρο. Αν δεν βρεθεί εκεί, τότε θα υπάρχει το κατάλληλο μήνυμα σφάλματος. Αν βρεθεί τότε πρέπει να κρατήσει πόσα επίπεδα ανέβηκε προκειμένου να φτάσει στο εγγράφημα δραστηριοποίησης που έχει την μεταβλητή αυτή. Σημειώνουμε πως στην υλοποίησή μας θα κατεβαίνουμε scope κάθε φορά αναζητώντας την μεταβλητή. Η λειτουργία της θα είναι να εκτελεί επαναληπτικά όλα τα παραπάνω μέχρι να βρει την μεταβλητή στο πίνακα συμβόλων, δηλαδή να διαπεράσει όσα entity χρειάζεται. Όταν βρεθεί, πρώτο βήμα θα είναι να μεταβεί στο γονέα της συνάρτησης (αυτό γίνεται διότι για να βρισκόμαστε στην gnlvcode() σίγουρα αναζητούμε την πληροφορία σε κάποιο πρόγονο). Για την υλοποίηση αυτού εκμεταλλευόμαστε πως ο sp δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης κάθε στιγμή, άρα αρκεί να πάμε στη θέση -4(sp). Θα χρησιμοποιήσουμε έναν προσωρινό καταχωρητή για να ανέβουμε τα επίπεδα έστω t0 και την εντολή lw. Αν δεν βρεθεί στο πρώτο πρόγονο(πατέρα), τότε θα πρέπει να θυμόμαστε πως όσα επίπεδα υπολογίσουμε πως πρέπει να ανέβουμε στο εγγράφημα δραστηριοποίησης να μειώσουμε κατά ένα (διότι το συμπληρώνουμε by default του πατέρα κάθε φορά). Οπότε θα συμπληρώσει όσα lw χρειάζονται ακόμα. Τέλος, μετά την ολοκλήρωση των μεταβάσεων ο καταχωρητής t0 θα δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης και θέλουμε να κατέβει κατά offset θέσεις. Αυτό γίνεται για να δείξει στη θέση μνήμης που βρίσκεται η πληροφορία που αναζητούμε, το offset που χρησιμοποιούμε υπάρχει για την μεταβλητή αυτή στο πίνακα συμβόλων. Το αποτέλεσμα θα υπάρχει στον t0

Η βοηθητική συνάρτηση loadvr() παράγει τελικό κώδικα για να διαβαστεί η τιμή μιας μεταβλητής από την θέση της στοίβας(μνήμη) και να μεταφερθεί σε έναν καταχωρητή. Σαν ορίσματα θα έχει το όνομα της μεταβλητής(v) που θα διαβαστεί και το όνομα του καταχωρητή(reg) στον οποίο θα τοποθετηθεί. Συμπεραίνουμε πως η βοηθητική συνάρτηση βοηθητική συνάρτηση βασίζεται στη πληροφορία που της

επιστρέφει ο πίνακας συμβόλων, για αυτό το λόγο εξετάζουμε το κάθε κώδικα που παράγεται ξεχωριστά. Θα πρέπει δηλαδή να αναγνωρίζουμε πρώτα κάποια γνωρίσματα της μεταβλητής και έπειτα να συμπληρώνουμε τον κατάλληλο κώδικα. Για την υλοποίηση αυτή θα χρησιμοποιήσουμε την εντολή lw της assembly η οποία κάνει ανάγνωση.

α) τοπική μεταβλητή ή παράμετρο που έχει περαστεί με τιμή ή προσωρινή μεταβλητή :
Σε αυτές τις περιπτώσεις η τιμή της μεταβλητής βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης της συνάρτησης που μεταφράζεται, για αυτό το λόγο θα πρέπει να μεταβούμε όσο το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων. Με αυτό το τρόπο θα μεταφερθεί η μεταβλητή στο καταχωρητή reg σωστά.

```
lw reg,-offset(sp)
```

β) παράμετρος που έχει περαστεί με αναφορά :
Σε αυτή την περίπτωση η παράμετρος στο εγγράφημα δραστηριοποίησης της συνάρτησης έχει τοποθετήσει τη διεύθυνσή της παραμέτρου. Θα ακολουθήσουμε το ίδιο βήμα που κάναμε παραπάνω όμως πρώτα πρέπει να μεταφέρουμε τη διεύθυνση της μεταβλητής από τη στοίβα σε έναν καταχωρητή, ώστε να μπορεί να γίνει η εντολή που περιγράψαμε παραπάνω.

```
lw t0,-offset(sp)
lw reg,(t0)
```

γ) τοπική μεταβλητή ή παράμετρο που έχει περαστεί με τιμή ή προσωρινή μεταβλητή η οποία θα ανήκει σε πρόγονο:
Η διαφορά είναι πως η τιμή της μεταβλητής θα είναι αποθηκευμένη στο εγγράφημα δραστηριοποίησης κάποιας συνάρτησης προγόνου. Αυτό σημαίνει πως θα πρέπει να ανακτήσουμε συνδέσμους προσπέλασης για να βρούμε την πληροφορία. Εδώ θα θέλαμε να επισημάνουμε πως όλα τα εγγραφήματα δραστηριοποίησης των προγόνων βρίσκονται στη στοίβα, αφού η εκτέλεση τους θα γίνει μετά την εκτέλεση του παιδιού. Αρχικά θα χρησιμοποιήσουμε την gnlvcode() με παράμετρο τη μεταβλητή που θέλουμε να διαβάσουμε, ώστε να τοποθετηθεί στο t0 η διεύθυνση της μεταβλητής που αναζητάμε. Στη συνέχεια μένει απλά να διαβάσουμε το περιεχόμενο της θέσης μνήμης. Αυτό θα γίνει περνώντας τον καταχωρητή t0 που αποθηκεύσαμε την θέση μνήμης

στον καταχωρητή που θέλουμε να περάσει η πληροφορία τον reg.

```
gnlvcode()
lw reg ,(t0)
```

δ) παράμετρος που έχει περαστεί με αναφορά η οποία θα ανήκει σε πρόγονο :
Η διαφορά είναι πως η διεύθυνση της μεταβλητής θα είναι αποθηκευμένη στο εγγράφημα δραστηριοποίησης του προγόνου. Η διαδικασία θα είναι η ίδια με παραπάνω δηλαδή θα καλέσουμε την βοηθητική συνάρτηση gnlvcode(), όπως ακριβώς αναφέρθηκε. Επίσης όπως ακριβώς αναφέρθηκε και στο πέρασμα με αναφορά χωρίς προγόνου θα πάρουμε πρώτα την διεύθυνση της μεταβλητής από τη στοίβα σε έναν καταχωρητή και μετά θα περάσουμε στο reg καταχωρητή.

```
gnlvcode()
lw t0,(t0)
lw reg ,(t0)
```

ε) καθολική μεταβλητή :

Στην περίπτωση αυτή η μεταβλητή έχει δηλωθεί στο κυρίως πρόγραμμα δηλαδή στη main. Επειδή η πρόσβαση σε καθολικές μεταβλητές είναι αρκετά συχνή σε όλες τις γλώσσες δεν θα χρησιμοποιήσουμε το gnlvcode() μέχρι να φτάσουμε στην main διότι θα ήταν χρονοβόρο. Για αυτό το λόγο θα αφιερώσουμε έναν καταχωρητή που θα σημειώνει μόνο την αρχή του εγγραφήματος δραστηριοποίησης του κυρίως προγράμματος, θα τον ονομάσουμε (gp: global pointer). Με αυτό το τρόπο αρκεί να πάμε offset(της μεταβλητής) θέσεις τον καταχωρητή gp για να και να αποθηκεύσουμε την επιθυμητή τιμή στο καταχωρητή reg. Για την αναγώριση πως μία μεταβλητή είναι δηλωμένη στο κυρίως πρόγραμμα θα εκμεταλλευτούμε το πίνακα συμβόλων στο οποίο κρατάμε πως η main θα βρίσκεται συνεχώς σε βάθος = 0.

```
lw reg,-offset(gp)
```

ζ) εκχώρηση αριθμητικής σταθεράς :

Στην περίπτωση αυτή η μεταβλητή είναι σταθερά αλλά δεν θέλουμε να την μετακινήσουμε στη μνήμη αλλά να την φορτώσουμε στο

καταχωρητή reg. Θυμίζουμε πως έχουμε μόνο ακέραιες σταθερές στη C-imple.

```
li reg, integer
```

Η βοηθητική συνάρτηση storevr() θα έχει την αντίστοιχη υλοποίηση με την βοηθητική συνάρτηση loadvr(). Η διαφοροποίηση θα είναι στις τελευταίες εντολές που παράγει η κάθε κλήση της storerv(), διότι αντί για εντολή ανάγνωσης (lw) θα χρησιμοποιεί την εντολή αποθήκευσης (sw). Οι περιπτώσεις θα είναι οι ίδιες που αναφέρθηκαν παραπάνω και οι μηχανισμοί που χρησιμοποιήθηκαν για την μνήμη θα είναι όμοιοι.

Στο σημείο αυτό θα δημιουργείται μία συνάρτηση η οποία θα γράφει στο αρχείο .asm που ανοίξαμε ώστε να δημιουργηθεί ο assembly κώδικας, η συνάρτηση αυτή ονομάστηκε assemblyCode(). Η συνάρτηση αυτή καλείται πριν διαγράψουμε ένα scope από τον πίνακα συμβόλων. Η συνάρτηση θα τρέχει επαναληπτικά ανάλογα με το πόσες τετράδες έχουν δημιουργηθεί για το scope το οποίο θα διαγραφεί. Για αυτές τις τετράδες θα ελέγχουμε το τελεστή τους ώστε να γράψουμε τον αντίστοιχο κώδικα που χρειάζεται κάθε εντολή για να αναπαρασταθεί στην assembly. Για κάθε τετράδα, στην αρχή σημειώνουμε την ετικέτα της, διότι θα πρέπει να συμπληρωθεί σαν label στο .asm αρχείο ώστε να γνωρίζει η assembly για κάθε εντολή ποιο label θα πρέπει να καλεστεί.

Εκχώρηση :

Μια εντολή του ενδιάμεσου κώδικα που αντιστοιχεί στη εκχώρηση της τιμής μιας μεταβλητής σε μια άλλη είναι η :=, x, _, z. Αυτή στον τελικό κώδικα μεταφράζεται ως z:=x. Για να κάνουμε αυτή την ανάθεση πρέπει πρώτα να φορτωθεί η μεταβλητή x σε έναν καταχωρητή και να μεταφερθεί στη θέση μνήμης που βρίσκεται η μεταβλητή. Για να παραχθεί ο ζητούμενος τελικός κώδικας για την ανάθεση καλούμε την loadvr(x, t0) και στη συνέχεια την storerv(t0, z) (ο t0 μπορεί να είναι οποιοσδήποτε καταχωρητής). Αν στη μεταβλητή z θέλουμε να περάσουμε αριθμητική σταθερά πχ το 8, τότε κάνουμε loadvr(8, t0) και στη συνέχεια storerv(t0, z).

Αριθμητικές πράξεις :

Για να γίνει μια αριθμητική πράξη πχ +, x, y, z ανάλογα τον τελεστή που θα διαβάσουμε (για το παράδειγμα +) κάνουμε φόρτωση των μεταβλητών x και y σε καταχωρητές με loadvr(x, t1) και loadvr(y, t2), κάνουμε πρόσθεση των καταχωρητών με produce('add t1, t2, t1') (η πράξη γίνεται σε επίπεδο εκτέλεσης) και το αποτέλεσμα το μεταφέρουμε στη μεταβλητή της έκφρασης με storerv(t1, z).

Άλμα:

Οι ετικέτες του ενδιάμεσου κώδικα γίνονται label για τον τελικό για να μπορούμε να κάνουμε αλλαγή ροής εκτέλεσης του κώδικα οπότε είναι απαραίτητο. Τα label που κάνουμε είναι της μορφής L34 όπου στη θέση του 34 πάει ο πρώτος αριθμός από τις πεντάδες που φτιάχτηκαν στον ενδιάμεσο κώδικα(ετικέτα). Για να γίνει ένα άλμα σε μια ένα label και να αλλάξει η ροή εκτέλεσης του κώδικα χρειάζεται να ξέρουμε μόνο την ετικέτα στην οποία θέλουμε να πάμε. Η εντολή που χρησιμοποιούμε είναι η j L34.

Διακλαδώσεις :

Για σε τελικό κώδικα ένα άλμα υπό συνθήκη πρέπει να εντοπίσουμε έναν από τους παρακάτω ελέγχους. Τον έλεγχο ισότητας(=), τον έλεγχο διαφοράς(<>), έναν από τους ελέγχους σύγκρισης μεγαλύτερο, μικρότερο, μεγαλύτερο ή ίσο και μικρότερο ή ίσο (>, <, >=, <=). Για να μεταφραστεί σε assembly κώδικα ένας τέτοιος έλεγχος μεταφέρουμε σε καταχωρητές (πχ t1 και t2) τις εμπλεκόμενες μεταβλητές που πήραμε από τις αποθηκευμένες τετράδες και φτιάχνουμε σε assembly τη φράση beq t1, t2, L34 όπου beq είναι ο έλεγχος που κάνουμε.

Τερματισμός κυρίως πρόγραμμα :

Για να τερματίσει το πρόγραμμα πρέπει να διαβάσουμε τη δεσμευμένη λέξη halt σε μια τετράδα του ενδιάμεσου κώδικα. Όταν γίνει αυτό ακολουθούμε την διαδικασία για τερματισμό του προγράμματος όπως είπαμε παραπάνω.

Τερματισμός συναρτήσεων και προγράμματος :

Όταν έχουμε τετράδα για κλείσιμο block συνάρτησης ή διαδικασίας διαβάζουμε από τον sp την τιμή του και την περνάμε στον ra ώστε να γυρίσουμε πίσω σε αυτόν που μας κάλεσε.

Αρχή προγράμματος :

Όταν έχουμε άνοιγμα block για μια συνάρτηση ή διαδικασία πρέπει να βάλουμε στη θέση μνήμης του sp την διεύθυνση του ra ώστε όταν τη χρειαστούμε να μπορούμε να την πάρουμε για να γυρίσουμε πίσω σ αυτόν που μας κάλεσε στο κλείσιμο του block. Αυτό το κάνουμε φτιάχνοντας την εντολή sw ra, (sp) σε assembly.
Όταν έχουμε άνοιγμα block για το κυρίως πρόγραμμα πρέπει πρώτα να ορίσουμε με ένα label ότι πρόκειται για τη main. Στη συνέχεια πρέπει να μετακινήσουμε τον sp όσο είναι το framelength της main (το οποίο παίρνουμε από τον πίνακα συμβολών) για να δείχνει στο σωστό σημείο με addi sp, sp, (main framelength) και να στον gp την τιμή του sp για να γνωρίζουμε που ορίζονται οι global μεταβλητές.

Επιστροφή τιμής Συνάρτησης:

Τις τιμές που επιστρέφουν οι συναρτήσεις τις παίρνουμε πριν κλείσει το block τους όταν βλέπουμε σε μια από τις τετράδες του ενδιάμεσου κώδικα το retv. Φορτώνουμε την τιμή που εχει η μεταβλητή σε ένα καταχωρητή με την loadvr, μετακινούμε τον t0 να δείχνει στην μεταβλητή που θα αποθηκευτεί το αποτέλεσμα που επιστρέφει η συνάρτηση με την lw t0, -8(sp) και αποθηκεύουμε την τιμή με
sw t1, (t0).

Input:

Όταν βλέπει το πρόγραμμα το inp στην τετράδα του ενδιάμεσου κώδικα που διαβάζει γίνεται η διαδικασία που αναφέραμε παραπάνω.

Print:

Όταν βλέπει το πρόγραμμα το out πρώτα τυπώνεται η τιμή που πρέπει και στη συνέχεια η αλλαγή γραμμής σε περίπτωση που χρειαστεί να ξανατυπώσουμε.

Πέρασμα παραμέτρων με τιμή :

Όταν αναγνωρίζετε, το πέρασμα παραμέτρου με τιμή, αναζητάμε αρχικά την τιμή και την τοποθετούμε προσωρινά σε έναν καταχωρητή. Αυτό θα υλοποιηθεί καλώντας την βοηθητική συνάρτηση loadvr(), η οποία θα αντλήσει την απαιτούμενη αυτή πληροφορία όπως έχει ήδη

αναλυθεί. Στην συνέχεια εφόσον έχουμε τοποθετήσει την τιμή της ζητούμενης παραμέτρου στο καταχωρητή θέλουμε να αντιγραφή η τιμή της παραμέτρου από τον t0 στην κατάλληλη θέση στη στοίβα. Για να υπολογίσουμε σωστά την θέση στην στοίβα πρέπει να υπολογίσουμε τον χώρο που έχουν καταλάβει οι παράμετροι που έχουν ήδη τοποθετηθεί εκεί. Αυτό θα γίνει μέσω του τύπου d=12+(i-1)*4, 12 είναι τα πρώτα bytes του εγγραφήματος δραστηριοποίησης που αποθηκεύεται η πληροφορία για την λειτουργία της συνάρτησης. Τέλος, θέλουμε η εντολή να αντιγράψει την τιμή της παραμέτρου από τον καταχωρητή t0 στην κατάλληλη θέση στη στοίβα, για αυτό καλούμε την εντολή `sw t0, -d(fp)`, όπου d είναι ακέραιος αριθμός και ο fp θα βρίσκεται ήδη στην αρχή του εγγραφήματος δραστηριοποίησης.

Πέρασμα παραμέτρων με αναφορά :

Όταν έχουμε τέτοιες παραμέτρους με αναφορά η συμπλήρωσή τους είναι πιο περίπλοκη. Αρχικά θα πρέπει να βρούμε το entity στο οποίο υπάρχει η μεταβλητή αυτή ώστε να βρούμε και το scope στο οποίο υπάρχει. Έπειτα, θα χωρίσουμε τις περιπτώσεις σε δύο κατηγορίες. Η πρώτη θα είναι όταν στη θέση της στοίβας που μας παραπέμπει ο πίνακας συμβόλων βρίσκεται η τιμή της μεταβλητής που θέλουμε να περάσουμε σαν παράμετρο. Η δεύτερη στη θέση αυτή θα έχει τη διεύθυνση. Θεωρούμε πως σε αυτό το σημείο πρέπει να αναφέρουμε πως αυτό μπορούμε πιο γρήγορα να το ελέγξουμε μέσω των scopes. Πιο συγκεκριμένα, αν η μεταβλητή αυτή που διαβάζουμε δεν έχει αποθηκευμένη την μεταβλητή σε πρόγονο της τότε την έχει στο scope που βρίσκεται η ίδια, άρα γνωρίζουμε πως στη στοίβα θα είναι αποθηκευμένη η τιμή της παραμέτρου. Αν όμως βρίσκετε σε πρόγονο τότε στη στοίβα θα έχουμε αποθηκευμένη την διεύθυνση της παραμέτρου.

1) παράμετρος με αναφορά, στην στοίβα υπάρχει η τιμή της παραμέτρου :
Θα διακρίνουμε άλλες τρεις περιπτώσεις στις οποίες διαφοροποιείται ο κώδικας στην assembly αλλά όχι το ζητούμενο, το οποίο παραμένει το εξής : η διεύθυνση της μεταβλητής που περνιέται ως παράμετρος να αντιγραφεί στην κατάλληλη θέση του εγγραφήματος δραστηριοποίησης που δημιουργείται.

α) τοπική μεταβλητή ή προσωρινή μεταβλητή ή παράμετρος που έχει περαστεί με τιμή στη συνάρτηση :
θέλουμε να πάμε offset bytes πάνω από τον sp, άρα d bytes πάνω από τον fp θα τοποθετήσουμε το offset(sp)

```
addi t0,sp,-offset
sw t0,-d(fp)
```

β) καθολική μεταβλητή :
θέλουμε να πάμε offset bytes πάνω από τον gp, άρα d bytes πάνω από τον fp θα τοποθετήσουμε το offset(gp)

```
addi t0,gp,-offset
sw t0,-d(gp)
```

2) παράμετρος με αναφορά, στην στοίβα υπάρχει η διεύθυνση της παραμέτρου :
Από την στιγμή που στην στοίβα υπάρχει διεύθυνση γνωρίζουμε πως θα πρέπει να αναζητήσουμε την τιμή αυτή σε κάποιο πρόγονο άρα θα χρειαστούμε ένα gnlvcode(). Η αποθήκευση θα γίνει στο καταχωρητή t0 καθώς εκεί βρίσκεται η τιμή της μεταβλητής που μας ενδιαφέρει. Οπότε αφού έχουμε εντοπίσει την διεύθυνση όπως και παραπάνω αναφέραμε, την αντιγράφουμε στη θέση μνήμης που έχουμε δεσμεύσει στο νέο εγγράφημα δραστηριοποίησης. Να μην παραλείψουμε πως, η gnlvcode() θα πρέπει να τοποθετήσει στο καταχωρητή t0 τη διεύθυνση στη στοίβα στην οποία βρίσκεται η διεύθυνση που αναζητούμε.

```
call gnlvcode(x)
lw t0,(t0)
sw t0,-d(fp)
```

Κλήση συνάρτησης :

Αρχικά γεμίζουμε το δεύτερο πεδίο του εγγραφήματος δραστηριοποίησης της κληθείσας συνάρτησης, τον σύνδεσμο προσπέλασης, με την διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα της. Με αυτό το τρόπο η κληθείσα γνωρίζει σε ποιο σημείο θα γίνει η προσπέλαση της μεταβλητής που δεν είναι δική της αλλά μπορεί να την διαβάσει. Οι περιπτώσεις που έχουμε είναι δυο. Στην πρώτη περίπτωση η καλούσα και η κληθείσα συνάρτηση είναι “αδερφές” με κοινό πρόγονο και στην δεύτερη η καλούσα και η

κληθείσα δεν έχουν κοινό πρόγονο. Αυτό το καταλαβαίνουμε όταν βλέπουμε ότι και οι δυο έχουν ίδιο βάθος φωλιάσματος. Στην πρώτη περίπτωση φορτώνουμε στον καταχωρητή t0 την τιμή 4 θέσεις μνήμης πάνω από τον καταχωρητή sp και φορτώνουμε την τιμή του t0 4 θέσεις μνήμης πριν από τον fp που δείχνει στο τέλος του εγγραφήματος δραστηριοποίησης της κληθείσας. Στην δεύτερη περίπτωση που η καλούσα και η κληθείσα δεν έχουν κοινό πρόγονο πρέπει να γράψουμε τη διεύθυνση που έχει κρατήσει ο sp 4 θέσεις μνήμης μετά τον fp. Στη συνέχεια και για τις δυο περιπτώσεις πρέπει να μεταφέρουμε τον δείκτη στοίβας στην κληθείσα με addi sp, sp, framelength (framelength της κληθείσας), να καλέσουμε τη συνάρτηση με Jump jal L34 (όπου L34 είναι το Label του κώδικα της συνάρτησης) και όταν επιστρέψουμε να πάρουμε πίσω τον δείκτη στοίβας στην καλούσα με
addi sp, sp, -framelenth.