Φωτεινή Υφαντή 4516

Κωνσταντίνος Χριστόπουλος 4527

COMPILER PROJECT REPORT

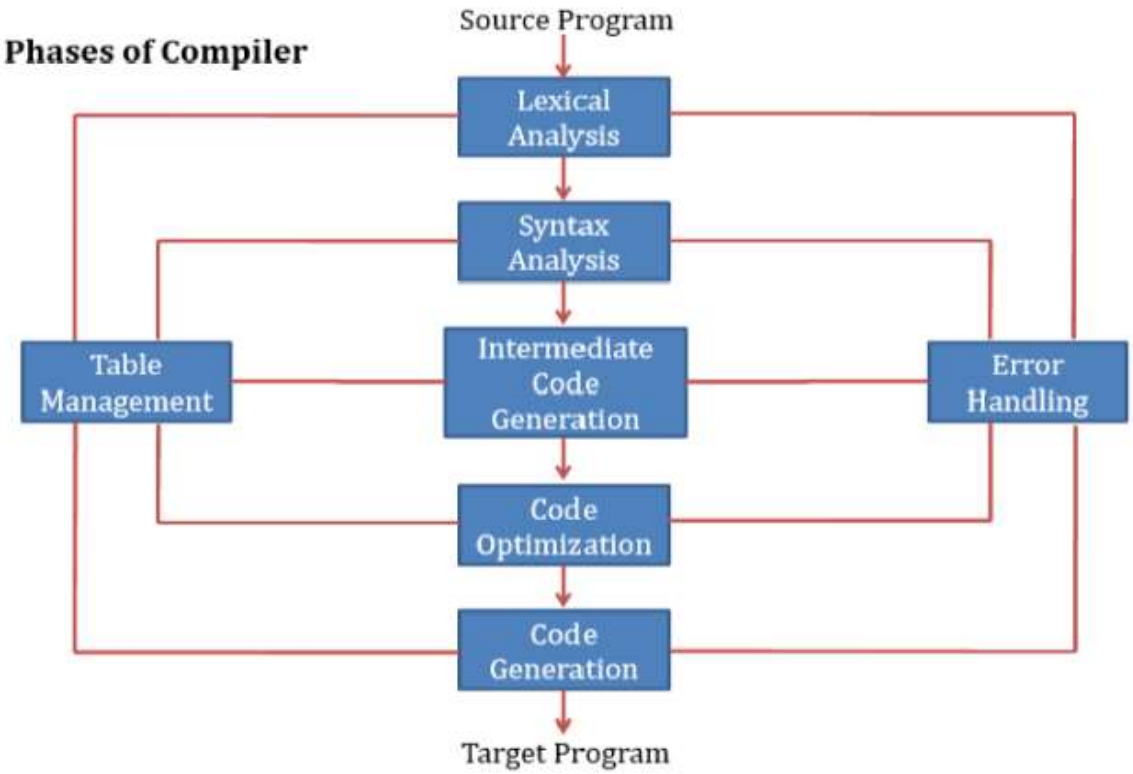
using

C-imple

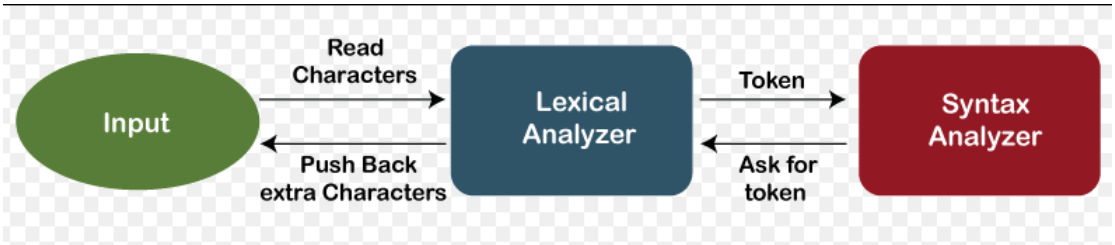
Ο μεταγλωττιστής που φτιάξαμε παράγει βηματικά τον Λεκτικό Αναλυτή, τον Συντακτικό Αναλυτή, τον Ενδιάμεσο κώδικα, τον Πίνακα Συμβόλων και τον τελικό κώδικα σε γλώσσα μηχανής (assembly code) σε RISC-V επεξεργαστή.

Κάθε .ci (Αρχικό Πρόγραμμα) αρχείο μεταγλωττίζεται σε αρχείο .asm (Τελικό Πρόγραμμα).

Η υλοποίηση του project έγινε σε γλώσσα Python (Γλώσσα Υλοποίησης του Μεταφραστή).



ΛΕΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ



Η λεκτική ανάλυση είναι η πρώτη φάση της μεταγλώττισης ενός προγράμματος. Η είσοδος του Λεκτικού Αναλυτή είναι αρχεία σε γλώσσα C-imple. Ο Συντακτικός Αναλυτής ζητάει τις λεκτικές μονάδες από τον Λεκτικό, όπου εκεί αναγνωρίζονται. Ο Συντακτικός καλεί τον Λεκτικό ως μια συνάρτηση, η οποία επιστρέφει την επόμενη λεκτική μονάδα κάθε φορά. Η λεκτική μονάδα είναι ένα αντικείμενο αποτελούμενο από τρία πεδία:

1. **Recognized_string** : Η συμβολοσειρά που αναγνωρίστηκε.

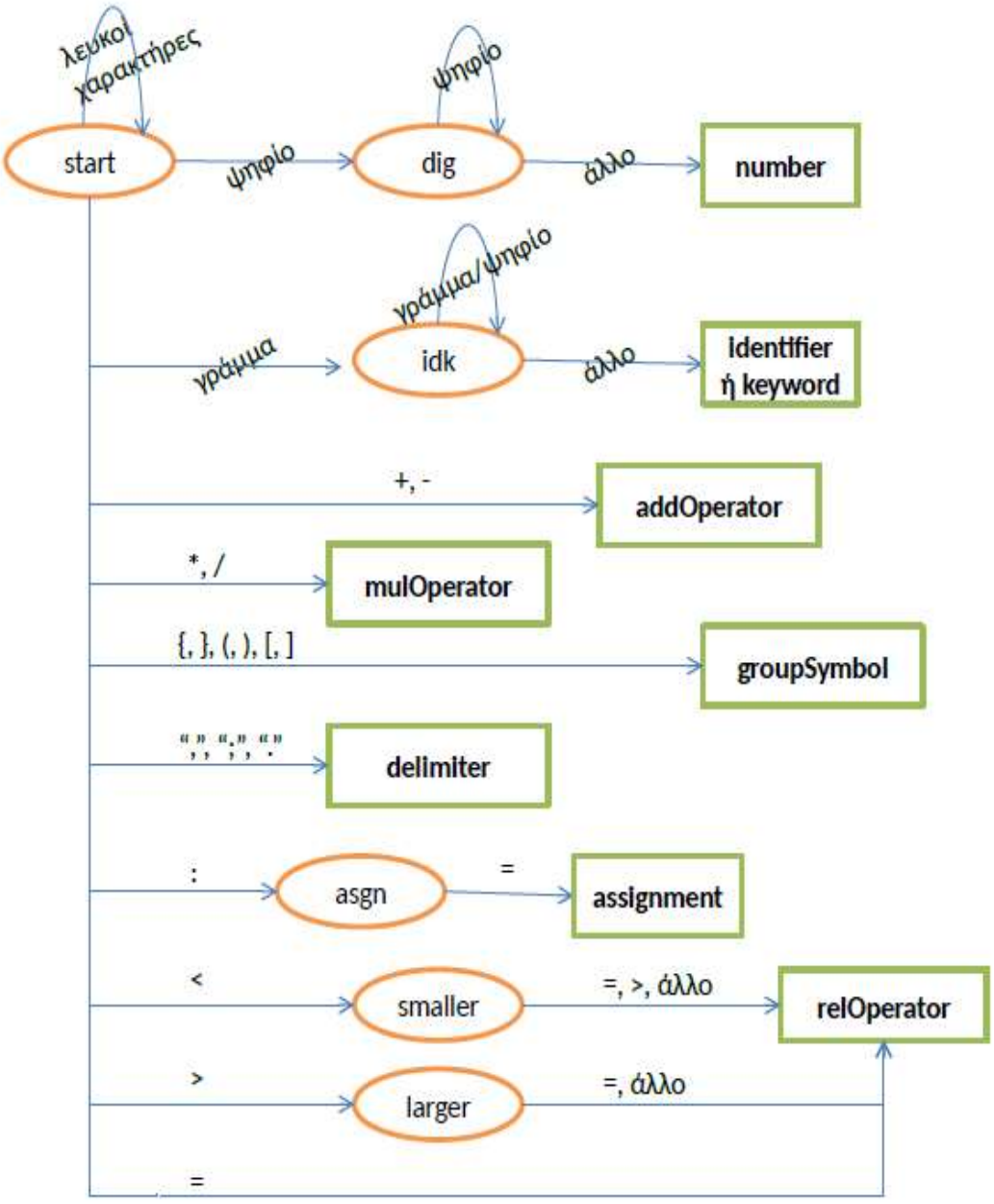
   Συγκεκριμένα, οι συμβολοσειρές μπορεί να είναι λέξεις του αγγλικού αλφαβήτου (κεφαλαία και μικρά γράμματα) ή λέξεις που περιέχουν ψηφία, αριθμοί, αριθμητικοί τελεστές («+», «-» , «*», «/»), σχεσιακοί τελεστές («<», «>», «=»), σημεία στίξης («.», «;», «,», «:»), σύμβολα ομαδοποίησης («(», «)», «{», «}», «[», «]»), το σύμβολο δημιουργίας σχολίων («#»), κενοί χαρακτήρες (tab, space) και ο χαρακτήρας αλλαγής γραμμής («\n») (διαφέρουν για την διευκόλυνση στην αρίθμηση γραμμών).

2. **Family:** Η κατηγορία όπου εντάσσεται το κάθε recognized string σύμφωνα με το είδος του.

   Ειδικότερα, οι κατηγορίες είναι ονόματα μεταβλητών, συναρτήσεων κλπ. (identifier), δεσμευμένες λέξεις (keyword), αριθμοί (number), προσθετικοί τελεστές (addOperator), πολλαπλασιαστικοί τελεστές (mulOperator), λογικοί τελεστές (relOperator), ο τελεστής εκχώρησης (assignment), διαχωριστές (delimiter) και σύμβολα ομαδοποίησης (groupSymbol).
3. **Αριθμός γραμμής:** Κάθε φορά που αναγνωρίζεται ο χαρακτήρας αλλαγής γραμμής, ο line counter αυξάνεται κατά ένα. Έτσι, γνωρίζουμε κάθε φορά

σε ποια γραμμή του προγράμματος βρισκόμαστε. Χρησιμοποιείται αργότερα στην συντακτική ανάλυση.

Ο Λεκτικός Αναλυτής προσομοιώνει τη λειτουργία ενός πεπερασμένου αυτόματου καταστάσεων όπου ελέγχει κάθε φορά που ένας νέος χαρακτήρας συναντάται αν θα παραμείνει ή εάν θα αλλάξει κατάσταση. Το αυτόματο που χρησιμοποιήθηκε είναι το ακόλουθο:



ΥΛΟΠΟΊΗΣΗ:

Για την υλοποίηση της λεκτικής μονάδας (token) αρχικοποιήσαμε μεταβλητές που αντιπροσωπεύουν κάθε πιθανό χαρακτήρα για την αναγνώριση του αλφαριθμητικού (recognized_string) σε αρίθμηση από 0 έως 23 (αυθαίρετα). Για την κατηγορία (family) έχουμε χρησιμοποιήσει αντίστοιχες μεταβλητές (variable_tk) με αρίθμηση από 24 έως 47 και την συνάρτηση *conversion_to_string*, η οποία σύμφωνα με την αρίθμηση αυτή σε κατευθύνει στην σωστή κατηγορία.

Για την υλοποίηση του αυτόματου χρησιμοποιήσαμε 7 διαφορετικές καταστάσεις. Τις:

- start_state: για την αρχική κατάσταση
- letter_digit_state: για την κατάσταση όπου έχει αναγνωριστεί λέξη ή λέξη που περιέχει αριθμό (ξεκινάει με γράμμα)
- digit_state: για κατάσταση που έχει αναγνωριστεί ακέραιος
- lessthan_state: για κατάσταση που έχει αναγνωριστεί σύμβολο «<»
- greaterthan_state: για κατάσταση που έχει αναγνωριστεί σύμβολο «>»
- assignment_state: για κατάσταση που έχει αναγνωριστεί σύμβολο «:=»
- comment_state: για κατάσταση που έχει αναγνωριστεί σύμβολο «#»

Επίσης, αρχικοποιήσαμε τα έξι διαφορετικά errors που συναντάμε στον Λεκτικό Αναλυτή σε αρνητικούς αριθμούς. Τα σφάλματα αυτά είναι τα εξής:

- ERROR_NOT_LEGAL_CHARACTER: σε περίπτωση που βρεθεί μη επιτρεπτός χαρακτήρας.
- ERROR_COMMENT_IN_EOF: σε περίπτωση που το τέλος του αρχείου βρεθεί πριν το κλείσιμο ενός σχολίου (που έχουμε ανοίξει νωρίτερα).
- ERROR_COLON_WITHOUT_EQUAL: σε περίπτωση που έχει αναγνωριστεί ή «:» και ύστερα δεν ακολουθείται από το «=», διότι η «:» ανήκει γλώσσα μόνο στην περίπτωση του τελεστή εκχωρήσεως.
- ERROR_DIGIT_WITH_LETTER : σε περίπτωση που έχει αναγνωριστεί ως πρώτος χαρακτήρας αριθμός και ύστερα αναγνωρίζεται γράμμα.
- ERROR_NUMBER_OUT_OF_SPACE: σε περίπτωση που ο αριθμός είναι εκτός επιτρεπτών ορίων (>= 2^{32}).

- ERROR_IDENTIFIER_MORE_THAN_30_CHARACTERS: σε περίπτωση που το αλφαριθμητικό περιέχει πάνω από 30 χαρακτήρες, είναι εκτός ορίων.

Βασιζόμενοι στο αυτόματο αυτό δημιουργήσαμε έναν πίνακα *transition_table* που απαρτίζεται από επτά λίστες, μία για κάθε κατάσταση. Η πρώτη λίστα αντιπροσωπεύει την start_state και για κάθε πιθανό χαρακτήρα σαν είσοδο οδηγεί σε διαφορετική κατάσταση ή στην αναγνώριση κάποιου αλφαριθμητικού.

Αρχικά, αναφέρεται η start_state. Όσο η είσοδος είναι κενός χαρακτήρας, παραμένει στην αρχική κατάσταση, καθώς δεν έχει αναγνωριστεί κανένα αλφαριθμητικό ακόμη. Στην πρώτη εμφάνιση ενός γράμματος οδηγούμαστε στην letter_digit_state. Ενώ αν ο πρώτος χαρακτήρας που εμφανιστεί είναι αριθμός οδηγούμαστε στην digit_state. Αν ο χαρακτήρας που εμφανιστεί είναι ο προσθετικός, αφαιρετικός, πολλαπλασιαστικός ή διαιρετικός τελεστής ή το ίσον, τότε το αυτόματο οδηγείται στην αναγνώριση της εκάστοτε λεκτικής μονάδας (πχ plus_tk). Στην περίπτωση που η είσοδος είναι ο λογικός τελεστής «<», το σύστημα μεταβαίνει στην κατάσταση lessthan_state, ενώ αν είναι ο «>» αντίστοιχα στην greaterthan_state. Εφόσον διαβαστεί το σύμβολο παραγωγής σχολίων «#» οδηγούμαστε στην κατάσταση comment_state. Αν η είσοδος είναι comma «,», questionmark «;», back_parenthesis «(», front_parenthesis «)», back_block «}», front_block «{», back_bracket «]», front_bracket «[» ή fullstop «.», τότε το σύστημα οδηγείται στην αναγνώριση του αντίστοιχου token. Εάν βρεθεί όμως «:» το αυτόματο πηγαίνει στην κατάσταση assignment_state. Ο new line character συμπεριφέρεται όμοια με τους κενούς χαρακτήρες και άρα το σύστημα παραμένει στην αρχική του κατάσταση. Αν φτάσουμε στο τέλος του αρχείου τότε αναγνωρίζεται το endoffile_tk και αν βρεθεί κάποιο σφάλμα το σύστημα οδηγείται στην ERROR_NOT_LEGAL_CHARACTER.

Ύστερα θα μελετήσουμε την letter_digit_state, δηλαδή την κατάσταση όπου έχει αναγνωριστεί σίγουρα ένα γράμμα και ελέγχουμε τις επόμενες εισόδους από αυτό. Αν η είσοδος είναι λευκός χαρακτήρας τελειώνει η αναγνώριση του αλφαριθμητικού και άρα οδηγούμαστε στο identifier_tk. Αν η είσοδος είναι γράμμα ή αριθμός παραμένουμε στην letter_digit_state. Σε κάθε άλλη περίπτωση πλην αυτής που συναντάται σφάλμα οδηγούμαστε στην τελική κατάσταση αναγνώρισης identifier_tk, ενώ αν βρεθεί σφάλμα το σύστημα πηγαίνει στην ERROR_NOT_LEGAL_CHARACTER.

Στην συνέχεια θα ερευνηθεί η digit_state, όπου έχει ήδη αναγνωριστεί ένα πρώτο ψηφίο. Εάν ο επόμενος χαρακτήρας είναι κενός τότε η διαδικασία της αναγνώρισης ολοκληρώνεται με τον constant_tk που βρέθηκε. Σε περίπτωση που βρεθεί γράμμα, το αυτόματο οδηγείται στο ERROR_DIGIT_WITH_LETTER σφάλμα. Εφόσον εξακολουθούν να εμφανίζονται ψηφία παραμένουμε στην digit_state. Σε κάθε άλλη περίπτωση πλην αυτής που συναντάται σφάλμα οδηγούμαστε στην τελική κατάσταση αναγνώρισης constant_tk, ενώ αν βρεθεί σφάλμα το σύστημα πηγαίνει στην ERROR_NOT_LEGAL_CHARACTER.

Για την κατάσταση του lessthan_state έχει ήδη αναγνωριστεί το σύμβολο «<» και στην περίπτωση που ακολουθείται από το ίσον «=» τότε οδηγούμαστε στην αναγνώριση του lessequal_tk token, ενώ αν βρεθεί ο τελεστής «>» οδηγούμαστε στην αναγνώριση του different_tk token. Σε κάθε άλλη περίπτωση πλην αυτής που συναντάται σφάλμα οδηγούμαστε στην τελική κατάσταση αναγνώρισης lessthan_tk, ενώ αν βρεθεί σφάλμα το σύστημα πηγαίνει στην ERROR_NOT_LEGAL_CHARACTER.

Για την κατάσταση του greaterthan_state έχει ήδη αναγνωριστεί το σύμβολο «>» και στην περίπτωση που ακολουθείται από το ίσον «=» τότε οδηγούμαστε στην αναγνώριση του greaterequal_tk token. Σε κάθε άλλη περίπτωση πλην αυτής που συναντάται σφάλμα οδηγούμαστε στην τελική κατάσταση αναγνώρισης greaterthan_tk, ενώ αν βρεθεί σφάλμα το σύστημα πηγαίνει στην ERROR_NOT_LEGAL_CHARACTER.

Στην assignment_state έχει ήδη αναγνωριστεί η άνω-κάτω τελεία. Σε κάθε περίπτωση πέραν αυτής που ακολουθείται από το ίσον οδηγούμαστε σε σφάλμα ERROR_COLON_WITHOUT_EQUAL, ενώ όταν αναγνωριστεί το ίσον το σύστημα οδηγείται στο assignment_tk. Εάν βρεθεί σφάλμα μεταβαίνει στο ERROR_NOT_LEGAL_CHARACTER.

Τέλος, για την comment_state ισχύει ότι παραμένουμε σε αυτήν εκτός μόνο αν βρεθεί το τέλος του αρχείου που οδηγεί στο ERROR_COMMENT_IN_EOF ή αν αναγνωριστεί ξανά ο χαρακτήρας έναρξης σχολίων «#» και έτσι μεταβαίνει στην αρχική κατάσταση (start_state).

Στην υλοποίηση αυτή χρειάστηκε να ορίσουμε την συνάρτηση recognized_string_function, η οποία επιστρέφει την κατάσταση και τη γραμμή που βρισκόμαστε κάθε φορά. Καλείται όταν το αυτόματο βρίσκεται σε ενδιάμεση κατάσταση (δηλαδή σε ένα από τα 7 states που θέσαμε) και διαβάζει τον επόμενο χαρακτήρα από το αρχείο. Μέσα στην συνάρτηση αναγνωρίζεται

αυτός ο χαρακτήρας και αρχικοποιείται σε έναν αριθμό βάσει της αρίθμησης που έχουμε κάνει από 0-23, όσοι είναι οι χαρακτήρες που επιφέρουν αλλαγές στο αυτόματο. Σύμφωνα με την κατάσταση που βρισκόμαστε και τον χαρακτήρα που αναγνωρίσαμε (δηλαδή οι δύο αριθμοί που τα χαρακτηρίζουν) μεταβαίνουμε σε επόμενη κατάσταση, η οποία αντιστοιχεί στην θέση του *transition_table*. Στην περίπτωση που αυτή η κατάσταση είναι τελική ή σφάλμα (το state δεν ανήκει στο διάστημα [0,6]) παύει να καλείται.

Ένα στιγμιότυπο εκτέλεσης θα μπορούσε να είναι όταν βρισκόμαστε στην lessthan_state που έχει αρχικοποιηθεί στον αριθμό 3 και ο επόμενος χαρακτήρας που αναγνωρίζεται είναι το σύμβολο «>», το οποίο αρχικοποιείται στον αριθμό 9. Έτσι, η θέση *transition_table[3][9] = different_tk* που είναι η αναγνώριση του συμβόλου «διάφορο».

Οπισθοδρόμηση

Σε περιπτώσεις που έχει αναγνωριστεί identifier_tk, constant_tk, lessthan_tk ή greaterthan_tk χρειάζεται να γίνει μια οπισθοδρόμηση. Πιο συγκεκριμένα, για την αναγνώρισή τους θα πρέπει να καταναλωθεί και ο επόμενος χαρακτήρας στο αρχείο εισόδου, ο οποίος είναι κομμάτι της επόμενης λεκτικής μονάδας. Ειδικότερα, επιστρέφουμε μία θέση πίσω στο αρχείο που διαβάζουμε και αφαιρούμε από το recognized_string τον τελευταίο του χαρακτήρα. Με αυτόν τον τρόπο καταφέρνουμε να επιστρέφουμε σωστά το recognized_string και σε επόμενη κλήση του Λεκτικού Αναλυτή να διαβάζουμε σωστά τον χαρακτήρα της επόμενης λεκτικής μονάδας. Για τους λογικούς τελεστές «<» και «>» η οπισθοδρόμηση χρησιμεύει για να αποκλειστούν ή να επιβεβαιωθούν οι περιπτώσεις του lessequal_tk και greaterequal_tk αντίστοιχα, όπως και η περίπτωση του different_tk, αναλόγως τι ακολουθεί τους τελεστές. Όσον αφορά το identifier_tk και το constant_tk, ο έλεγχος χρειάζεται για να επιβεβαιωθεί ότι το αλφαριθμητικό ή ο αριθμός αντίστοιχα έχουν ολοκληρωθεί, καταναλώνοντας τον επόμενο χαρακτήρα τους και επιστρέφοντας πάλι έναν χαρακτήρα πίσω για να κατοχυρωθεί η αναγνώριση.

Επί παραδείγματι, όταν έχει διαβαστεί ένα ψηφίο γνωρίζουμε ότι πρέπει να αναγνωριστεί ένας αριθμός, εκτός περίπτωσης σφάλματος. Για να σιγουρευτούμε για το πλήθος των ψηφίων του αριθμού “κρυφοκοιτάζουμε” τον επόμενο χαρακτήρα. Αν αυτός εξακολουθεί να είναι ψηφίο, παραμένουμε στην ίδια κατάσταση. Αν ακολουθείται από ο,τιδήποτε άλλο, καταλαβαίνουμε ότι η εμφάνιση του αριθμού ολοκληρώθηκε και έτσι επιστρέφουμε στο τελευταίο ψηφίο του και επιβεβαιώνουμε την αναγνώρισή του. Στην επόμενη

κλήση του Λεκτικού Αναλυτή θα επιστραφεί ο χαρακτήρας που “κρυφοκοιτάξαμε” νωρίτερα.

Στην συνέχεια της Λεκτικής Ανάλυσης, οφείλει να γίνει ένας διαχωρισμός ανάμεσα στα identifiers και τα keywords, καθώς το αυτόματο αναγνωρίζει και τις δεσμευμένες λέξεις της γλώσσας. Για αυτόν τον λόγο, δημιουργήσαμε μια λίστα *bounded_words* όπου συμπεριλάβαμε σε strings όλες τις λέξεις κλειδιά της C-imple. Ύστερα, αρχικοποιήσαμε και αυτές στον αντίστοιχο keyword_tk ώστε αργότερα στον Συντακτικό Αναλυτή να γίνουν οι ανάλογες λειτουργίες τους, όπως περιγράφονται. Έτσι, ελέγχουμε όταν βρισκόμαστε στην κατάσταση αναγνώρισης ενός identifier_tk αν ανήκει στην λίστα των δεσμευμένων λέξεων, αν όχι τότε γνωρίζουμε πως είναι όνομα προγράμματος, μεταβλητής, διαδικασίας ή συνάρτησης.

Τέλος, παράγεται **το αποτέλεσμα του Λεκτικού Αναλυτή** που είναι το **token**, το οποίο έχει υλοποιηθεί σε μια λίστα με τρία στοιχεία. Το πρώτο στοιχείο αντιπροσωπεύει το recognized_string, το δεύτερο την κατηγορία family που ανήκει και το τρίτο την γραμμή που έγινε η αναγνώριση στο αρχείο εισόδου.

ΣΥΝΤΑΚΤΙΚΟΣ ΑΝΑΛΥΤΗΣ

Η δεύτερη φάση της μεταγλώττισης είναι η Συντακτική Ανάλυση, η οποία είναι υπεύθυνη για την τήρηση των γραμματικών κανόνων της γλώσσας. Όποτε υπάρχει παραβίασή τους, το σύστημα οδηγείται σε συντακτικά λάθη, τα οποία τυπώνονται στην κονσόλα.

ΛΕΙΤΟΥΡΓΙΑ ΣΥΝΤΑΚΤΙΚΟΥ ΑΝΑΛΥΤΗ:

Η λειτουργία του Συντακτικού Αναλυτή είναι βασισμένη στη γραμματική LL(1), δηλαδή:

- L for Left to right: αναγνωρίζει από αριστερά προς δεξιά

- L for Leftmost derivation: την αριστερότερη δυνατή παραγωγή
- (1) for one look-ahead symbol: όταν είναι σε δίλημμα ποιον κανόνα να ακολουθήσει, ελέγχει το αμέσως επόμενο σύμβολο από την είσοδο

Ο Συντακτικός Αναλυτής δημιουργεί το κατάλληλο περιβάλλον μέσα από το οποίο αργότερα θα κληθούν οι σημαντικές ρουτίνες. Εμβαθύνοντας, για κάθε κανόνα της γραμματικής C-imple δημιουργούμε από μία συνάρτηση. Σε κάθε κανόνα/συνάρτηση συναντάμε τερματικά και μη τερματικά σύμβολα. Για τα μη τερματικά συνεχίζουμε καλώντας την αντίστοιχη συνάρτηση του συμβόλου αυτού. Ενώ, για τα τερματικά καλούμε τον Λεκτικό Αναλυτή, ο οποίος μας επιστρέφει μια λεκτική μονάδα. Αν αυτή η λεκτική μονάδα που περιμένει ο Συντακτικός Αναλυτής, αντιστοιχεί σε αυτό το τερματικό σύμβολο, τότε το έχουμε αναγνωρίσει σωστά. Αντιθέτως, αν δεν αντιστοιχεί, τότε εμφανίζεται στην κονσόλα μήνυμα σφάλματος και τερματίζεται η μετάφραση. Επιτυχώς τερματίζεται όταν αναγνωριστεί σωστά και η τελευταία λέξη.

<u>Γραμματική της C-imple:</u>

```
# "program" is the starting symbol
# followed by its name and a block
# Every program ends with a fullstop
program         →   program ID
                    block
                    .

# a block consists of declarations, subprograms and statements
block           →   {

                        declarations
                        subprograms
                        blockstatements
                    }
```

```
# declaration of variables
# Kleene star implies zero or more "declare" statements
declarations  →   ( declare varlist ; )*

# a list of variables following the declaration keyword
varlist       →    ID
                   ( , ID )*
              |    ε

# zero or more subprograms
subprograms   →    ( subprogram )*


# a subprogram is a function or a procedure,
# followed by parameters and block
subprogram    →    function ID ( formalparlist )
                   block
              |    procedure ID ( formalparlist )
                   block

# list of formal parameters
# one or more parameters are allowed
formalparlist →    formalparitem
                   ( , formalparitem )*
              |    ε

# a formal parameter
# "in": by value, "inout" by reference
formalparitem →    in ID
              |    inout ID

# one or more statements
# more than one statements should be grouped with brackets
statements    →    statement ;
              |    {
                        statement
                        ( ; statement )*
                   }


 # statements considered as block (used in program and subprogram)
 blockstatements→   statement
                    ( ; statement )*

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

```
# assignment statement
assignStat    →    ID := expression

# if statement
ifStat        →    if ( condition )
                        statements
                   elsepart

# else part is optional
elsepart      →    else
                        statements
              |    ε

# while statement
whileStat     →    while ( condition )
                        statements

# switch statement
switchcaseStat →   switchcase
                     ( case ( condition ) statements )*
                     default statements

# forcase statement
forcaseStat   →    forcase
                       ( case ( condition ) statements )*
                       default statements

# incase statement
incaseStat    →    incase
                        ( case ( condition ) statements )*

# return statement
returnStat    →    return( expression )

# call statement
callStat      →    call ID( actualparlist )

# print statement
printStat     →    print( expression )

# input statement
inputStat     →    input( ID )
```

```
# list of actual parameters
actualparlist →    actualparitem
                   ( , actualparitem )*
               |    ε

# an actual parameter
# "in": by value, "inout" by reference
actualparitem →    in expression
               |   inout ID

# boolean expression
condition     →    boolterm
                   ( or boolterm )*

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

# factor in boolean  expression
boolfactor    →    not [ condition ]
               |   [ condition ]
               |   expression REL_OP expression

# arithmetic expression
expression    →    optionalSign term
                   ( ADD_OP term )*

# term in arithmetic expression
term          →    factor
                   ( MUL_OP factor )*

# factor in arithmetic expression
factor        →    INTEGER
               |   ( expression )
               |   ID idtail

# follows a function or procedure
# describes parethneses and parameters
idtail        →    ( actualparlist )
               |    ε

# symbols "+" and "-" (are optional)
optionalSign  →    ADD_OP
               |   ε

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

```
ADD_OP      →    + | -
MUL_OP      →    * | /
INTEGER     →    [0-9]+
ID          →    [a-zA-Z][a-zA-Z0-9]*
```

Βασισμένοι σε αυτή τη γραμματική δημιουργήσαμε τις συναρτήσεις του Συντακτικού Αναλυτή. Οτιδήποτε αναφέρεται με κόκκινο χρώμα στο παραπάνω documentation δεν είναι μέρος της γραμματικής αλλά δυνατότητα επιλογής. Ενώ ό,τι αναφέρεται με πράσινο χρώμα είναι κομμάτι της σύνταξης. Το Kleene star («*») δηλώνει την εμφάνιση καμίας, μίας ή περισσότερων φορών αυτού που περιέχεται μέσα στην παρένθεση. Προγραμματιστικά το υλοποιήσαμε με τη χρήση while. Συγκεκριμένα, για όσο εμφανίζεται η πρώτη λέξη της παρένθεσης εκτελείται το εκάστοτε κομμάτι κώδικα μέσα στην λούπα. Μια ακόμη περίπτωση που συναντάται στη γραμματική μας είναι αυτή του OR («|»), το οποίο μεταφράστηκε προγραμματιστικά σε if-elif-else ελέγχους για την εξέταση όλων των ενδεχομένων. Συμπεριλαμβάνεται ακόμη σε κάποιους κανόνες και η δυνατότητα του κενού («ε»), η οποία πρακτικά δεν αντιστοιχεί σε κάποιο κομμάτι κώδικα, δηλαδή δεν υπάρχουν συντακτικά λάθη. Αντιθέτως, οι κανόνες που δεν έχουν αυτήν την επιλογή (του κενού), πέραν της βασικής λειτουργίας τους, αν ο Λεκτικός Αναλυτής δεν επιστρέφει αυτό που περιμένει ο Συντακτικός, τότε υποχρεωτικά τυπώνονται errors.

Κάθε συνάρτηση που φτιάξαμε ελέγχει αρχικά αν το recognized_string (*lexical_analyzer_result[0]*) που επιστρέφεται από τον Λεκτικό Αναλυτή ισούται με την δεσμευμένη λέξη που αναφέρεται η συνάρτηση. Ειδική περίπτωση αποτελεί η αναγνώριση ονομάτων (προγράμματος, μεταβλητής, διαδικασίας, συνάρτησης κλπ.) όπου ελέγχεται το family/κατηγορία (*lexical_analyzer_result[1]*). Αυτό συμβαίνει καθώς τον Συντακτικό Αναλυτή δεν τον ενδιαφέρει να γνωρίζει το recognized_string της συγκεκριμένης λεκτικής μονάδας που επέστρεψε ο Λεκτικός. Τον ενδιαφέρει μόνο να ελέγχει ότι είναι της κατηγορίας identifier. Έπειτα, καταναλώνεται, δηλαδή καλούμε τον Λεκτικό Αναλυτή και διαβάζουμε την επόμενη λεκτική μονάδα, προχωρώντας στην υλοποίηση της σύνταξης του κάθε κανόνα. Στην περίπτωση που δεν επιστραφεί από τον Λεκτικό αυτό που «περιμένουμε» βάσει του εκάστοτε κανόνα, τότε τυπώνουμε ένα περιγραφικό μήνυμα σφάλματος, στο οποίο αναφέρεται και ο

αριθμός της λεκτικής μονάδας που βρέθηκε το error τερματίζοντας την εκτέλεση του μεταγλωττιστή μας.

Παρακάτω αναφέρουμε κάποια χαρακτηριστικά παραδείγματα της υλοποίησης των κανόνων.

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

1. Ελέγχουμε με μια if ότι το πρώτο στοιχείο της λίστας *lexical_analyzer_result* (δηλ. του token), που επιστρέφεται από τον λεκτικό αναλυτή, ταυτίζεται με το string “program” και αν ναι προχωράμε στην κατανάλωση αυτής της λεκτικής μονάδας, αν όχι τυπώνεται μήνυμα σφάλματος «*ERROR : Keyword 'program' was expected in line 1. All programs should start with the keyword 'program'. Instead the word '…' appeared*».
2. Ελέγχουμε με χρήση φωλιασμένης if αν ακολουθεί το όνομα του προγράμματος όπως κανονικά αναμένουμε και αν ναι, τότε αυτό καταναλώνεται και καλείται η *programBlock()*, η οποία θα εξεταστεί στην πορεία. Αν όχι, τυπώνεται το μήνυμα σφάλματος «*ERROR : The name of the program expected after the keyword 'program' in line 1. The illegal program name '…' appeared*».
3. Εφόσον στην *programBlock()* έχουν ακολουθηθεί σωστά όλοι οι κανόνες, τότε η λεκτική μονάδα που επιστρέφεται θα πρέπει να είναι τελεία. Έτσι, προχωράμε στον επόμενο έλεγχο, την ύπαρξη της τελείας «.» και αφού ισχύει, καταναλώνεται. Σε αντίθετη περίπτωση, τυπώνεται το μήνυμα σφάλματος «*ERROR : Every program should end with a fullstop, fullstop at the end is missing*».
4. Ο τελευταίος έλεγχος είναι αν η παρούσα λεκτική μονάδα ταυτίζεται με το κενό, δηλαδή ότι έχουμε φτάσει στο τέλος του αρχείου. Αν αυτό αληθεύει τότε βγαίνουμε από την συνάρτηση, ειδάλλως τυπώνεται το μήνυμα σφάλματος «*ERROR : No characters are allowed after the fullstop indicating the end of the program, line…*».

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

1. Ελέγχουμε με μια if ότι το δεύτερο στοιχείο της λίστας *lexical_analyzer_result* (δηλ. του token) ανήκει στην κατηγορία family : identifier, καθώς αναμένουμε για ένα όνομα και αν ναι το καταναλώνει.
2. Για το Kleene star («*») χρησιμοποιούμε την while loop με συνθήκη όσο εμφανίζεται ο πρώτος χαρακτήρας (μετά το άνοιγμα παρένθεσης), δηλαδή το κόμμα. Το οποίο στην συνέχεια καταναλώνεται. Στην περίπτωση που το αστέρι Kleene χρησιμοποιηθεί για 0 φορές εμφάνισης, τότε βγαίνουμε εκτός συνάρτησης.
3. Με φωλιασμένη if γίνεται ο έλεγχος αν ακολουθεί ID με τον ίδιο τρόπο που παρουσιάστηκε στο βήμα 1 και σε περίπτωση που δεν ακολουθεί όνομα, όπως αναμένεται, τότε τυπώνεται το μήνυμα σφάλματος «*ERROR : Variable was expected but didn't appear. Instead appeared ….*'».
4. Λόγω του ε δεν υπάρχει συντακτικό λάθος.

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

Σε αυτή την συνάρτηση υπάρχουν 3 διαφορετικές περιπτώσεις κανόνων:

1. Ελέγχει αν η λεκτική μονάδα που εμφανίζεται (δηλ. το

lexical_analyzer_result[0]) είναι το «*not*», αν ναι την καταναλώνει.

   i. Η αμέσως επόμενη λεκτική μονάδα που περιμένει ως είσοδο είναι το «[», οπότε γίνεται ο απαιτούμενος έλεγχος με χρήση φωλιασμένης if και εφόσον επαληθευτεί καταναλώνεται. Αν δεν επαληθευτεί εμφανίζεται μήνυμα σφάλματος «*ERROR : Back bracket was expected but didn't appear. Instead appeared…*».
   ii. Καλείται η συνάρτηση *condition()*.
   iii. Αμέσως μετά πρέπει να ακολουθεί «]». Έτσι, ελέγχεται και αν ισχύει, καταναλώνεται. Αν όχι, τυπώνεται μήνυμα σφάλματος «*Front bracket was expected but didn't appear. Instead appeared…*».

2. Αν το *lexical_analyzer_result[0]* όμως ταυτίζεται με το *«[»* μπαίνει στην elif και καταναλώνεται εκεί.

i. Καλείται η *condition().*
ii. Ελέγχεται αν ακολουθεί «]» και αν ισχύει, καταναλώνεται. Αν όχι, τυπώνεται μήνυμα σφάλματος «*Front bracket was expected but didn't appear. Instead appeared…*».

3. Αλλιώς, καλούνται ακολουθιακά η *expression()*, η *REL_OP()* και ξανά η *expression()*.

Σε αυτή την περίπτωση δεν έχουμε προσθέσει να τυπώνει πουθενά συντακτικό λάθος γιατί αν υπάρχει θα τυπωθεί μέσα από τις κλήσεις των συναρτήσεων του βήματος 3.

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

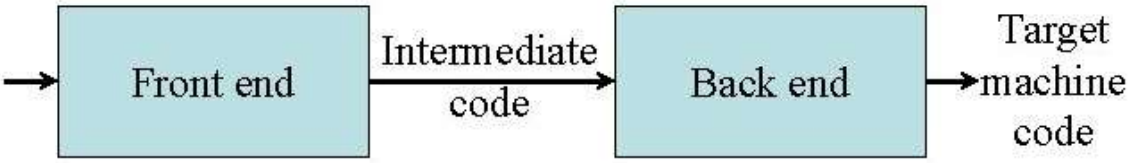
1. Για να μπούμε σε κάθε μια από αυτές τις συναρτήσεις ελέγχουμε την λεκτική μονάδα που έχει επιστραφεί από τον λεκτικό αναλυτή :

   - Αν η κατηγορία της είναι «family : identifier» τότε θα έχουμε εκχώρηση μεταβλητής και γι' αυτό καλείται η *assignStat()*.
   - Αν το recognized_string (*lexical_analyzer_result[0]*) ταυτίζεται με if, while, switchcase, forcase, incase, call, return, input ή print, τότε καλείται η εκάστοτε συνάρτηση . Σε κάθε μία από αυτές τις περιπτώσεις δεν γίνεται επανέλεγχος του recognized_string μέσα στην συνάρτηση , αλλά καταναλώνεται κατευθείαν και συνεχίζεται κανονικά ο έλεγχος της σύνταξης. Αυτό γίνεται διότι, εφόσον μπήκαμε στην υπορουτίνα αυτή, δεν γίνεται να απουσιάζει η δεσμευμένη λέξη που μας οδήγησε μέσα στον συγκεκριμένο κανόνα.

```
# switch statement
switchcaseStat →   switchcase
                       ( case ( condition ) statements )*
                       default statements
```

Στο συγκεκριμένο παράδειγμα μπαίνοντας στην switchcaseStat() έχει προηγηθεί ο έλεγχος της δεσμευμένης λέξης “switchcase”. Οπότε στην συνάρτηση δεν χρειάζεται να ξανά ελέγξουμε αν “recognized_string = switchcase“. Διαβάζουμε την επόμενη λεκτική μονάδα και συνεχίζουμε κανονικά την συντακτική ανάλυση του κανόνα. Εξάλλου, δεν υπάρχει μεταφραστής που να τυπώνει error “*ERROR : switchcase was expected but didn’t appear!*”.

ΕΝΔΙΑΜΕΣΟΣ ΚΩΔΙΚΑΣ



Ο ενδιάμεσος κώδικας έχει ως είσοδό του το αποτέλεσμα του Συντακτικού Αναλυτή και παράγεται ανάμεσα από το εμπρόσθιο (front) και το οπίσθιο τμήμα (back end). Το εμπρόσθιο τμήμα αποτελείται από τον source code που λαμβάνουμε ως είσοδο, σε γλώσσα υψηλού επιπέδου. Αντιθέτως, το οπίσθιο τμήμα είναι το αποτέλεσμα που παράγεται, δηλαδή ο τελικός κώδικας σε γλώσσα μηχανής. Το ενδιάμεσο κομμάτι αυτών των τμημάτων εξακολουθεί να είναι σε γλώσσα υψηλού επιπέδου και χρησιμεύει στην επικοινωνία μεταξύ τους, διευκολύνοντας την σχεδίαση του μεταγλωττιστή. Είναι σημαντικό να αναφερθεί ότι η ενδιάμεση αναπαράσταση είναι ανεξάρτητη του source code αλλά και του machine code, καθώς επίσης κι ότι είναι ένα μη ορατό στάδιο στον χρήστη του μεταγλωττιστή. Αλληλοεπιδρά επίσης με τον Πίνακα Συμβόλων, καθώς προσθέτει σε αυτόν πληροφορία και εγγραφές.

ΕΝΔΙΑΜΕΣΗ ΑΝΑΠΑΡΑΣΤΑΣΗ:

Ένα πρόγραμμα συμβολισμένο στην ενδιάμεση γλώσσα αποτελείται από μια σειρά από **τετράδες**. Κάθε τετράδα απαρτίζεται από έναν τελεστή και τρία τελούμενα, όπως επίσης και από τον χαρακτηριστικό αριθμό της, την ετικέτα.

- Η **ετικέτα (*label*)** δεν περιλαμβάνεται μέσα στα τέσσερα στοιχεία που εμπεριέχονται, αλλά είναι χρήσιμη για τον ορισμό της κάθε τετράδας. Ειδικότερα, υπάρχει ώστε να μπορούμε να αναφερόμαστε σε κάθε τετράδα με την ετικέτα της όταν χρειάζεται να εκτελεστούν ορισμένες λειτουργίες. Για τις ετικέτες χρησιμοποιήσαμε την κανονική αρίθμηση, ξεκινώντας από το 1.
- Ο **τελεστής** προσδιορίζει την ενέργεια που θα συμβεί στην κάθε τετράδα.
- Τα **τελούμενα** είναι αυτά στα οποία θα εκτελεστεί η εκάστοτε ενέργεια.

ΥΛΟΠΟΙΗΣΗ:

Για την αποθήκευση των τετράδων δημιουργήσαμε μια λίστα ονόματι quadsList. Στην πραγματικότητα είναι ένας πίνακας (4+1)*N καθώς θα αποτελείται από N αριθμό τετράδων, όπου κάθε τετράδα αναπαρίσταται επίσης σε λίστα. Το «+1» αναφέρεται στις υπο-λίστες στις οποίες αποθηκεύουμε και την ετικέτα που θα έχει κάθε τετράδα. Στην υλοποίηση του Ενδιάμεσου Κώδικα αναγκαία παρουσιάστηκε η δημιουργία ορισμένων συναρτήσεων, εφόσον οι λειτουργίες τους ήταν χρήσιμες σε αρκετά σημεία με σκοπό την αποφυγή επαναλήψεων κομματιών κώδικα. Οι βοηθητικές συναρτήσεις που ορίσαμε είναι οι εξής:

- genQuad(operator, operand1, operand2, operand3)

  Δημιουργεί νέες τετράδες, με τελεστή τον operator που δέχεται ως όρισμα και τελεστέους τα operand1, operand2, operand3. Η ετικέτα της προκύπτει αυτόματα σύμφωνα με την τελευταία που υπήρχε αυξημένη κατά ένα.

- nextQuad()

  Επιστρέφει την ετικέτα της επόμενης τετράδας που θα δημιουργηθεί, όταν κληθεί η genQuad().

- newTemp()

  Δημιουργεί μια νέα προσωρινή μεταβλητή και την επιστρέφει. Το όνομα που της δίνει προκύπτει αυτόματα αυξάνοντας κατά ένα τον counter που αναφέρεται στις temporary variables.

- emptyList()

  Δημιουργεί μια νέα κενή λίστα όπου αργότερα θα έχει ως στοιχεία της ετικέτες τετράδων και την επιστρέφει.

- makeList(label)

  Δημιουργεί μια νέα λίστα που περιλαμβάνει μόνο την ετικέτα της τετράδας που δέχεται ως όρισμα και την επιστρέφει.

- mergeList(list1, list2)

  Δημιουργεί μια νέα λίστα στην οποία συνενώνει τις λίστες που δέχεται ως ορίσματα επιστρέφοντάς την. Προγραμματιστικά έχουμε επεκτείνει την λίστα που δέχεται ως πρώτο όρισμα προσθέτοντας σε αυτή την list2.

- backpatch(list, label)

  Διαβάζει την λίστα (list) η οποία αποτελείται από δείκτες σε τετράδες (labels) των οποίων το τελευταίο τελούμενο είναι κενό . Αναζητάει τις λίστες τετράδων με τα συγκεκριμένα labels στη δομή quadsList και εισάγει στο τελευταίο στοιχείο της λίστας (το οποίο είναι κενό) την ετικέτα label.

Σημείωση:

Όσον αφορά τις προσωρινές μεταβλητές, τα ονόματα που τους έχουν δοθεί είναι της μορφής «T_x», όπου x ακέραιος. Αυτό συμβαίνει διότι τα ονόματά τους πρέπει να είναι μοναδικά. Για να είμαστε σίγουροι ότι δεν έχουν ξαναχρησιμοποιηθεί από τον προγραμματιστή ορίσαμε το πρώτο κομμάτι «T_» με «T» for temporary (συμβολικά) και «_» να ακολουθεί καθώς δεν ανήκει στην γλώσσα C-imple και άρα δεν μπορεί να την έχει χρησιμοποιήσει ο προγραμματιστής. Το δεύτερο κομμάτι, του ακεραίου, εξασφαλίζει ότι η προσωρινή μεταβλητή που ορίζεται δεν έχει οριστεί ξανά νωρίτερα. Έτσι, χρησιμοποιούμε έναν counter ο οποίος αυξάνει κατά ένα κάθε φορά που καλείται η *newTemp()*. Η αρίθμηση ξεκινάει από το 1.

ΕΝΤΟΛΕΣ:

Παραθέτουμε τις εντολές της ενδιάμεσης γλώσσας:

1. Αρχή και Τέλος Ενότητας

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

Χρησιμοποιείται για να δηλώσουμε την αρχή του ενδιάμεσου κώδικα για το κύριο πρόγραμμα ή για μια συνάρτηση ή για μια διαδικασία. Εισάγεται πριν την πρώτη εκτελέσιμη εντολή.

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

Ανάλογα χρησιμοποιείται για να δηλώσουμε το τέλος του ενδιάμεσου κώδικα για το κύριο πρόγραμμα, για μια συνάρτηση ή για μια διαδικασία και τοποθετείται στο τέλος του αντίστοιχου κώδικα.

Επιτακτική η ανάγκη να αναφέρουμε ότι οι εντολές ομαδοποίησης δεν επιτρέπεται να είναι εμφωλευμένες, ακόμη και σε περιπτώσεις εμφωλευμένων διαδικασιών ή συναρτήσεων. Η σειρά εμφάνισης αυτών των εντολών στον ενδιάμεσο κώδικα δεν είναι ίδια με την σειρά εμφάνισης των συναρτήσεων / διαδικασιών στον source code, αλλά η σειρά εκτέλεσής τους.

```
<PROGRAMBLOCK (name) > ::=
                        <DECLARATIONS>
                        <SUBPROGRAMS>
                        genquad("begin_block",name,"_","_")
                        <BLOCK>
                        if (this is the main program block)
                                genquad("halt","_","_","_")
                        genquad("end_block",name,"_","_")
```

Δημιουργούμε αρχικά την τετράδα του begin_block με το όνομα που δέχεται ως όρισμα η *programBlock()*. Ακολουθεί ο έλεγχος αν βρισκόμαστε στην main του προγράμματος χρησιμοποιώντας ένα flag (*mainFlag*), το οποίο επίσης είναι όρισμα στην *programBlock()*. Στην main, στο τέλος του

προγράμματος μπαίνει η τετράδα “halt. Τέλος ακολουθεί το “end_block” με το όνομα του block ανεξαρτήτως του αν βρισκόμαστε στην main.

2. Εκχώρηση

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

Εκχωρεί το source, όπου μπορεί να είναι μεταβλητή, ακέραιος ή συμβολική σταθερά, στο target, όπου είναι μια μεταβλητή.

```
S -> id := E {P1};

    {P1} :    genQuad(":=",E.place,"_",id)
```

Αποθηκεύω το source που παίρνω από την κλήση της *expression()* στο *Eplace* και το target που παίρνω από το *lexical_analyzer_result[0]* στην μεταβλητή *myid.* Δημιουργούμε την ανάλογη τετράδα αμέσως μετά την εκχώρηση, αφού έχουμε συλλέξει τη μεταβλητή και τον ακέραιο ή τη συμβολική σταθερά μας.

3. Αριθμητική πράξη

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

Το *op* αντιπροσωπεύει τα εξής: «+», «-», «*», «/» και τα *operand1, operand2* είναι τα τελούμενα στα οποία θα γίνει η αριθμητική πράξη. Το αποτέλεσμα της πράξης αποθηκεύεται στο *target.*

Οι αριθμητικές πράξεις υλοποιήθηκαν σύμφωνα με την ακόλουθη γραμματική, η οποία είναι μια απλούστευση της γραμματικής της C-imple. Σημειώνεται ότι η γραμματική της πρόσθεσης είναι ίδια με της αφαίρεσης, όπως ισχύει αντίστοιχα με τον πολλαπλασιασμό και τη διαίρεση. Η χρήση παρενθέσεων είναι για να διευκρινίζεται η προτεραιότητα πράξεων όπου αυτό είναι αναγκαίο.

```
# addition
E   →   T(1) ( + T(2) )*
# multiplication
T   →   F(1) ( * F(2) )*
# priority with parentheses
F   →   ( E )
# terminal symbols
F   →   ID
```

Κάθε κανόνας επιστρέφει μια μεταβλητή σαν αποτέλεσμα στον κανόνα που τον ενεργοποίησε. Την μεταβλητή την ονομάζουμε *place*! Για παράδειγμα, το αποτέλεσμα που θα πάρει από τον κανόνα «E» θα αποθηκευτεί στο Eplace.

- Πρόσθεση / Αφαίρεση:

  Ο κανόνας E αντιστοιχίζεται στον κώδικά μας στην συνάρτηση expression() καθώς σε αυτήν περιγράφονται αυτές οι αριθμητικές πράξεις.

```
E   →   T(1) ( + T(2) {p1} )* {p2}

{p1} :  w = newTemp()
        genQuad('+', T(1).place, T(2).place, w)
        T(1).place = w
{p2} :  E.place = T(1).place
```

  Τα $T^{(1)}$ και $T^{(2)}$ είναι διαφορετικές εμφανίσεις του ίδιου κανόνα και προγραμματιστικά ισοδυναμούν σε διαφορετικές κλήσεις της ίδιας συνάρτησης (term()). Τα αποτελέσματά τους αποθηκεύονται στις μεταβλητές T1place και T2place. Τα *{p1}* και *{p2}* υποδηλώνουν σε ποιο σημείο του κανόνα θα γίνει η κάθε σημασιολογική ενέργεια .

  Στην αρχή της εκτέλεσης του κώδικα, καλούμε την συνάρτηση *term(),* στην οποία το αποτέλεσμά της εκχωρείται σε μια μεταβλητή T1place. Στην συνέχεια, για όσο εμφανίζεται ο τερματικός χαρακτήρας + / - μπαίνουμε στην while loop όπου εξετάζεται το πλήθος των τελούμενων που θα υπάρχουν. Προγραμματιστικά αυτό μεταφράζεται στο πλήθος των τετράδων που θα δημιουργηθούν. Για κάθε εμφάνιση «+» ή «-», αποθηκεύουμε αυτόν τον χαρακτήρα στην μεταβλητή addOper (η οποία

καλεί την *ADD_OP()*) και καλούμε την *term(),* όπου αποθηκεύει το αποτέλεσμά της στην T2place. Η *{p1}* εκφράζει την περίπτωση των περισσότερων από δύο προσθετέων / αφαιρετέων. Για αυτόν τον λόγο δημιουργούνται N-1 προσωρινές μεταβλητές (για N τελεστέους), καθώς τα ενδιάμεσα αποτελέσματα των πράξεων αποθηκεύονται προσωρινά εκεί. Με τη σειρά τους οι προσωρινές μεταβλητές προστίθενται ή αφαιρούνται με τον αμέσως επόμενο τελεστέο και το αποτέλεσμά τους εκχωρείται στην T1place. Εάν δεν έχουμε κανένα τερματικό σύμβολο, ο κανόνας T εμφανίζεται μόνο μία φορά και άρα το αποτέλεσμά του ισούται με το T1.place. Σε κάθε περίπτωση το τελικό αποτέλεσμα (T1place) εκχωρείται στο E.place.

- Πολλαπλασιασμός / Διαίρεση:

```
T   →   F(1) ( * F(2) {p1} )* {p2}

{p1} :  w = newTemp()
        genQuad('*', F(1).place, F(2).place, w)
        F(1).place = w
{p2} :  T.place = F(1).place
```

Η λογική είναι ίδια με αυτής της πρόσθεσης / αφαίρεσης, με τη διαφορά ότι οι τερματικοί χαρακτήρες που συναντώνται εδώ είναι «*» ή «/» και ο κανόνας είναι ο F (*factor()*). Το τελικό αποτέλεσμα αποθηκεύεται στην Tplace.

- Προτεραιότητα

```
F   →   ( E ) {p1}

{p1} : F.place = E.place
```

Η προτεραιότητα των πράξεων υλοποιείται στον Συντακτικό Αναλυτή. Το Eplace περιέχει τη μεταβλητή που έχει το αποτέλεσμα των πράξεων που δημιουργήθηκαν από αυτόν τον κανόνα και εκχωρείται τελικώς στο Fplace.

- Αναγνώριση τερματικών συμβόλων

```
F   →   ID   {p1}

{p1} : F.place = ID.place
```

Στο Fplace αποθηκεύεται το ID που αναγνώρισε ο Συντακτικός Αναλυτής.

4. Εντολή άλματος

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

Γίνεται άλμα, δηλαδή μεταφορά ελέγχου στην εντολή που έχει την ετικέτα *label.*

5. Εντολή λογικού άλματος

```
conditional_jump, a, b, label
```

Το *conditional_jump* εκφράζει τα λογικά σύμβολα: «=», «<», «>», «<=», «>=» και «<>». Τα *a, b* είναι οι τελεστέοι ανάμεσα στους οποίους γίνεται η σύγκριση, ενώ το *label* είναι η ετικέτα στην οποία γίνεται άλμα (*jump*) εφόσον ισχύει η εκάστοτε σύγκριση.

Οι λογικές συνθήκες υλοποιήθηκαν σύμφωνα με την ακόλουθη γραμματική.

```
# boolean expression
B  →   Q ( or Q )*
# term in boolean expression
Q  →   R ( and R )*
# factor in boolean expression
R  →  not [ B ]
   |   [ B ]
   |   E rel_op E
```

Κάθε κανόνας που καλείται παίρνει δύο λίστες και τις προετοιμάζει για τον κανόνα που τον κάλεσε, οι οποίες αποτελούνται από τις ασυμπλήρωτες τετράδες, λόγω του κανόνα που αδυνατούσε να τις συμπληρώσει.

- **True list:** οι ασυμπλήρωτες τετράδες πρέπει να συμπληρωθούν με την ετικέτα της τετράδας που θα μεταβεί ο έλεγχος αν η λογική συνθήκη ισχύει.
- **False list:** οι ασυμπλήρωτες τετράδες πρέπει να συμπληρωθούν με την ετικέτα της τετράδας που θα μεταβεί ο έλεγχος αν η λογική συνθήκη δεν ισχύει.

➢ Λογικό **OR**:

```
B → Q(1) {p1} ( or {p2} Q(2) {p3} )*

{p1} : B.true = Q(1).true
       B.false = Q(1).false
{p2} : backpatch(B.false, nextquad())
{p3} : B.true = mergeList(B.true,Q(2).true)
       B.false = Q(2).false
```

Ο B ενεργοποιεί τους κανόνες Q (*boolterm()*) και διαχειρίζεται τις ασυμπλήρωτες τετράδες που έρχονται σημειωμένες στις λίστες $Q^{(1)}$.true (Q1[0]), $Q^{(1)}$.false(Q1[1]), $Q^{(2)}$.true(Q2[0]), $Q^{(2)}$.false (Q2[1]). Στην λογική συνθήκη OR χρειάζεται να ισχύει μόνο ένα εκ των δύο που εξετάζονται. Αυτό μας οδηγεί στην απόρριψη ελέγχου της δεύτερης συνθήκης αν ισχύει η πρώτη. Επομένως, μας ενδιαφέρει η περίπτωση όπου η πρώτη συνθήκη αποτιμηθεί ψευδής. Οι τετράδες που βρίσκονται σε αυτή τη λίστα, δηλαδή στην *conditionsTable_false*, θα συμπληρωθούν με την ετικέτα της αμέσως επόμενης τετράδας μέσω της backpatch(). Οι λίστες *conditionsTable_true, Q2[0]* θα κάνουν λογικό άλμα στο ίδιο σημείο, για αυτό τον λόγο τις κάνουμε merge. Η *conditionsTable_false* μετά το backpatch ταυτίζεται με την *Q2[1],* αφού θα αποτελείται από τετράδες που θα πρέπει να κάνουν άλμα στο σημείο που θέλουμε να μεταβεί ο κώδικας όταν η συνθήκη που εξετάζει ο συγκεκριμένος κανόνας δεν ισχύει.

Σε περίπτωση που δεν εμφανιστεί κανένα OR, τότε ο κανόνας Q2 δε θα ενεργοποιηθεί ποτέ, δηλαδή δεν θα μπούμε ποτέ στην while loop. Έτσι, ο κανόνας B θα έχει να διαχειριστεί μόνο τις $Q^{(1)}$.true (Q1[0]), $Q^{(1)}$.false(Q1[1]), οι οποίες θα εκχωρηθούν αντίστοιχα στις *conditionsTable_true* και *conditionsTable_false.*

Αντιθέτως, στην περίπτωση που εμφανιστούν περισσότερα από ένα OR, ενεργοποιείται το αστεράκι Kleene. Σε κάθε επανάληψη εκτελούμε backpatch() την *conditionsTable_false* και συνενώνουμε

τις νέες τετράδες που θα έρθουν από την Q2[0], έχοντας ως βάση τις *conditionsTable_true* και *conditionsTable_false.*

- Λογικό **AND**:

```
Q  →   R(1) {p1} ( and {p2} R(2) {p3} )*

{p1} : Q.true = R(1).true
       Q.false = R(1).false
{p2} : backpatch(Q.true, nextquad())
{p3} : Q.false = mergeList(Q.false,R(2).false)
       Q.true = R(2).true
```

Ο Q ενεργοποιεί τους κανόνες R (*boolfactor()*) και διαχειρίζεται τις ασυμπλήρωτες τετράδες που έρχονται σημειωμένες στις λίστες $R^{(1)}$.true (R1[0]), $R^{(1)}$.false(R1[1]), $R^{(2)}$.true(R2[0]), $R^{(2)}$.false (R2[1]). Στην λογική συνθήκη AND χρειάζεται να ισχύουν όλες οι συνθήκες που εξετάζονται. Αυτό έχει ως συνέπεια ο έλεγχος να διακόπτεται εφόσον κάποια συνθήκη αποτιμηθεί ψευδής. Αλλιώς, θα πρέπει να εξεταστούν όλες οι συνθήκες μέχρι να τελειώσουν ή να βρεθεί κάποια ψευδής. Η λίστα που πρέπει να γίνει *backpatch()* είναι η Q.true (*conditionsTable_true*), ενώ η λίστα που πρέπει να συγκεντρώσει τις τετράδες που θα μας οδηγήσουν εκτός της συνθήκης όταν δεν ισχύει είναι η Q.false (*conditionsTable_false)*.

Σε περίπτωση που δεν εμφανιστεί κανένα AND, τότε ο κανόνας $R^{(2)}$ δε θα ενεργοποιηθεί ποτέ, δηλαδή δεν θα μπούμε ποτέ στην while loop. Έτσι, ο κανόνας Q θα έχει να διαχειριστεί μόνο τις $R^{(1)}$.true (R1[0]), $R^{(1)}$.false(R1[1]), οι οποίες θα εκχωρηθούν αντίστοιχα στις *conditionsTable_true* και *conditionsTable_false.*
Αντιθέτως, στην περίπτωση που εμφανιστούν περισσότερα από ένα AND, ενεργοποιείται το αστεράκι Kleene. Σε κάθε επανάληψη εκτελούμε backpatch() την *conditionsTable_true* και συνενώνουμε τις νέες τετράδες που θα έρθουν από την R2[1], έχοντας ως βάση τις *conditionsTable_true* και *conditionsTable_false.*

- Λογικό **NOT**:

Χωρίζεται σε τρεις υποπεριπτώσεις, για αυτό στον κώδικά μας στην *boolfactor()* τις εξετάσαμε ξεχωριστά με χρήση if-elif-else.

i.

```
R  →  not [ B ] {p1}
{p1} : R.true = B.false
       R.false = B.true
```

Ο R ενεργοποιεί τον κανόνα B (*condition()*) και διαχειρίζεται τις ασυμπλήρωτες τετράδες που έρχονται σημειωμένες στις λίστες B.true (*conditionsTable[0]*), B.false(*conditionsTable[1]*). Λόγω της λογικής συνθήκης NOT όταν το B επιστρέφει αληθές, το R θα πρέπει να επιστρέφει ψευδές και αντιστρόφως. Οι τετράδες της *conditionsTable[0]* γίνονται οι τετράδες της *conditionsTable_false,* ενώ οι τετράδες της *conditionsTable[1]* γίνονται οι τετράδες της *conditionsTable_true.*

ii.

```
R  →   [ B ] {p1}

{p1} : R.true = B.true
       R.false = B.false
```

Ανάλογος με τον κανόνα του ορισμού προτεραιοτήτων με τις παρενθέσεις στις αριθμητικές εκφράσεις. Ο R ενεργοποιεί τον κανόνα B (*condition()*) και διαχειρίζεται τις ασυμπλήρωτες τετράδες που έρχονται σημειωμένες στις λίστες B.true (*conditionsTable[0]*), B.false(*conditionsTable[1]*). Λόγω της απουσίας της λογικής συνθήκης NOT, γνωρίζουμε ότι με το που αναγνωριστεί η «[» έχουμε απλή παράθεση του κανόνα B. Επομένως, οι ασυμπλήρωτες τετράδες του θα μεταφερθούν αυτούσιες στο R. Δηλαδή, οι τετράδες της *conditionsTable[0]* γίνονται οι τετράδες της *conditionsTable_true* και οι τετράδες της *conditionsTable[1]* γίνονται οι τετράδες της *conditionsTable_false.*

iii.

```
R  →  E(1) rel_op E(2) {p1}

{p1}: R.true = makeList(nextQuad())
      genQuad(rel_op, E(1).place, E(2).place}, '_')
      R.false = makeList(nextQuad())
      genQuad('jump', '_', '_', '_')
```

Ο κανόνας R περιγράφει τη σύγκριση δύο αριθμητικών εκφράσεων, των $E^{(1)}$ και $E^{(2)}$ οι οποίες καλούν την *expression()* και αποθηκεύουν τα αποτελέσματά τους στις E1place, E2place αντίστοιχα. Το rel_op αντιπροσωπεύει έναν λογικό τελεστή και για αυτό καλεί την *REL_OP().* Εδώ, οι λίστες R.true (*conditionsTable_true*), R.false(*conditionsTable_false*) δεν προκύπτουν από κάποιον άλλον κανόνα καθώς δημιουργούνται εδώ με χρήση της *makeList(nextQuad()).* Η πρώτη τετράδα που δημιουργείται από την *genQuad()* θα εκτελεστεί αν ισχύει η συνθήκη που ορίζεται σε αυτήν και θα προκύψει άλμα στην εντολή με αυτήν την ετικέτα. Αν δεν ισχύει, ο έλεγχος θα μεταβεί στην επόμενη εντολή.

```
x:   rel_op, E(1).place, E(2).place, ...
x+1: jump, _, _, ...
```

6. Κλήση συνάρτησης ή διαδικασίας

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

Καλείται η διαδικασία ονόματι *name.*

```
callStat      →    call ID( actualparlist {p1} )

{p1} :  genQuad ('call', name, '_', '_')
```

Δημιουργείται η ανάλογη τετράδα, εφόσον έχει ολοκληρωθεί η κλήση, έχοντας αποθηκεύσει σε μια μεταβλητή *name* το identifier που έχουμε λάβει.

```
idtail        →    ( actualparlist {p1} )
              |    ε

{p1}  w = newTemp()
      genQuad('par', w, 'ret', '_')
      genQuad('call', name, '_', '_')
```

Για την κλήση συναρτήσεων υπάρχει μια μικρή διαφοροποίηση σε σχέση με την κλήση διαδικασιών. Συγκεκριμένα, θα δημιουργήσουμε μία ακόμα τετράδα η οποία αντιστοιχεί στην τιμή που επιστρέφει η συνάρτηση. Οπότε,

στην συνάρτηση του συντακτικού αναλυτή *idtail()* εφόσον αναγνωριστεί η παρένθεση καλούμε την συνάρτηση *actualparlist()*, η οποία παράγει τετράδες για τις παραμέτρους του υποπρογράμματος αυτού (περισσότερες λεπτομέρειες παρακάτω). Στην συνέχεια, θα δημιουργήσουμε μια νέα προσωρινή μεταβλητή, όπου εκεί θα αποθηκευτεί το αποτέλεσμα της συνάρτησης. Τέλος, θα δημιουργήσουμε δύο ακόμη τετράδες όπως φαίνεται παραπάνω στην εικόνα.

7. Πέρασμα πραγματικής παραμέτρου

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

Γίνεται πέρασμα της παραμέτρου *name* με τρόπο *mode,* όπου το *mode* μπορεί να είναι εκ των οποίων: πέρασμα με τιμή (*cv*), πέρασμα με αναφορά (*ref*), ή επιστροφή τιμή συνάρτησης (*ret*). Στο πέρασμα με τιμή, όπως είναι γνωστό οι αλλαγές που θα γίνουν αργότερα δεν επηρεάζουν την μεταβλητή, ενώ στο πέρασμα με αναφορά οι αλλαγές αυτές ενημερώνουν την μεταβλητή μας.

```
actualparitem →    in expression {p1}
               |   inout ID  {p2}

{p1}   genQuad('par', expre, 'cv', '_')

{p2}   genQuad('par', parName, 'ref', '_')
```

Το πέρασμα των παραμέτρων υλοποιείται στην συνάρτηση *actualparitem().* Ειδικότερα, η μετάδοση τιμών με σταθερή τιμή δηλώνεται με τη λεκτική μονάδα *in* και ακολουθεί η κλήση της *expression()*. Αμέσως μετά, εφόσον έχει αποτιμηθεί τιμή στην παράμετρο, δημιουργείται η αντίστοιχη τετράδα, που περιέχει την παράμετρο, την τιμή της (expre, ότι δηλαδή επέστρεψε η *expression()*) και τον τρόπο περάσματος (cv). Η μετάδοση με αναφορά δηλώνεται με χρήση του *inout*, το οποίο ακολουθείται αυστηρώς από identifier. Μόλις αναγνωριστεί το ID του, δημιουργείται η ανάλογη τετράδα, που περιλαμβάνει την παράμετρο, την τιμή της (parName, ότι δηλαδή επέστρεψε η *lexical_analyzer_result[0]*) και τον τρόπο περάσματος (ref).

8. Επιστροφή τιμής

```
ret, source, _, _
```

Επιστρέφεται η τιμή μιας συνάρτησης και αποθηκεύεται στο *source.*

```
S -> return (E) {P1}

    {P1}:    genquad("retv",E.place,"_","_")
```

Η τιμή της συνάρτησης που επιστρέφεται μέσω της *returnStat()* αποθηκεύεται στο Eplace. Στην συνέχεια δημιουργείται η τετράδα που τυπώνει αυτό το αποτέλεσμα.

9. Είσοδος – Έξοδος

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

Αντιστοιχεί στο *inputStat()*, για την είσοδο από το πληκτρολόγιο.

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

Αντιστοιχεί στην *printStat()*, για την τύπωση μιας τιμής στην οθόνη.

```
S -> input (id) {P1}
    {P1}:    genquad("inp",id.place,"_","_")
```

Το όνομα του identifier αποθηκεύεται σε μια μεταβλητή *idPlace* και στην συνέχεια δημιουργείται η τετράδα εισόδου.

```
S -> print (E) {P2}
    {P2}:    genquad("out",E.place,"_","_")
```

Το αποτέλεσμα της *expression()* αποθηκεύεται στην *Eplace* και στην συνέχεια δημιουργείται η τετράδα εξόδου.

10. Τερματισμός προγράμματος

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

Τοποθετείται πριν από το *end_block* κι έτσι τερματίζεται κάθε πρόγραμμα. Περιλαμβάνεται μόνο στα κυρίως προγράμματα (όχι σε συναρτήσεις, διαδικασίες).

ΔΟΜΕΣ:

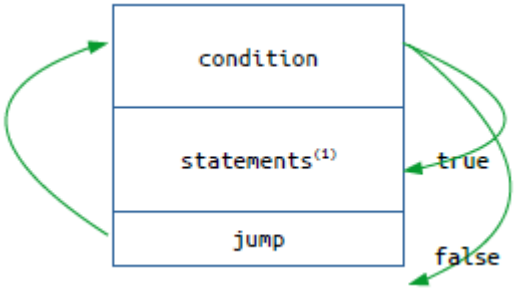
- whileStat:

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

{p0} : condQuad = nextQuad()
{p1} : backpatch(condition.true,nextQuad())
{p2} : genQuad('jump','_','_',condQuad)
       backpatch(condition.false,nextQuad())
```

Αρχικά, η condition είναι μια συνθήκη που εκτελείται στην συνάρτηση *condition()* και κατά την κλήση της θα επιστρέψει είτε αληθές, είτε ψευδές. Κρατάμε το αληθές αποτέλεσμα στην λίστα *conditionsTable[0]* και το ψευδές στην λίστα *conditionsTable[1],* καθώς περιέχουν ασυμπλήρωτες τετράδες στις οποίες θα χρειαστεί να κάνουμε backpatch(). Θα γνωρίζουμε σε τι αποτιμάται η συνθήκη αμέσως μετά την κλήση της, οπότε εκεί μπαίνει το *{p1}* (είτε πριν είτε μετά την παρένθεση, δεν έχει σημασία). Επίσης, η συνάρτηση αυτή γνωρίζουμε ότι είναι μια λούπα που συμβαίνει για όσο ισχύει η συνθήκη. Αυτό σημαίνει, ότι μετά από κάθε εκτέλεσή της, θα πρέπει να επανεξετάζεται η συνθήκη. Οδηγούμαστε, λοιπόν, στην τοποθέτηση του *{p0}* πριν την συνθήκη, καθώς εκεί γίνεται το άλμα σε κάθε «γύρο» της λούπας, και στην τοποθέτηση του *{p2}* στο τέλος της, καθώς τότε είναι που πρέπει να εκτελεστεί το άλμα. Μένει μόνο να εξετάσουμε την συνθήκη πλέον. Στην περίπτωση που αυτή είναι αληθής, τότε θα ακολουθήσει η εκτέλεση των statements (μέσω της *statements()*). Άρα, οι ασυμπλήρωτες τετράδες της *conditionsTable[0]* , πρέπει να συμπληρωθούν με την πρώτη τετράδα των statements, η οποία κιόλας είναι η αμέσως επόμενη και την παίρνουμε με την nextQuad(). Στην περίπτωση, όμως, που είναι ψευδής, τα statements δεν θα εκτελεστούν ποτέ καθώς βγαίνουμε εκτός λούπας πλέον. Τότε εκτελείται η αμέσως επόμενη εντολή μετά το while loop. Για να βρούμε αυτήν την εντολή, θα πρέπει να βάλουμε ως τελευταία εντολή στο σχέδιο ενδιάμεσου κώδικα κάποια κλήση nextQuad() η οποία να μην ακολουθείται από τίποτα άλλο, ώστε να μη χαθεί η τετράδα

που θέλουμε να κρατήσουμε. Έτσι, συμπληρώνουμε τις ασυμπλήρωτες τετράδες της *conditionsTable[1]* έπειτα από ο,τιδήποτε άλλο, με την τετράδα της εντολής που ακολουθεί, τερματίζοντας έτσι την while.

Παραθέτουμε στην συνέχεια σχηματικά την παραγωγή ενδιάμεσου κώδικα για την δομή while.



- <u>ifStat:</u>

```
ifStat    →  if ( condition ) {p1} statements(1)  {p2}
             elsePart {p3}
elsePart  →  else statements(2)
          |  ε

{p1} :  backpatch(condition.true,nextquad())
{p2} :  ifList = makeList(nextQuad())
        genQuad('jump','_','_','_')
        backpatch(condition.false,nextquad())
{p3} :  backpatch(ifList,nextquad())
```

Το elsepart είναι προαιρετικό, κάτι που φαίνεται από το « |ε». Εξετάζεται σε δική του συνάρτηση. Εδώ, επίσης, η condition είναι μια συνθήκη που εκτελείται στην συνάρτηση *condition()* και κατά την κλήση της θα επιστρέψει είτε αληθές, είτε ψευδές. Κρατάμε το αληθές αποτέλεσμα στην λίστα *conditionsTable[0]* και το ψευδές στην λίστα *conditionsTable[1],* καθώς περιέχουν ασυμπλήρωτες τετράδες στις οποίες θα χρειαστεί να κάνουμε backpatch(). Θα γνωρίζουμε σε τι αποτιμάται η συνθήκη αμέσως μετά την κλήση της, οπότε εκεί μπαίνει το *{p1}* (είτε πριν είτε μετά την παρένθεση, δεν έχει σημασία).

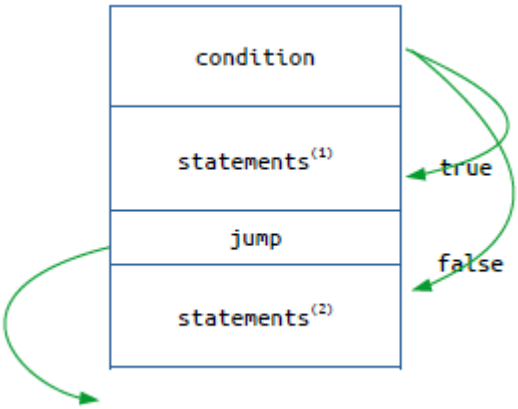
Στην περίπτωση που αυτή είναι αληθής, τότε θα ακολουθήσει η εκτέλεση των statements(1) (μέσω της *statements()*). Άρα, οι ασυμπλήρωτες τετράδες της *conditionsTable[0]* , πρέπει να συμπληρωθούν με την πρώτη

τετράδα των statements, η οποία κιόλας είναι η αμέσως επόμενη και την παίρνουμε με την nextQuad().

Στην περίπτωση, όμως, που είναι ψευδής, τα statements(1) δεν θα εκτελεστούν ποτέ καθώς βγαίνουμε εκτός της if. Τότε, εάν υπάρχει else part εκτελείται αυτό, δηλαδή εκτελούνται τα statements(2). Το backpatch() για την *conditionsTable[1]* πρέπει να γίνει πριν το else (αν αυτό υπάρχει), ή στο τέλος block της if. Αυτό γίνεται διότι αν υπάρχει else, τότε η επόμενη τετράδα που θα παραχθεί θα είναι από το statements(2) , ενώ αν δεν υπάρχει else part και συνεπώς δεν υπάρχουν και statements(2), οι τελευταίες τετράδες που θα παραχθούν θα είναι από το statements(1) και άρα το backpatch() γίνεται μετά την τελευταία τετράδα που θα παραγάγει το if. Οδηγούμαστε λοιπόν στην τοποθέτηση του *{p2}* στο τέλος των statements(1). Ανάμεσα στα statements(1), statements(2) παρεμβάλλεται jump που θα στείλει την εκτέλεση στην τετράδα μετά την τελευταία τετράδα του if. Αν δεν υπάρχει else, τότε το jump θα κάνει άλμα στην ακριβώς επόμενη εντολή μετά το τέλος block της if. Λόγω του ότι το άλμα γίνεται σε κώδικα που δεν έχει παραχθεί ακόμη, δημιουργούμε την ifList λίστα για να σημειώσουμε την μη συμπληρωμένη τετράδα, στην οποία θα κάνουμε backpatch(). Η καταλληλότερη θέση για αυτό το backpatch() είναι στο τέλος του κανόνα ifStat, ώστε να είναι η τελευταία ενέργεια που θα συμβεί. Έτσι τοποθετούμε το *{p3}* στη θέση που φαίνεται παραπάνω.

Αν το statements(1) είναι κενό τότε το backpatch() της *conditionsTable[0]* θα γίνει στην jump της ifList και ο έλεγχος θα βγει εκτός της if.

Για την καλύτερη κατανόηση παραθέτουμε εικονογραφημένα την παραγωγή ενδιάμεσου κώδικα της δομής if.



- elsepart:

```
elsePart  →  else statements(2)
          |  ε
```

Στο elsepart δεν συναντάμε κάτι καθώς έχει συμπεριληφθεί στην λογική του ifStat. Εκτελούνται κανονικά τα statements(2), εφόσον αυτά υπάρχουν και μετά ακολουθεί το *{p3}*.

- switchcaseStat:

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

{p0} :  exitList = emptyList()
{p1} :  backpatch(condition.true,nextQuad())
{p2} :  t = makeList(nextQuad)
        genQuad('jump','_','_','_')
        exitList = mergeList(exitList,t)
        backpatch(condition.false,nextQuad())
{p3} :  backpatch(exitList,nextQuad())
```

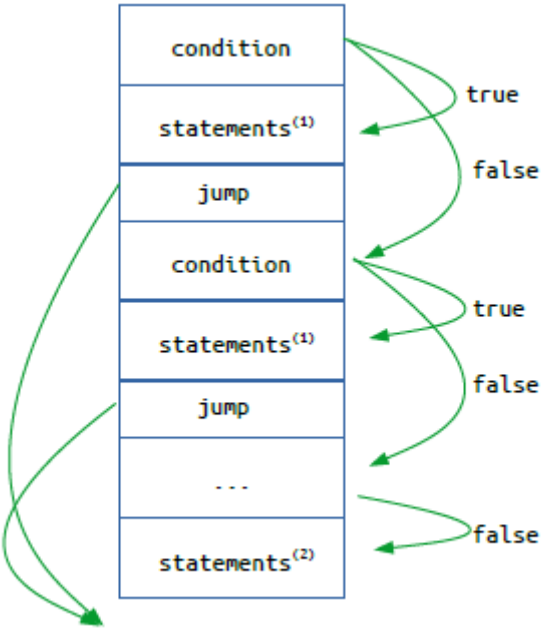
Υπάρχει μια condition, που είναι μια συνθήκη που εκτελείται στην συνάρτηση *condition()* και κατά την κλήση της θα επιστρέψει είτε αληθές, είτε ψευδές. Κρατάμε το αληθές αποτέλεσμα στην λίστα *conditionsTable[0]* και το ψευδές στην λίστα *conditionsTable[1],* καθώς περιέχουν ασυμπλήρωτες τετράδες στις οποίες θα χρειαστεί να κάνουμε backpatch(). Θα γνωρίζουμε σε τι αποτιμάται η συνθήκη αμέσως μετά την κλήση της, οπότε εκεί μπαίνει το *{p1}* (είτε πριν είτε μετά την παρένθεση, δεν έχει σημασία). Γνωρίζουμε επίσης, ότι η συνάρτηση αυτή είναι μια λούπα που συμβαίνει για όσο ισχύει η συνθήκη. Αυτό σημαίνει, ότι μετά από κάθε εκτέλεσή της, θα πρέπει να επανεξετάζεται η συνθήκη. Οδηγούμαστε, λοιπόν, στην τοποθέτηση του *{p0}* πριν την συνθήκη και εκτός του (…)* (εκτός της επανάληψης), καθώς εκεί γίνεται το άλμα σε κάθε επανάληψη της λούπας, και στην τοποθέτηση του *{p2}* στο τέλος της λούπας, καθώς τότε είναι που πρέπει να εκτελεστεί το άλμα. Όταν πάψει να ισχύει η συνθήκη, εκτελείται το default κομμάτι με τα statements(2) και στη συνέχεια βγαίνουμε εκτός της switchcaseStat.

Μένει μόνο να εξετάσουμε την συνθήκη πλέον. Στην περίπτωση που αυτή είναι αληθής, τότε θα ακολουθήσει η εκτέλεση των statements(1) (μέσω της *statements()*). Άρα, οι ασυμπλήρωτες τετράδες της *conditionsTable[0]*, πρέπει να συμπληρωθούν με την πρώτη τετράδα των statements(1), η οποία κιόλας είναι η αμέσως επόμενη και την παίρνουμε με την nextQuad(). Μετά από κάθε εκτέλεση των statements(1) χρειαζόμαστε ένα άλμα έξω από τη δομή. Όλα αυτά τα jump τα τοποθετούμε σε μια κενή λίστα exitList με σκοπό να συμπληρωθούν στο τέλος. Φτιάχνουμε ακόμα μια λίστα t με το ασυμπλήρωτο άλμα, την οποία τη συνενώνουμε με την exitList. Η συμπλήρωση της λίστας, μετά την συνένωση, γίνεται τελευταία κάνοντας backpatch(), ώστε να δείχνει στην επόμενη τετράδα μετά το τέλος block της switchcase.

Στην περίπτωση που είναι ή γίνει στην πορεία ψευδής η συνθήκη, συμπληρώνουμε τις ασυμπλήρωτες τετράδες της *conditionsTable[1]* στο *{p2}* με την αμέσως επόμενη τετράδα, που είναι αυτή των statements(2), λόγω της θέσης του *{p2}*.

Αν τα statements(1) είναι κενά, τότε το *{p1}* θα στείλει το *conditionsTable[0]* στο άλμα εκτός της δομής. Αν τα statements(2) είναι κενά, τότε ο έλεγχος θα βγει ξανά εκτός της δομής. Έχουμε δηλαδή το επιθυμητό αποτέλεσμα σε όλες τις περιπτώσεις.

Πιο παραστατικά, μπορούμε να αντιληφθούμε τον τρόπο με τον οποίο παράγεται ο ενδιάμεσος κώδικας για την switchcase με την επόμενη εικόνα.



- forcaseStat:

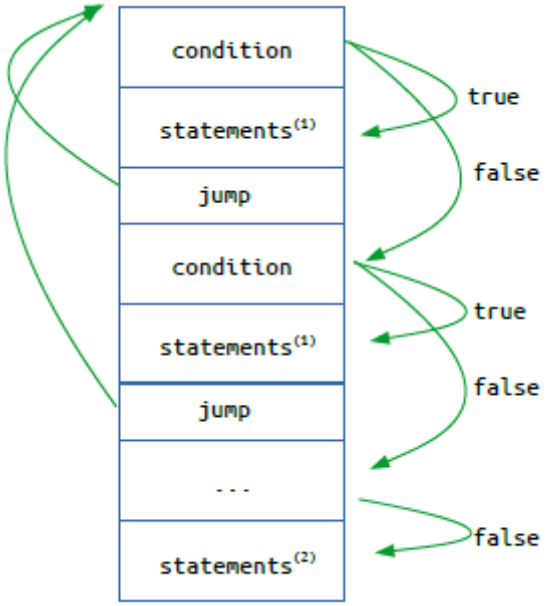
```
forcaseStat  →   forcase  {p1}
                 ( case ( condition ) {p2}
                     statements(1)  {p3} )*
                 default statements(2)

{p1} :  firstCondQuad = nextQuad()
{p2} :  backpatch(condition.true,nextQuad())
{p3} :  genQuad('jump','_','_','firstCondQuad')
        backpatch(condition.false,nextQuad())
```

Αρχικά θα εξηγήσουμε την λειτουργία της δομής. Η forcase ελέγχοντας τις συνθήκες, εκτελεί τα statements(1) μόλις βρεθεί αληθής έστω και μια condition. Στην συνέχεια ο έλεγχος ξεκινάει ξανά από την αρχή, κάτι που μας οδηγεί στο συμπέρασμα ότι χρειαζόμαστε ένα άλμα στην πρώτη τετράδα. Αν ωστόσο δεν βρεθεί καμία αληθής, εκτελείται το default part και τερματίζεται με αυτόν τον τρόπο η forcase.

Κατά συνέπεια, έχουμε να διαχειριστούμε τις δύο γνωστές λίστες *conditionsTable[0]* για «αληθές» και *conditionsTable[1]* για την αποτίμηση «ψευδές». Χρειαζόμαστε ακόμη την πρώτη τετράδα μετά την εμφάνιση της δεσμευμένης λέξης forcase, όπως προαναφέρθηκε. Την κρατάμε, λοιπόν, ονομάζοντάς την firstCondQuad. Στην συνέχεια και εφόσον έχει κληθεί η συνάρτηση *statements(),* συμπληρώνουμε τις ασυμπλήρωτες τετράδες της *conditionsTable[0]* με τις τετράδες των statements(1), καθώς το *{p2}* βρίσκεται αμέσως πριν την κλήση τους και με τη βοήθεια της nextQuad() παίρνουμε την αμέσως επόμενη τετράδα. Ο λόγος που το *{p3}* εμφανίζεται εντός της λούπας (βλέπε (…)*), είναι ότι το άλμα πρέπει να συμβαίνει κάθε φορά που μια condition είναι true. Ακολουθεί το άλμα στην τετράδα πριν την συνθήκη με χρήση της εντολής “jump” για να καλυφθεί το επαναληπτικό κομμάτι της δομής. Τέλος, συμπληρώνονται οι ασυμπλήρωτες τετράδες της *conditionsTable[1]* με την πρώτη τετράδα των statements(2), αφού το *{p3}* βρίσκεται πριν την εμφάνιση της «default» και με τη βοήθεια της nextQuad() παίρνουμε την αμέσως επόμενη τετράδα. Αυτό συμβαίνει διότι τα statements(2) εκτελούνται μόνο στην περίπτωση που όλες οι συνθήκες είναι ψευδείς και άρα απευθύνεται στην λίστα *conditionsTable[1].* Το *{p3}* εντός της λούπας δεν μας εμποδίζει, καθώς ακόμη κι αν δεν χρειαζόμαστε το άλμα προς τα πίσω, η backpatch γίνεται σωστά στην επόμενη τετράδα κάθε φορά

των statements(2) . Αν πάλι δεν υπάρχει επόμενο condition, τότε ο κώδικας που ακολουθεί το statements(1) είναι το statements(2).

Παρακάτω παραθέτουμε μια σχηματική απεικόνιση της παραγωγής ενδιάμεσου κώδικα για την forcase δομή.



- <u>incaseStat:</u>

```
incaseStat   →   incase  {p1}
                 ( case ( condition ) {p2}
                     statements(1)  {p3} )* {p4}
```

```
{p1} :  flag = newTemp()
        firstCondQuad = nextQuad()
        genQuad(':=',0,_,flag)
{p2} :  backpatch(condition.true,nextQuad())
{p3} :  genQuad(':=',1,_,flag)
        backpatch(condition.false,nextQuad())
{p4} :  genQuad('=', 1 ,flag,firstQuad)
```

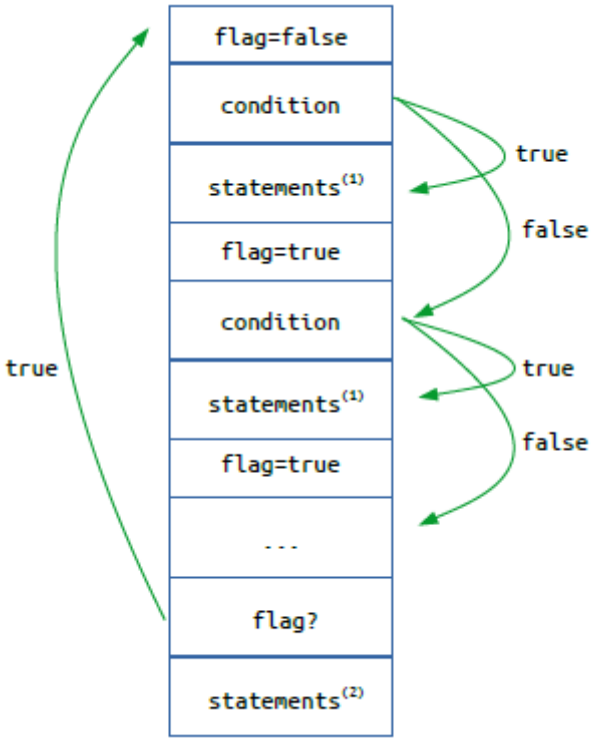
Η incase έχει επίσης επαναληπτικό χαρακτήρα. Ειδικότερα, ελέγχει μία προς μία τις συνθήκες και για όσες ισχύουν εκτελεί τα αντίστοιχα statements(1). Όταν οι συνθήκες τελειώσουν, αν έχει εκτελεστεί έστω μια από τις statements, τότε ο έλεγχος μεταβαίνει στην αρχή της incase ξανά. Αν όμως δεν έχει εκτελεστεί καμία, τότε βγαίνουμε εκτός του incase block. Αυτό μας οδηγεί να ορίσουμε το *{p1}* στο σημείο μετά την εμφάνιση της δεσμευμένης

λέξης incase, ώστε να εκτελείται εκεί το άλμα. Επίσης, θα γνωρίζουμε αν πρέπει να συμβεί το άλμα, εφόσον έχουν κληθεί οι statements(1) και ξέρουμε πλέον αν εκτελέστηκε κάποια ή όχι. Για τον λόγο αυτό, τοποθετούμε το *{p3}* στο υποφαινόμενο σημείο. Όσον αφορά τη θέση του *{p2}* χρησιμεύει στην περίπτωση που κάποια συνθήκη αποτιμηθεί αληθής, να συμπληρωθεί η ασυμπλήρωτη λίστα της τότε, κάτι που το μαθαίνουμε εφόσον έχουν διαβαστεί οι conditions.

Στην συγκεκριμένη δομή θα χρειαστούμε μια μεταβλητή, η οποία θα ελέγχει αν έστω και μία από τις statements(1) έχει εκτελεστεί. Θα την ονομάσουμε *flag* και θα χρησιμεύει στην απόφαση αν πρέπει να γυρίσουμε στην αρχή της δομής ή να βγούμε έξω από αυτήν. Η *flag* αποτιμάται σε χρόνο εκτέλεσης και όχι σε χρόνο μετάφρασης και αυτό διότι τότε γνωρίζουμε αν ισχύει τελικά μια συνθήκη. Καταλήγουμε λοιπόν στο συμπέρασμα ότι η *flag* οφείλει να είναι προσωρινή, καθώς αν δεν ήταν θα έπρεπε να οριστεί ως *flag := τιμή* και αυτό θα σήμαινε ότι παίρνει τιμή σε χρόνο μετάφρασης, μέσω του Συντακτικού Αναλυτή. Αρχικοποιούμε το *flag* σε 0 και αμέσως μετά τα statements(1) αλλάζουμε την τιμή του σε 1. Η αρχικοποίηση είναι μέρος κάθε επανάληψης. Αν, λοιπόν, το *flag* ισούται με 0, τότε πρέπει να συμβεί το άλμα, ενώ αν ισούται με 1, πρέπει να ακολουθήσει το άλμα. Το άλμα εκτελείται στην πρώτη τετράδα της πρώτης condition, πριν επαναρχικοποιηθεί το *flag*. Την κρατάμε, ονομάζοντάς την firstCondQuad.

Συναντώνται και εδώ condition, συνθήκες δηλαδή που εκτελούνται στην συνάρτηση *condition()* και κατά την κλήση τους θα επιστραφεί αληθές ή ψευδές. Κρατάμε το αληθές αποτέλεσμα στην λίστα *conditionsTable[0]* και το ψευδές στην λίστα *conditionsTable[1],* καθώς περιέχουν ασυμπλήρωτες τετράδες στις οποίες θα χρειαστεί να κάνουμε backpatch(). Στην περίπτωση που αυτή είναι αληθείς, τότε οι ασυμπλήρωτες τετράδες της *conditionsTable[0]*, πρέπει να συμπληρωθούν με την πρώτη τετράδα των statements(1), η οποία κιόλας είναι η αμέσως επόμενη και την παίρνουμε με την nextQuad(). Αν ωστόσο είναι ψευδείς, συμπληρώνουμε τις ασυμπλήρωτες τετράδες της *conditionsTable[1]* με την αμέσως επόμενη τετράδα, η οποία είναι η πρώτη τετράδα εκτός της δομής, που την παίρνουμε με την nextQuad().

Παρακάτω φαίνεται σχηματικά η παραγωγή ενδιάμεσου κώδικα για την incase δομή.



Εφόσον έχουμε τελειώσει με την παραγωγή ενδιάμεσου κώδικα, θέλουμε να σημειώσουμε ότι κατά την μετάφραση των προγραμμάτων μας, αν τα αρχεία που δίνονται ως είσοδοι δεν περιέχουν συντακτικά λάθη, τότε πρέπει να παράγονται τα αρχεία test.int, test.c. Τα αρχεία αυτά έχουν την ακόλουθη μορφή όταν τρέχουμε το πρόγραμμα με αρχείο εισόδου το *_HappyDay.ci*:

Test.int:

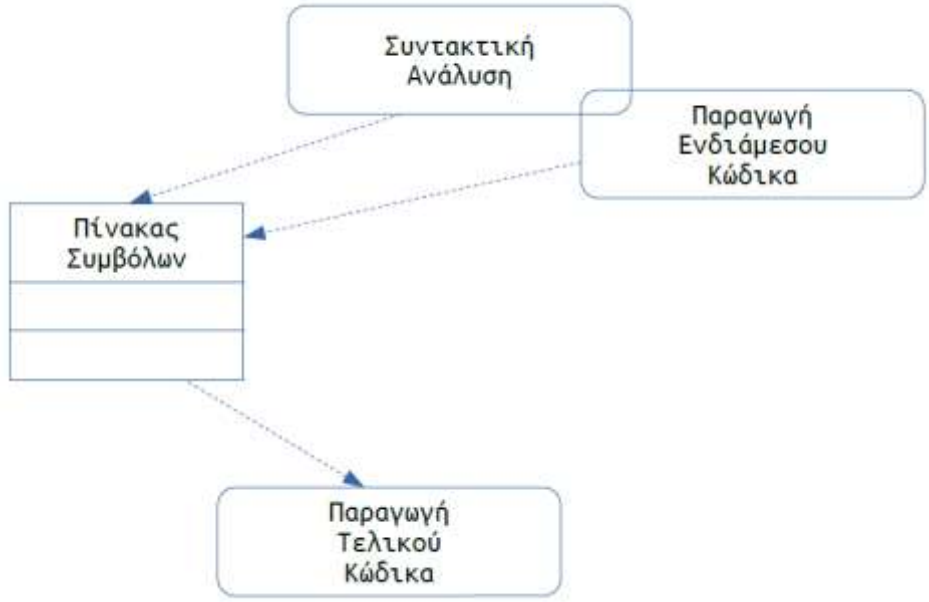
```
1, begin_block, happyDay, _, _,
2, >, x, 0, 4,
3, jump, _, _, 9,
4, >, y, z, 6,
5, jump, _, _, 9,
6, +, w, 1, T_1,
7, :=, T_1, _, w,
8, jump, _, _, 2,
9, <, z, 1, 11,
10, jump, _, _, 14,
11, +, z, 1, T_2,
12, :=, T_2, _, z,
13, jump, _, _, 9,
14, -, z, 1, T_3,
15, :=, T_3, _, z,
16, >, y, z, 18,
17, jump, _, _, 20,
18, out, y, _, _,
19, jump, _, _, 22,
20, +, y, z, T_4,
21, out, T_4, _, _,
22, :=, 0, _, T_5,
23, >=, z, y, 25,
24, jump, _, _, 28,
25, +, z, y, T_6,
26, :=, T_6, _, z,
27, :=, 1, _, T_5,
28, <, x, y, 30,
29, jump, _, _, 33,
30, -, x, y, T_7,
31, :=, T_7, _, x,
32, :=, 1, _, T_5,
33, =, T_5, 1, 22,
34, +, x, y, T_8,
35, +, T_8, z, T_9,
36, /, w, x, T_10,
37, *, T_10, y, T_11,
38, *, T_11, z, T_12,
39, *, T_12, w, T_13,
40, +, T_9, T_13, T_14,
41, retv, T_14, _, _,
42, end_block, happyDay, _, _,
43, begin_block, subfunc, _, _,
44, +, w, x, T_15,
45, retv, T_15, _, _,
46, end_block, subfunc, _, _,
47, begin_block, proc, _, _,
48, +, w, 1, T_16,
49, :=, T_16, _, w,
50, end_block, proc, _, _,
51, begin_block, happyday, _, _,
52, in, x, _, _,
53, in, y, _, _,
54, in, z, _, _,
55, in, w, _, _,
56, call, proc, _, _,
57, par, x, cv, _,
58, par, y, cv, _,
59, par, z, cv, _,
60, par, w, cv, _,
61, par, T_17, ret, _,
62, call, happyDay, _, _,
63, out, T_17, _, _,
64, halt, _, _, _,
65, end_block, happyday, _, _,
```

test.c

```
int main()
{
int x, y, z, w;
int T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_10, T_11, T_12, T_13, T_14, T_15, T_16, T_17;

L_1 :
L_2 : if (x>0) goto L_4;
L_3 : goto L_9;
L_4 : if (y>z) goto L_6;
L_5 : goto L_9;
L_6 : T_1=w+1;
L_7 : w=T_1;
L_8 : goto L_2;
L_9 : if (z<1) goto L_11;
L_10 : goto L_14;
L_11 : T_2=z+1;
L_12 : z=T_2;
L_13 : goto L_9;
L_14 : T_3=z-1;
L_15 : z=T_3;
L_16 : if (y>z) goto L_18;
L_17 : goto L_20;
L_18 : printf("y = %d", y);
L_19 : goto L_22;
L_20 : T_4=y+z;
L_21 : printf("T_4 = %d", T_4);
L_22 : T_5=0;
L_23 : if (z>=y) goto L_25;
L_24 : goto L_28;
L_25 : T_6=z+y;
L_26 : z=T_6;
L_27 : T_5=1;
L_28 : if (x<y) goto L_30;
L_29 : goto L_33;
L_30 : T_7=x-y;
L_31 : x=T_7;
L_32 : T_5=1;
L_33 : if (T_5==1) goto L_22;
L_34 : T_8=x+y;
```

```
L_35 : T_9=T_8+z;
L_36 : T_10=w/x;
L_37 : T_11=T_10*y;
L_38 : T_12=T_11*z;
L_39 : T_13=T_12*w;
L_40 : T_14=T_9+T_13;
L_41 :
L_42 :
L_43 :
L_44 : T_15=w+x;
L_45 :
L_46 :
L_47 :
L_48 : T_16=w+1;
L_49 : w=T_16;
L_50 :
L_51 :
L_52 :
L_53 :
L_54 :
L_55 :
L_56 :
L_57 :
L_58 :
L_59 :
L_60 :
L_61 :
L_62 :
L_63 : printf("T_17 = %d", T_17);
L_64 : {}

}
```

ΠΙΝΑΚΑΣ ΣΥΜΒΟΛΩΝ

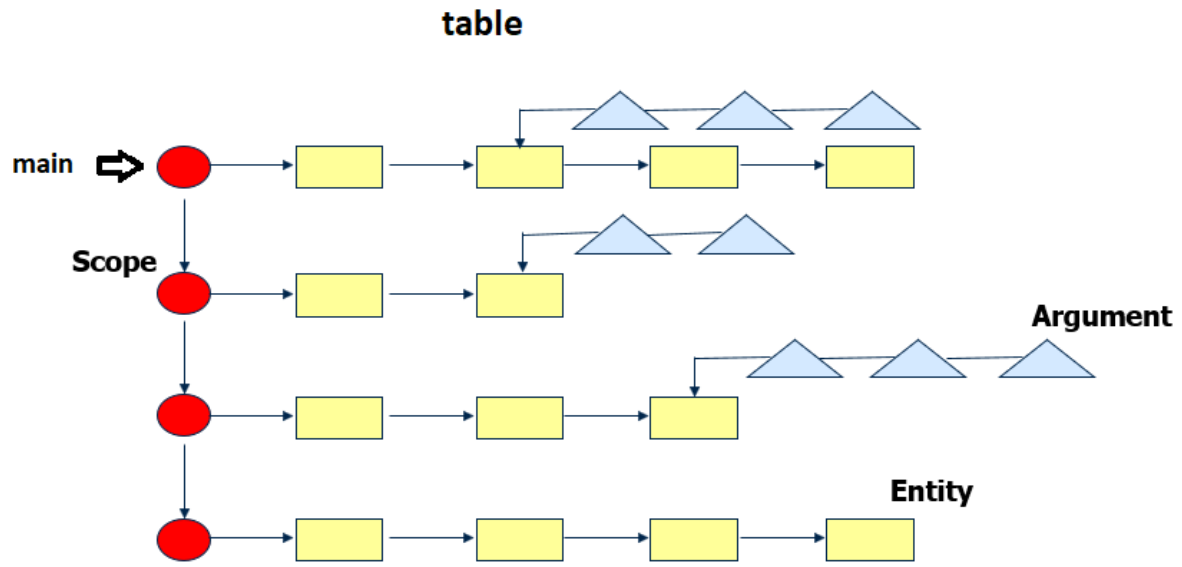


Ο πίνακας συμβόλων είναι μια δυναμική δομή στην οποία αποθηκεύεται πληροφορία σχετιζόμενη με τα συμβολικά ονόματα που χρησιμοποιούνται στο πρόγραμμα που θέλουμε να μεταγλωττίσουμε. Η δομή αυτή μεταβάλλεται δυναμικά, προσθέτοντας ή αφαιρώντας πληροφορία σύμφωνα με τους κανόνες εμβέλειας της γλώσσας. Έτσι, σε διαφορετικές στιγμές της μεταγλώττισης μπορεί να υπάρχουν διαφορετικές εγγραφές στον πίνακα συμβόλων. Οι εγγραφές που αποθηκεύονται διακρίνονται σε:

- Μεταβλητές
- Προσωρινές Μεταβλητές
- Παράμετροι Συναρτήσεων ή Διαδικασιών
- Συναρτήσεις ή Διαδικασίες

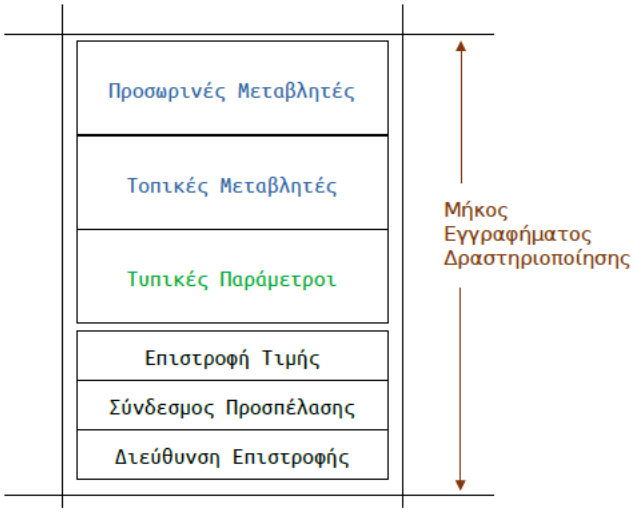
Όλη η πληροφορία για τη δημιουργία του πίνακα συμβόλων ανακτάται κατά τη φάση της Συντακτικής Ανάλυσης και της παραγωγής ενδιάμεσου κώδικα. Η πληροφορία αυτή είναι χρήσιμη για την εκσφαλμάτωση της διαδικασίας της μεταγλώττισης και θα χρησιμοποιηθεί κατά κόρον στη φάση παραγωγής του τελικού κώδικα.

ΔΟΜΗ ΠΙΝΑΚΑ ΣΥΜΒΟΛΩΝ:



Ο πίνακας συμβόλων (*table*) αποτελείται από επίπεδα (*scopes*). Τα επίπεδα δημιουργούνται όταν ξεκινάει η μετάφραση μιας συνάρτησης ή διαδικασίας, ενώ αφαιρούνται, όταν τερματίζεται. Τα νέα επίπεδα προστίθενται στον πίνακα συμβόλων με τη λογική της στοίβας, δηλαδή πάνω στο επίπεδο που μέχρι στιγμής έχει δημιουργηθεί. Αντίστοιχα, όταν αφαιρούμε ένα επίπεδο διαγράφουμε ταυτόχρονα μαζί με το επίπεδο και όλες του τις εγγραφές (*Entities*). Πάντα στον πίνακα συμβόλων το «κατώτερο» επίπεδο αντιστοιχεί στο κυρίως πρόγραμμα (*main*).

ΕΓΓΡΑΦΗΜΑ ΔΡΑΣΤΗΡΙΟΠΟΙΗΣΗΣ:



Το Εγγράφημα Δραστηριοποίησης δημιουργείται για κάθε συνάρτηση ή διαδικασία από αυτή που την καλεί και μόλις ολοκληρωθεί η εκτέλεσή της καταστρέφεται. Πρόκειται για χώρο όπου παραχωρείται στο υποπρόγραμμα αυτό, δίνοντάς του την δυνατότητα να αποθηκεύσει πληροφορίες για την εκτέλεσή του και τον τερματισμό του καθώς και μεταβλητές που χρησιμοποιεί, όπως είναι οι παράμετροί του , οι προσωρινές μεταβλητές του αλλά και οι τοπικές μεταβλητές του. Όταν τερματίζεται ένα subprogram ο χώρος που καταλαμβάνει το Εγγράφημα επιστρέφεται στο σύστημα.

Είναι σημαντικό να σημειωθεί ότι στον επεξεργαστή που θα χρησιμοποιήσουμε στην συνέχεια της μεταγλώττισης (παραγωγή τελικού κώδικα) μία διεύθυνση μνήμης είναι 4 bytes. Επίσης, ξέρουμε ότι ο μοναδικός τύπος δεδομένων στην C-imple είναι ο ακέραιος, όπου και αυτός είναι 4 bytes. Άρα μπορούμε να πούμε με σιγουριά ότι κάθε θέση στο Εγγράφημα Δραστηριοποίησης θα αποτελείται από 4 bytes.

Πιο αναλυτικά:

- Η Διεύθυνση Επιστροφής της συνάρτησης αποθηκεύεται στην πρώτη θέση του Εγγραφήματος Δραστηριοποίησης και καταλαμβάνει 4 bytes. Είναι η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης. Περισσότερες λεπτομέρειες για το πώς θα αξιοποιηθεί αυτή η πληροφορία βρίσκονται ενότητα παραγωγής Τελικού Κώδικα.
- Ο Σύνδεσμος Προσπέλασης αποθηκεύεται στην δεύτερη θέση του Εγγραφήματος Δραστηριοποίησης και καταλαμβάνει 4 bytes. Συγκεκριμένα, είναι ένας δείκτης όπου δείχνει στο υποπρόγραμμα (Εγγράφημα Δραστηριοποίησης) που πρέπει να αναζητηθεί από την συνάρτηση ή την διαδικασία μια μεταβλητή ή παράμετρος η οποία δεν βρίσκεται στο δικό της Εγγράφημα Δραστηριοποίησης. Αυτό έγκειται στους κανόνες εμβέλειας της γλώσσας C-imple, όπου κάθε υποπρόγραμμα έχει πρόσβαση στις δικές του μεταβλητές και παραμέτρους, στις τοπικές μεταβλητές και παραμέτρους του γονέα του και σε οποιοδήποτε άλλο υποπρόγραμμα ανήκει στο γενεαλογικό του δέντρο, συμπεριλαμβανομένων και των καθολικών μεταβλητών.
- Η Επιστροφή Τιμής της συνάρτησης (οι διαδικασίες στην C-imple δεν επιστρέφουν κάποια τιμή οπότε η θέση αυτή μένει κενή) αποθηκεύεται στην τρίτη θέση του Εγγραφήματος Δραστηριοποίησης και

καταλαμβάνει 4 bytes. Εδώ θα αποθηκευτεί η διεύθυνση της μεταβλητής στην οποία θα επιστραφεί η τιμή της συνάρτησης.

- Οι Παράμετροι τοποθετούνται στη στοίβα με την σειρά εμφάνισής τους κατά την κλήση του υποπρογράμματος καταλαμβάνοντας 4 bytes η κάθε μία. Αν πρόκειται για πέρασμα με τιμή αποθηκεύεται η τιμή, αλλιώς αποθηκεύεται η διεύθυνση στο πέρασμα με αναφορά.
- Τέλος, στο Εγγράφημα Δραστηριοποίησης αποθηκεύονται οι Τοπικές Μεταβλητές και οι Προσωρινές Μεταβλητές με την σειρά εμφάνισής τους κατά την μεταγλώττιση του προγράμματος. Καταλαμβάνουν και αυτές 4 bytes η κάθε μία.

Αξιοσημείωτα χαρακτηριστικά του Εγγραφήματος Δραστηριοποίησης:

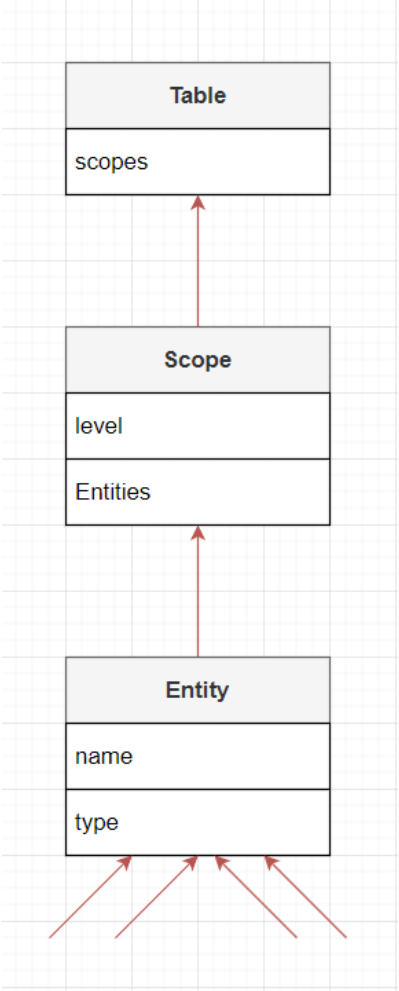
- Μήκος Εγγραφήματος Δραστηριοποίησης : Πρόκειται για τον συνολικό χώρο σε bytes που καταλαμβάνει το Εγγράφημα Δραστηριοποίησης. Ισούται με: *12 + (εγγραφές) * 4*, όπου ο αριθμός 12 αντιστοιχεί στις 3 πρώτες πληροφορίες (3*4=12) για την εκτέλεση και το τερματισμό του υποπρογράμματος , ενώ ο όρος εγγραφές αντιστοιχεί στο πλήθος των υπολοίπων εγγραφών που υπάρχουν στο συγκεκριμένο υποπρόγραμμα. Πολλαπλασιάζουμε αυτό το πλήθος με 4 γιατί κάθε θέση στο Εγγράφημα Δραστηριοποίησης καταλαμβάνει 4 bytes.

- Απόσταση από την Αρχή του Εγγραφήματος Δραστηριοποίησης: Πρόκειται για την απόσταση σε bytes από την αρχή του Εγγραφήματος Δραστηριοποίησης.

ΕΓΓΡΑΦΕΣ ΤΟΥ ΠΙΝΑΚΑ ΣΥΜΒΟΛΩΝ ΣΕ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΕΦΗ ΣΧΕΔΙΑΣΗ:

Στην υλοποίηση σε κώδικα, ορίστηκε μία κλάση *Table()* η οποία περιέχει μοναδικό πεδίο ένα πίνακα(*scopes[]*) από αντικείμενα *Scope()*. Το αντικείμενο *Table()* αντιστοιχεί στον πίνακα συμβόλων με τον πίνακα από objects τύπου *Scope()* να δηλώνει τα επίπεδα που θα έχουμε σε κάθε φάση της μετάφρασης. Έτσι, όταν χρειάζεται να προσθέσουμε / αφαιρέσουμε επίπεδα από τον πίνακα συμβόλων ουσιαστικά προσθέτουμε στην τελευταία θέση / αφαιρούμε το

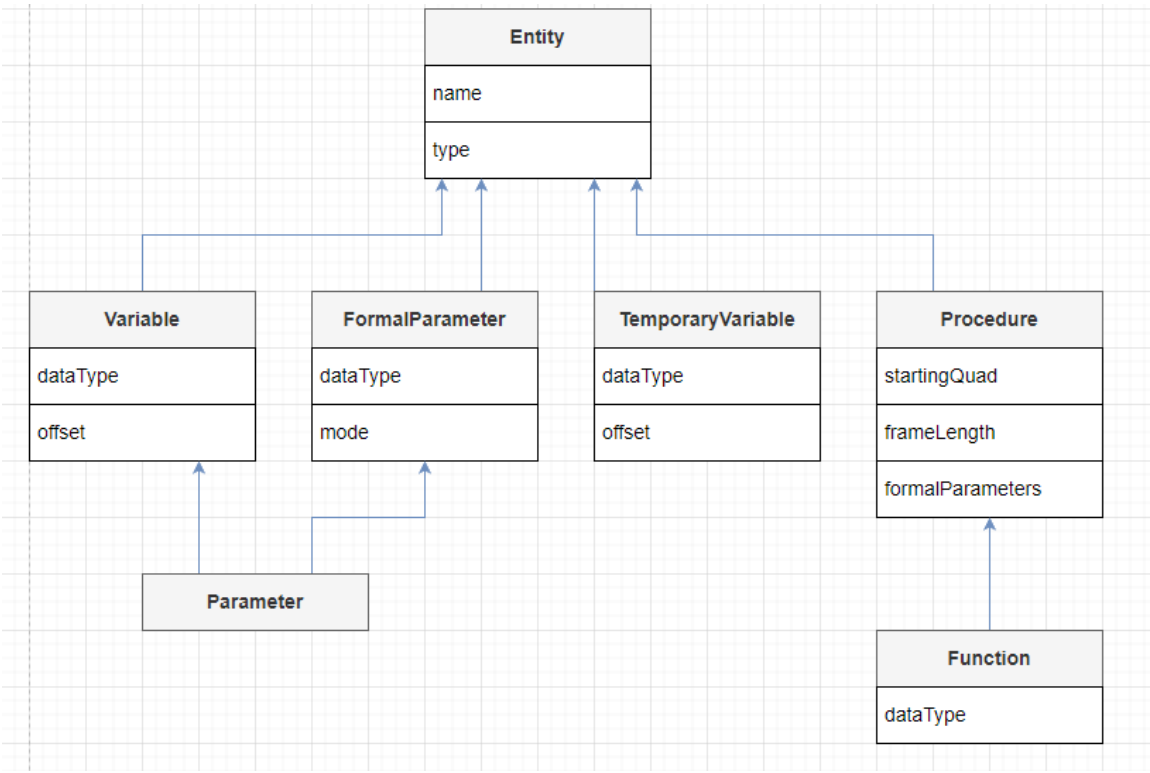
τελευταίο στοιχείο του *scopes[]*. Με αυτόν τον τρόπο πετυχαίνουμε τη λογική της στοίβας κατά την δημιουργία / διαγραφή επιπέδων.



Η κλάση *Scope()* μοντελοποιεί την έννοια του επιπέδου. Συγκεκριμένα, αποτελείται από 2 χαρακτηριστικά , το *level* και το *Entities.* Το πεδίο *level* το χρειαζόμαστε για να κρατάμε τον αύξοντα αριθμό του κάθε επιπέδου, ενώ το πεδίο *Entities* είναι μια λίστα όπου εκεί αποθηκεύονται αντικείμενα τύπου *Entity()* .

Κάθε εγγραφή (Entity) που αποθηκεύεται στη δομή διατηρεί διαφορετική πληροφορία ανάλογα με το είδος του συμβολικού ονόματος. Όμως, παρατηρήθηκε ότι όλες οι εγγραφές έχουν 2 ίδια χαρακτηριστικά. Έτσι, στην κλάση *Entity()* ορίσαμε 2 πεδία που αντιστοιχούν στο όνομα (*name*) και στον τύπο (*type*) κάθε εγγραφής (αν είναι μεταβλητή, τυπική παράμετρος, κλπ.).

Παρόλα αυτά, παρατηρήθηκαν και άλλα παρόμοια χαρακτηριστικά μεταξύ κλάσεων οπότε η σχεδίαση πήρε την μορφή που φαίνεται παρακάτω. Στην δημιουργία των κλάσεων εκμεταλλευτήκαμε την ιεραρχική σχεδίαση και την κληρονομικότητα.



Ανάλογα τον τύπο της εγγραφής (Entity), αποθηκεύουμε σε κάθε κλάση τις εξής πληροφορίες:

- **Variable** (Μεταβλητή):
  - name: Το όνομα της μεταβλητής.
  - type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “Var”.
  - dataType: Τον τύπο δεδομένων της μεταβλητής.
  - offset: Την απόσταση της μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης.

- **FormalParameter** (Τυπική Παράμετρος) :
  - name: Το όνομα της τυπικής παραμέτρου.
  - type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “formalParameter”.
  - dataType: Τον τύπο δεδομένων της τυπικής παραμέτρου.

- mode: Τον τρόπο περάσματος της τυπικής παραμέτρου. Οι τιμές που μπορεί να πάρει αυτό το πεδίο είναι “ref” αν έχουμε πέρασμα με αναφορά και “cv” αν έχουμε πέρασμα με τιμή.

❖ **Parameter** (Παράμετρος) :

- name: Το όνομα της παραμέτρου.
- type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “Parameter”.
- dataType: Τον τύπος δεδομένων της παραμέτρου.
- mode: Τον τρόπο περάσματος της παραμέτρου. Οι τιμές που μπορεί να πάρει αυτό το πεδίο είναι “ref” αν έχουμε πέρασμα με αναφορά και “cv” αν έχουμε πέρασμα με τιμή.
- offset: Την απόσταση της παραμέτρου από την αρχή του εγγραφήματος δραστηριοποίησης.

❖ **TemporaryVariable** (Προσωρινή Μεταβλητή):

- name: Το όνομα της προσωρινής μεταβλητής.
- type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “TempVar”.
- dataType: Τον τύπο δεδομένων της μεταβλητής.
- offset: Την απόσταση της προσωρινής μεταβλητής από την αρχή του εγγραφήματος δραστηριοποίησης.

❖ **Procedure** (Διαδικασία):

- name: Το όνομα της διαδικασίας.
- type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “procedure”.
- startingQuad: Την ετικέτα της πρώτης εκτελέσιμης τετράδας της διαδικασίας. Έτσι, σημειώνουμε την τεράδα που θα πρέπει το

καλόν υποπρόγραμμα να κάνει άλμα προκειμένου να αρχίσει η εκτέλεση της κληθείσας.

- frameLength: Το μήκος του εγγραφήματος δραστηριοποίησης (σε bytes).
- formalParameters: Τη λίστα με τις τυπικές παραμέτρους της διαδικαίας. Στη δομή αυτή αποθηκεύονται αντικείμενα τυπου *FormalParameter()* που είναι οι παράμετροι του υποπρογράμματος. Η αποθήκευση γίνεται με την σειρά που δηλώθηκαν στο πρόγραμμα προς μετάφραση. Ο λόγος που διατηρούμε τη λίστα με τις τυπικές παραμέτρους είναι για να μπορέσουμε να ελέγξουμε, αργότερα, ότι μια διαδικασία κλήθηκε όπως ακριβώς δηλώθηκε.

❖ **Function** (Συνάρτηση):

- name: Το όνομα της συνάρτησης.
- type: String που την χαρακτηρίζει, στην συγκεκριμένη περίπτωση το “function”.
- startingQuad: Την ετικέτα της πρώτης εκτελέσιμης τετράδας της συνάρτησης. Έτσι, σημειώνουμε την τεράδα που θα πρέπει το καλών υποπρόγραμμα να κάνει άλμα προκειμένου να αρχίσει η εκτέλεση της κληθείσας.
- frameLength: Το μήκος του εγγραφήματος δραστηριοποίησης (σε bytes).
- formalParameters: Τη λίστα με τις τυπικές παραμέτρους της συνάρτησης. Στη δομή αυτή αποθηκεύονται αντικείμενα τυπου *FormalParameter()* που είναι οι παράμετροι του υποπρογράμματος. Η αποθήκευση γίνεται με την σειρά που δηλώθηκαν στο πρόγραμμα προς μετάφραση. Ο λόγος που διατηρούμε τη λίστα με τις τυπικές παραμέτρους είναι για να μπορέσουμε να ελέγξουμε , αργότερα, ότι μια διαδικασία κλήθηκε όπως ακριβώς δηλώθηκε.
- dataType: Τον τύπο δεδομένων που επιστρέφει η συνάρτηση. Στην διαδικασία δεν περιλαμβάνεται αυτό το πεδίο επειδή δεν επιστρέφει κάποια τιμή σε αυτόν που την κάλεσε.

Πρέπει να σημειωθεί ότι στην C-imple έχουμε μόνο έναν τύπο δεδομένων, τους ακεραίους (Integers). Άρα, οι κλάσεις που έχουν ως πεδίο το *dataType,* το συγκεκριμένο πεδίο έχει αρχικοποιηθεί σε όλες τις περιπτώσεις σε «int».

ΛΕΙΤΟΥΡΓΙΕΣ ΠΙΝΑΚΑ ΣΥΜΒΟΛΩΝ:

Σκοπός αυτών των λειτουργιών είναι το στιγμιότυπο του πίνακα συμβόλων, σε κάθε σημείο της παραγωγής του τελικού κώδικα, να περιέχει μόνο εκείνες τις εγγραφές (Entities) που το μεταφραζόμενο πρόγραμμα έχει πρόσβαση. Οι εγγραφές όπου ο τελικός κώδικας έχει δικαίωμα να προσπελάσει σε συγκεκριμένη χρονική στιγμή βασίζονται στους κανόνες της γλώσσας C-imple.

- Πρόσθεση Τυπικής Παραμέτρου (addFormalParameter()): Η καινούρια τυπική παράμετρος προστίθεται στη λίστα (*formalParameters[]*) με τις τυπικές παραμέτρους της συνάρτησης ή της διαδικασίας πάντα στο ανώτερο level που βρισκόμαστε. Η λίστα αυτή είναι πεδίο του object *Procedure.* Αυτή η συνάρτηση καλείται όταν συναντάμε δήλωση τυπικών παραμέτρων (αλλιώς arguments) σε ένα υποπρόγραμμα.
- Πρόσθεση Νέας Εγγραφής (addEntity()): Η καινούρια εγγραφή προστίθεται στην τελευταία θέση του πίνακα συμβόλων στο ανώτερο επίπεδο (*topScope*). Αυτή η συνάρτηση καλείται στις περιπτώσεις που συναντάμε δημιουργία προσωρινής μεταβλητής (*Temporary Variable*), δήλωση παραμέτρων συνάρτησης ή διαδικασίας (*Formal Parameters*), δήλωση μεταβλητής (Variable, ανεξαρτήτως αν βρίσκεται σε υποπρόγραμμα ή στην main) και στην δήλωση διαδικασιών ή συναρτήσεων (Procedures or Functions).
- Προσθήκη Νέου Επιπέδου (addLevel()): Καλείται μόνο όταν ξεκινάει η μετάφραση της main ή υποπρογραμμάτων (procedures or functions). Ακολουθεί την λογικής της στοίβας, προσθέτοντας το νέο επίπεδο πάνω από το ήδη υπάρχων. Κάθε φορά που δημιουργείται ένα καινούριο επίπεδο αυξάνουμε το πεδίο Level κατά 1 σε σχέση με το level που έχει το προηγούμενό του. Ειδική περίπτωση αποτελεί το επίπεδο που αντιστοιχεί στο κυρίως πρόγραμμα το οποίο έχει τιμή 0 στο level και θα είναι πάντα το πρώτο που δημιουργείται.
- Αφαίρεση Επιπέδου (deleteLevel()): Καλείται όταν ολοκληρώνεται η μετάφραση ενός υποπρογράμματος ή γενικά του κυρίως προγράμματος.

Συγκεκριμένα, διαγράφουμε το επίπεδο με το υψηλότερο level μαζί με όλες του τις εγγραφές.

- Ενημέρωση Πεδίων (updateFields()): Η χρησιμότητα αυτής της συνάρτησης είναι για να προσθέσουμε πληροφορία στον πίνακα συμβόλων η οποία δεν ήταν διαθέσιμη κατά την δημιουργία Entity. Εμβαθύνοντας, αναφερόμαστε στα πεδία *frameLength* (Μήκος Εγγραφήματος Δραστηριοποίησης) και *startingQuad* της κλάσης *Procedure.* Για δική μας διευκόλυνση, ορίσαμε και μια συνάρτηση *computeOffset(),* η οποία υπολογίζει το μήκος του Εγγραφήματος αναλόγως με το πόσες εγγραφές έχει σύμφωνα πάντα με τον κανόνα που ορίσαμε παραπάνω. Το *frameLength* ενός υποπρογράμματος ή του κυρίως προγράμματος το μαθαίνουμε όταν ολοκληρωθεί η μετάφρασή του. Αντίστοιχα, το *startingQuad* ενός υποπρογράμματος ή του κυρίως προγράμματος το μαθαίνουμε μόλις μεταφραστεί η πρώτη του εντολή.
- Αναζήτηση Εγγραφής (searchEntity()): Θα χρειαστεί κατά την παραγωγή του τελικού κώδικα παρακάτω. Αυτή η συνάρτηση αναζητάει πληροφορία στον πίνακα συμβόλων. Συγκεκριμένα, ξεκινώντας την αναζήτηση από το ανώτερο επίπεδο και οδεύοντας προς το κατώτερο μέχρι να βρει την συγκεκριμένη εγγραφή. Στην περίπτωση που την βρει, επιστρέφει και το επίπεδο που βρέθηκε το συγκεκριμένο Entity αλλά και το Entity αυτό. Σε κάθε άλλη περίπτωση, η συνάρτηση επιστρέφει μήνυμα λάθους τερματίζοντας την διαδικασία της μεταγλώττισης.

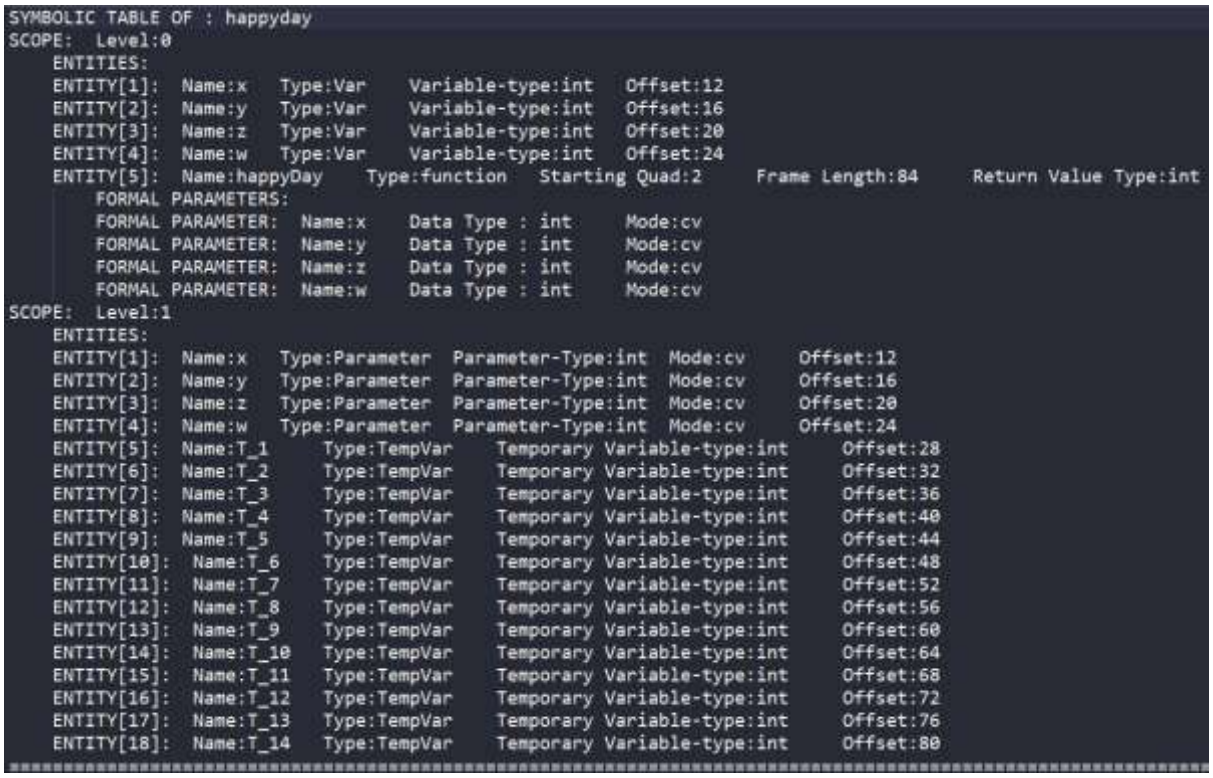
Με τον μηχανισμό της αναζήτησης μιας εγγραφής υλοποιούμε και την υπερκάλυψη μεταβλητών. Έτσι, οι τοπικές μεταβλητές και οι παράμετροι που ανήκουν σε ένα υποπρόγραμμα υπό μετάφραση, έχουν προτεραιότητα σε σχέση με τις αντίστοιχες εγγραφές που έχουν δηλωθεί στον γονέα. Ακόμα, και αυτά τα Entities του γονέα έχουν προτεραιότητα σε σχέση τα Entities που βρίσκονται σε κάποιο άλλον πρόγονο.

Παρακάτω, παραθέτουμε μερικές ακόμα συναρτήσεις που αποτέλεσαν αρωγό στην δημιουργία του Πίνακα Συμβόλων:

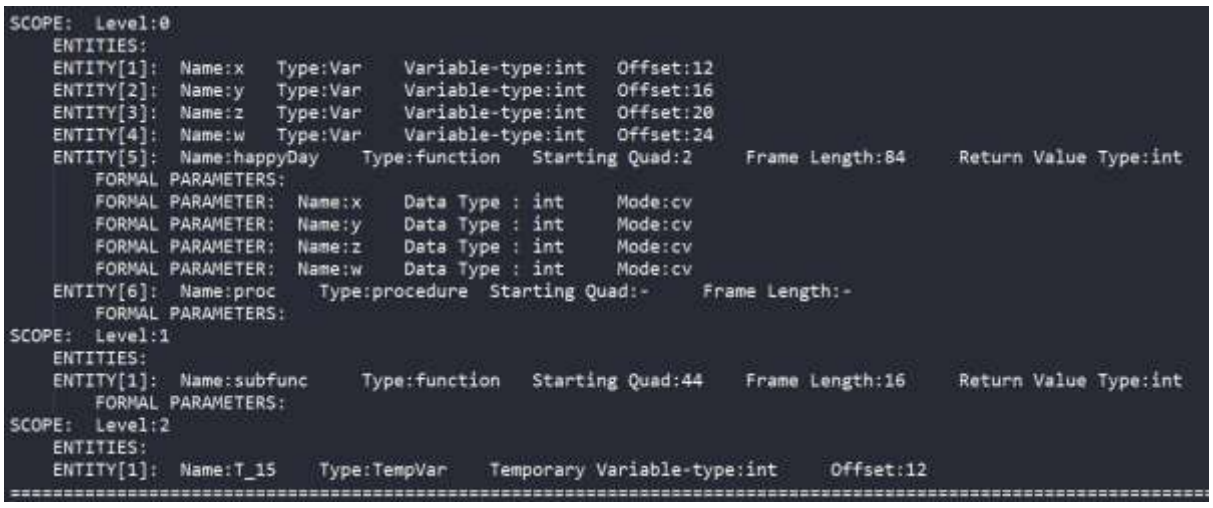
- addParametersToNewEntity(): Καλείται όταν προστίθεται ένα καινούριο επίπεδο μόνο όταν ξεκινάει η μετάφραση ενός υποπρογράμματος (procedure or function). Συγκεκριμένα, δημιουργεί Entities για όλες τις παραμέτρους του υποπρογράμματος αυτού με την σειρά εμφάνισής τους.

- writeSymbolicTable(): Συνάρτηση η οποία γράφει σε ένα αρχείο τύπου «.symb» το στιγμιότυπο του πίνακα συμβόλων πριν από κάθε close scope (κλήση συνάρτησης *deleteScope()*).

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ: Αποτέλεσμα αρχείου test.symb με αρχείο εισόδου *_HappyDay.ci. :*



```
SYMBOLIC TABLE OF : happyday
SCOPE:  Level:0
    ENTITIES:
    ENTITY[1]:  Name:x   Type:Var    Variable-type:int   Offset:12
    ENTITY[2]:  Name:y   Type:Var    Variable-type:int   Offset:16
    ENTITY[3]:  Name:z   Type:Var    Variable-type:int   Offset:20
    ENTITY[4]:  Name:w   Type:Var    Variable-type:int   Offset:24
    ENTITY[5]:  Name:happyDay    Type:function   Starting Quad:2     Frame Length:84     Return Value Type:int
        FORMAL PARAMETERS:
        FORMAL PARAMETER:  Name:x    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:y    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:z    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:w    Data Type : int     Mode:cv
SCOPE:  Level:1
    ENTITIES:
    ENTITY[1]:  Name:x   Type:Parameter  Parameter-Type:int  Mode:cv     Offset:12
    ENTITY[2]:  Name:y   Type:Parameter  Parameter-Type:int  Mode:cv     Offset:16
    ENTITY[3]:  Name:z   Type:Parameter  Parameter-Type:int  Mode:cv     Offset:20
    ENTITY[4]:  Name:w   Type:Parameter  Parameter-Type:int  Mode:cv     Offset:24
    ENTITY[5]:  Name:T_1     Type:TempVar    Temporary Variable-type:int     Offset:28
    ENTITY[6]:  Name:T_2     Type:TempVar    Temporary Variable-type:int     Offset:32
    ENTITY[7]:  Name:T_3     Type:TempVar    Temporary Variable-type:int     Offset:36
    ENTITY[8]:  Name:T_4     Type:TempVar    Temporary Variable-type:int     Offset:40
    ENTITY[9]:  Name:T_5     Type:TempVar    Temporary Variable-type:int     Offset:44
    ENTITY[10]:  Name:T_6    Type:TempVar    Temporary Variable-type:int     Offset:48
    ENTITY[11]:  Name:T_7    Type:TempVar    Temporary Variable-type:int     Offset:52
    ENTITY[12]:  Name:T_8    Type:TempVar    Temporary Variable-type:int     Offset:56
    ENTITY[13]:  Name:T_9    Type:TempVar    Temporary Variable-type:int     Offset:60
    ENTITY[14]:  Name:T_10   Type:TempVar    Temporary Variable-type:int     Offset:64
    ENTITY[15]:  Name:T_11   Type:TempVar    Temporary Variable-type:int     Offset:68
    ENTITY[16]:  Name:T_12   Type:TempVar    Temporary Variable-type:int     Offset:72
    ENTITY[17]:  Name:T_13   Type:TempVar    Temporary Variable-type:int     Offset:76
    ENTITY[18]:  Name:T_14   Type:TempVar    Temporary Variable-type:int     Offset:80
================================================================================================================
```



```
SCOPE:  Level:0
    ENTITIES:
    ENTITY[1]:  Name:x   Type:Var    Variable-type:int   Offset:12
    ENTITY[2]:  Name:y   Type:Var    Variable-type:int   Offset:16
    ENTITY[3]:  Name:z   Type:Var    Variable-type:int   Offset:20
    ENTITY[4]:  Name:w   Type:Var    Variable-type:int   Offset:24
    ENTITY[5]:  Name:happyDay    Type:function   Starting Quad:2     Frame Length:84     Return Value Type:int
        FORMAL PARAMETERS:
        FORMAL PARAMETER:  Name:x    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:y    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:z    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:w    Data Type : int     Mode:cv
    ENTITY[6]:  Name:proc    Type:procedure  Starting Quad:-     Frame Length:-
        FORMAL PARAMETERS:
SCOPE:  Level:1
    ENTITIES:
    ENTITY[1]:  Name:subfunc     Type:function   Starting Quad:44    Frame Length:16     Return Value Type:int
        FORMAL PARAMETERS:
SCOPE:  Level:2
    ENTITIES:
    ENTITY[1]:  Name:T_15    Type:TempVar    Temporary Variable-type:int     Offset:12
================================================================================================================
```

```
SCOPE:  Level:0
    ENTITIES:
    ENTITY[1]:  Name:x   Type:Var    Variable-type:int   Offset:12
    ENTITY[2]:  Name:y   Type:Var    Variable-type:int   Offset:16
    ENTITY[3]:  Name:z   Type:Var    Variable-type:int   Offset:20
    ENTITY[4]:  Name:w   Type:Var    Variable-type:int   Offset:24
    ENTITY[5]:  Name:happyDay    Type:function   Starting Quad:2     Frame Length:84     Return Value Type:int
        FORMAL PARAMETERS:
        FORMAL PARAMETER:  Name:x    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:y    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:z    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:w    Data Type : int     Mode:cv
    ENTITY[6]:  Name:proc    Type:procedure  Starting Quad:48    Frame Length:16
        FORMAL PARAMETERS:
SCOPE:  Level:1
    ENTITIES:
    ENTITY[1]:  Name:subfunc     Type:function   Starting Quad:44    Frame Length:16     Return Value Type:int
        FORMAL PARAMETERS:
    ENTITY[2]:  Name:T_16    Type:TempVar    Temporary Variable-type:int     Offset:12
================================================================================================================
SCOPE:  Level:0
    ENTITIES:
    ENTITY[1]:  Name:x   Type:Var    Variable-type:int   Offset:12
    ENTITY[2]:  Name:y   Type:Var    Variable-type:int   Offset:16
    ENTITY[3]:  Name:z   Type:Var    Variable-type:int   Offset:20
    ENTITY[4]:  Name:w   Type:Var    Variable-type:int   Offset:24
    ENTITY[5]:  Name:happyDay    Type:function   Starting Quad:2     Frame Length:84     Return Value Type:int
        FORMAL PARAMETERS:
        FORMAL PARAMETER:  Name:x    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:y    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:z    Data Type : int     Mode:cv
        FORMAL PARAMETER:  Name:w    Data Type : int     Mode:cv
    ENTITY[6]:  Name:proc    Type:procedure  Starting Quad:48    Frame Length:16
        FORMAL PARAMETERS:
    ENTITY[7]:  Name:T_17    Type:TempVar    Temporary Variable-type:int     Offset:28
================================================================================================================
```

ΤΕΛΙΚΟΣ ΚΩΔΙΚΑΣ

Αποτελεί την τελευταία φάση της μεταγλώττισης η οποία στο τέλος παράγει τον τελικό κώδικα σε γλώσσα μηχανής. Για τη C-imple θα παράγουμε τελικό κώδικα σε συμβολική γλώσσα μηχανής (*assembly code*) του επεξεργαστή RISC-V. Από κάθε γραμμή ενδιάμεσου κώδικα (τετράδα), προκύπτει μία σειρά εντολών τελικού κώδικα, η οποία για να παραχθεί δέχεται πληροφορίες από τον πίνακα συμβόλων.

ΚΑΤΑΧΩΡΗΤΕΣ ΕΠΕΞΕΡΓΑΣΤΗ RISC-V:

Οι τιμές στους καταχωρητές αποτιμώνται με τρείς διαφορετικούς τρόπους:

a) εκχώρηση αριθμητικής σταθεράς
b) μεταφορά τιμής από άλλον καταχωρητή
c) εκτέλεση αριθμητικής πράξης

Οι διαθέσιμοι καταχωρητές του RISC-V για ακέραιους που χρειαζόμαστε είναι οι ακόλουθοι:

- zero: καταχωρητής που έχει πάντα την τιμή 0
- sp: δείχνει στη στοίβα και χρησιμοποιείται για να σημειώνει την αρχή του Εγγραφήματος δραστηριοποίησης κάθε συνάρτησης / διαδικασίας τη στιγμή που αυτή εκτελείται
- fp: δείχνει στη στοίβα και χρησιμοποιείται για να σημειώνει την αρχή του Εγγραφήματος δραστηριοποίησης κάθε συνάρτησης / διαδικασίας τη στιγμή που αυτή δημιουργείται
- t0-t6: προσωρινοί καταχωρητές που χρησιμοποιούνται για ποικίλους σκοπούς
- s1-s11: καταχωρητές, των οποίων οι τιμές αποθηκεύονται ανάμεσα σε κλήσεις συναρτήσεων ή διαδικασιών
- a0-a7: καταχωρητές που χρησιμεύουν στο πέρασμα παραμέτρων και στην επιστροφή τιμών
- ra: εδώ αποθηκεύεται η διεύθυνση που θα επιστρέψει ο έλεγχος του προγράμματος κατά την κλήση μιας συνάρτησης / διαδικασίας, όταν αυτή τερματιστεί
- pc: κρατάει την διεύθυνση της εντολής που εκτελείται σε κάθε στιγμή
- gp: δείκτης που δείχνει στην αρχή των καθολικών μεταβλητών ενός προγράμματος

ΕΝΤΟΛΕΣ ΣΕ ASSEMBLY:

- li : εκχώρηση αριθμητικής σταθεράς
- mv : μεταφορά τιμής από άλλον καταχωρητή
- add : πρόσθεση μεταξύ καταχωρητών
- sub : αφαίρεση μεταξύ καταχωρητών
- mul : πολλαπλασιασμός μεταξύ καταχωρητών
- div : διαίρεση μεταξύ καταχωρητών
- addi : πρόσθεση ακεραίου σε καταχωρητή
- lw : ανάγνωση 4 bytes
- sw : εγγραφή 4 bytes
- b : εκτέλεση άλματος (χωρίς κάποια συνθήκη)
- beq : εκτέλεση άλματος αν οι τιμές των καταχωρητών είναι ίσες
- bne : εκτέλεση άλματος αν η τιμή του πρώτου καταχωρητή είναι διαφορετική από την τιμή του δεύτερου
- blt : εκτέλεση άλματος αν η τιμή του πρώτου καταχωρητή είναι μικρότερη από την τιμή του δεύτερου
- bgt : εκτέλεση άλματος αν η τιμή του πρώτου καταχωρητή είναι μεγαλύτερη από την τιμή του δεύτερου
- ble : εκτέλεση άλματος αν η τιμή του πρώτου καταχωρητή είναι μικρότερη ή ίση της τιμής του δεύτερου
- bge : εκτέλεση άλματος αν η τιμή του πρώτου καταχωρητή είναι μεγαλύτερη ή ίση της τιμής του δεύτερου
- jr : εκτέλεση άλματος μέσω καταχωρητή
- jal : κλήση συνάρτησης ή διαδικασίας
- ecall : υλοποιεί τις κλήσεις του συστήματος

Επιπλέον έχουμε:

- *Είσοδος τιμής* : Η είσοδος δεδομένων από το πληκτρολόγιο γίνεται μέσω των καταχωρητών a0 και a7. Όταν η τιμή του καταχωρητή a7 ισούται με 5 πρόκειται να διαβαστεί ακέραιος. Ο ακέραιος που θα διαβαστεί από το πληκτρολόγιο (κλήση συστήματος ecall) τοποθετείται στον a0.

  li a7, 5

  ecall

  move t1, a0

- *Έξοδος τιμής*: Για εμφάνιση στην οθόνη ακεραίων αριθμών θα ακολουθήσουμε την ίδια λογική. Θα εισάγουμε στον καταχωρητή a7 την τιμή 1 και στον a0 τον ακέραιο που θέλουμε να τυπώσουμε

  li a7, 1

  li a0, t1

- *Τερματισμός Προγράμματος:*

  li a0, 0

  li a7, 93

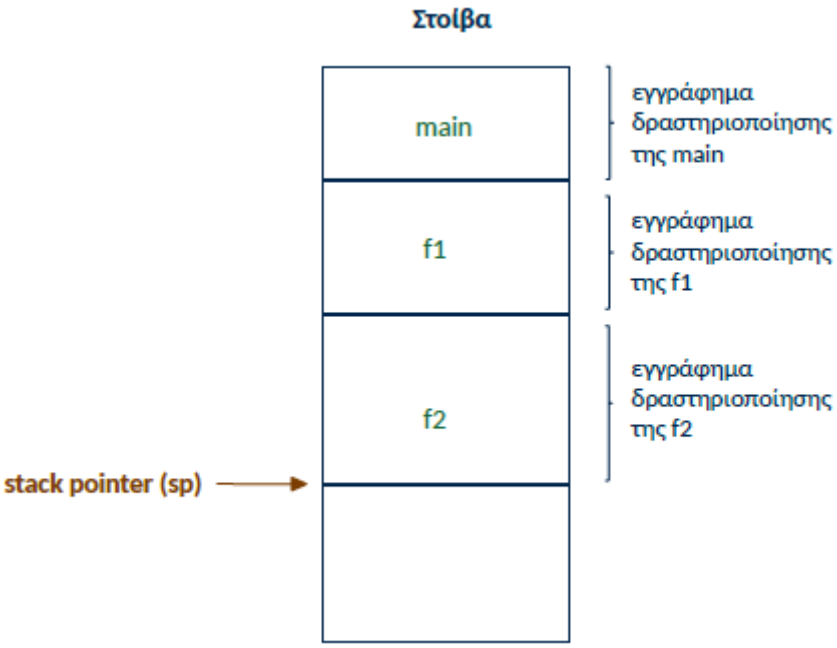
  ecall

ΕΓΓΡΑΦΗΜΑ ΔΡΑΣΤΗΡΙΟΠΟΙΗΣΗΣ:

Αναφερθήκαμε σε αυτό αναλυτικά κατά την παραγωγή του πίνακα συμβόλων, οπότε τώρα θα επεξηγήσουμε τι συμβαίνει με αυτό κατά την παραγωγή τελικού κώδικα. Κατά την εκκίνηση ενός προγράμματος το λειτουργικό σύστημα δεσμεύει χώρο που λειτουργεί σαν στοίβα. Εκεί ακριβώς τοποθετείται ο stack pointer (sp), ο οποίος μετατοπίζεται κατά τον αριθμό των θέσεων που θα περιλαμβάνει το εγγράφημα συνολικά. Ο δείκτης στοίβας δείχνει πάντα στην αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης που εκτελείται κάθε χρονική στιγμή. Κάθε φορά που κάποια συνάρτηση καλεί κάποια άλλη προστίθενται στην στοίβα τα αντίστοιχα εγγραφήματα δραστηριοποίησης. Όταν φτάσουμε σε κάποια συνάρτηση που δεν καλεί καμία επόμενη, τότε ο έλεγχος οφείλει να γυρίσει στην αμέσως προηγούμενη θέση. Ο χώρος που καταλάμβανε η συγκεκριμένη συνάρτηση στην στοίβα πλέον αποδεσμεύεται,

δίχως να σημαίνει ότι τα δεδομένα του διαγράφονται. Ειδικότερα, τα δεδομένα αυτά ονομάζονται «σκουπίδια». Κατά τον τερματισμό εκτέλεσης του κυρίως προγράμματος απελευθερώνεται όλος ο χώρος στην στοίβα και επιστρέφεται στο λειτουργικό σύστημα.

Δίνεται ένα παράδειγμα ακολούθως.



Παραπάνω φαίνεται το εγγράφημα δραστηριοποίησης από εμφωλευμένες κλήσεις συναρτήσεων στη στοίβα. Στην αρχή έχουμε πάντα το κομμάτι της main. Εδώ, η main μας καλεί την συνάρτηση f1, όπου τοποθετείται ακριβώς από κάτω της. Αυτό συμβαίνει με τη βοήθεια του stack pointer όπου μετακινείται τόσες θέσεις, όσες χρειαζόμαστε για το εγγράφημα δραστηριοποίησης της f1. Στην συνέχεια παρατηρούμε ότι υπάρχει εμφωλευμένη κλήση συνάρτησης (της f2) μέσα στην f1. Ομοίως τοποθετείται το εγγράφημά της στην στοίβα μέσω του sp. Το εγγράφημα της main παραμένει καθώς, εφόσον καλούνται σε αυτό συναρτήσεις που δεν έχει ολοκληρωθεί η εκτέλεσή τους ακόμα, δεν μπορεί να αποδεσμεύσει προς το παρόν τον χώρο που καταλαμβάνει. Εφόσον η f2 τερματίσει, θα απελευθερώσει τον χώρο της στη στοίβα και θα επιστρέψει ο δείκτης στοίβας στην f1. Η f1 με την ολοκλήρωση της f2 τερματίζει και αυτή την λειτουργία της, αποδεσμεύοντας και αυτή τον χώρο της. Φτάνουμε έτσι στο σημείο όπου ο stack pointer δείχνει ξανά στην main, η οποία έχει τελειώσει επίσης την λειτουργία της. Έχει έρθει

λοιπόν η σειρά της να αποδεσμεύσει και τον τελευταίο δεσμευμένο χώρο στην στοίβα, ο οποίος με την σειρά του θα επιστραφεί ολόκληρος στο λειτουργικό σύστημα ελεύθερος, με σκοπό να επαναχρησιμοποιηθεί αργότερα.

ΒΟΗΘΗΤΙΚΕΣ ΣΥΝΑΡΤΗΣΕΙΣ:

Και σε αυτή τη φάση της μετάφρασης υλοποιήσαμε μερικές βοηθητικές συναρτήσεις. Σκοπός τους ο ευκολότερος σχεδιασμός, η ευανάγνωστη περιγραφή αλλά και η απλούστευση του κώδικα. Οι βοηθητικές συναρτήσεις που υλοποιήσαμε είναι οι εξής:

- gnlvcode(v): Παράγει τελικό κώδικα για την προσπέλαση μεταβλητών ή διευθύνσεων που είναι αποθηκευμένες σε κάποιο εγγράφημα δραστηριοποίησης κάποιου προγόνου της συνάρτησης ή της διαδικασίας που αυτή τη στιγμή μεταφράζεται. Ως όρισμα δέχεται μια μεταβλητή (v), η οποία είναι και αυτή που θα ψάξουμε για την τιμή της ή για την διεύθυνσή της στον πίνακα συμβόλων. Συγκεκριμένα, από την βοηθητική συνάρτηση του πίνακα συμβόλων (*searchEntity()*) βρίσκουμε το επίπεδο (scope) που βρίσκεται αυτή η εγγραφή μαζί με τις επιπλέον πληροφορίες της (π.χ. offset, mode κλπ.). Έπειτα μπορούμε εύκολα να υπολογίσουμε πόσα επίπεδα παραπάνω βρίσκεται η μη τοπική μεταβλητή στο γενεαλογικό δέντρο του υποπρογράμματος. Έτσι, για αρχή μεταφέρουμε στον καταχωρητή *t0* την διεύθυνση για την στοίβα του γονέα. Αυτή η πληροφορία βρίσκεται αποθηκευμένη στον σύνδεσμο προσπέλασης του υποπρογράμματος ($2^{η}$ θέση στο Εγγράφημα Δραστηριοποίησης) και εκτελούμε την εντολή «*lw t0, -4(sp)*». Στην συνέχεια, για όσες φορές χρειαστεί παράγουμε την εντολή «*lw t0, -4(t0)*» μέχρι να φτάσουμε στην στοίβα του προγόνου που έχει την συγκεκριμένη εγγραφή. Μετά την ολοκλήρωση των μεταβάσεων μέσω των συνδέσμων προσπέλασης ο καταχωρητής t0 δείχνει στην αρχή του Εγγραφήματος Δραστηριοποίησης, στο οποίο ο πίνακας συμβόλων υπέδειξε. Τέλος, θα πρέπει ο καταχωρητής *t0* να κατέβει κατά offset θέσεις ώστε να δείξει στην ακριβή θέση μνήμης που βρίσκεται η πληροφορία που αναζητείται. Το αποτέλεσμα που επιστρέφει η gnlvcode() είναι το περιεχόμενο του *t0*. Θα πρέπει να σημειωθεί ότι αν αναζητείται η τιμή μιας μεταβλητής, τότε θα μεταφερθεί στον καταχωρητή *t0* η διεύθυνση της μεταβλητής που αναζητείται. Αντίθετα, αν αναζητείται η διεύθυνση μιας μεταβλητής, τότε στον καταχωρητή *t0*

θα μεταφερθεί η διεύθυνση, η οποία περιέχει τη διεύθυνση της μεταβλητής που αναζητείται.

- loadvr (v, reg): Παράγει τελικό κώδικα ο οποίος διαβάζει μια μεταβλητή (v) που είναι αποθηκευμένη στη μνήμη και την μεταφέρει σε έναν καταχωρητή (reg). Πρόκειται για μια μεγάλη δομή πολλαπλής επιλογής, η οποία διακρίνει τις εξής περιπτώσεις παράγοντας για κάθε περίπτωση διαφορετικές γραμμές κώδικα assembly:
  - Αν η *v* είναι αριθμητική σταθερά. Στην συγκεκριμένη περίπτωση φορτώνουμε την αριθμητική σταθερά σε έναν καταχωρητή.
    - *li reg, integer*

  Αν η v δεν είναι αριθμητική σταθερά, τότε καλούμε την *searchEntity()* για να βρούμε σε ποιο scope βρίσκεται η μεταβλητή αυτή. Αναλόγως το επίπεδο που βρέθηκε διακρίνουμε 3 περιπτώσεις :
  - Επίπεδο = 0: Σε αυτή την περίπτωση η v είναι καθολική μεταβλητή, αφού βρίσκεται στο Εγγράφημα Δραστηριοποίησης του κυρίως προγράμματος. Επειδή, η πρόσβαση σε καθολικές μεταβλητές είναι συχνή, επιλέγουμε τον καταχωρητή gp (global pointer) ο οποίος θα δείχνει στην αρχή της στοίβας του κυρίως προγράμματος και θα χρησιμοποιηθεί μόνο για αυτόν τον σκοπό. Αυτό γίνεται για το λόγο ότι η διαδικασία φόρτωσης επιπέδων έχει κάποιο κόστος, ιδιαίτερα αν έχουμε να ανέβουμε περισσότερα επίπεδα. Έτσι, η μεταβλητή v προσπελαύνεται μέσω του καταχωρητή gp και βρίσκεται offset θέσεις πάνω από αυτόν.
    - *lw reg, -offset(gp)*
  - Επίπεδο = Ανώτερο Επίπεδο (top scope) : Σε αυτήν την περίπτωση η v έχει δηλωθεί σε υποπρόγραμμα που αυτή την στιγμή εκτελείται. Αν είναι **τοπική μεταβλητή ή προσωρινή μεταβλητή ή τυπική παράμετρος που περνάει με τιμή**, τότε η τιμή της μεταβλητής βρίσκεται offset θέσεις παραπάνω από την αρχή του Εγγραφήματος. Έτσι, παράγεται ο παρακάτω τελικός κώδικας:
    - *lw reg, -offset(sp)* , όπου offset το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.

Αν είναι **τυπική παράμετρος που περνάει με αναφορά**, τότε στο Εγγράφημα Δραστηριοποίησης του υποπρογράμματος έχει τοποθετηθεί η διεύθυνσή της. Άρα για να μεταφερθεί η τιμή μιας τέτοιας μεταβλητής στον καταχωρητή reg, απαιτείται ένα ακόμα βήμα. Πρέπει πρώτα να μεταφερθεί η διεύθυνση της εγγραφής από την στοίβα σε έναν καταχωρητή (π.χ. t0) και μετά να χρησιμοποιηθεί ο καταχωρητής αυτός σαν καταχωρητής δείκτης ώστε να μεταφερθεί στον καταχωρητή reg η τιμή της εγγραφής. Άρα, παράγεται ο παρακάτω κώδικας σε assembly:

- *lw t0, -offset(sp)* , όπου offset το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.
- *lw reg, (t0)*

- Επίπεδο < Ανώτερου Επιπέδου: Σε αυτή την περίπτωση η v έχει δηλωθεί σε κάποιο πρόγονο. Αυτό σημαίνει ότι η πληροφορία που αναζητούμε βρίσκεται σε κάποιο Εγγράφημα Δραστηριοποίησης στο οποίο μπορούμε να έχουμε πρόσβαση μέσα από μια σειρά ανακτήσεων συνδέσμων προσπέλασης. Οπότε στην αρχή καλούμε την *gnlvcode()*, όπου θα τοποθετήσει στον t0 την διεύθυνση στην οποία βρίσκεται αποθηκευμένη η εγγραφή. Αν η εγγραφή αυτή είναι **τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή** τότε διαβάζουμε το περιεχόμενο της θέσης μνήμης. Ο τελικός κώδικας που παράγεται είναι ο εξής:

  - *gnlvcode(v)*
  - *lw reg, (t0)*

  Αν είναι **τυπική παράμετρος που περνάει με αναφορά** και σε αυτή την περίπτωση θα καλέσουμε την *gnlvcode()* όπου θα αποθηκεύσει στο t0 τη διεύθυνση στην οποία είναι αποθηκευμένη η διεύθυνση της μεταβλητής. Στην συνέχεια, ακολουθούμε την ίδια λογική όπως αναφέρθηκε και πιο πάνω παράγοντας τον εξής κώδικα σε assembly:

  - *gnlvcode(v)*
  - *lw t0, (t0)*
  - lw reg, t0

❖ storerv (reg , v): Αντίθετα με την *loadvr()*, η *storerv()* παράγει τελικό κώδικα, ο οποίος αποθηκεύει στη μνήμη την τιμή μιας μεταβλητής (v), η οποία βρίσκεται σε έναν καταχωρητή (reg). Η διαφοροποίησης της *storerv()* με την *loadvr()* βρίσκεται στις τελευταίες εντολές που παράγει η κάθε κλήση της , όπου αντί για την εντολή ανάγνωσης (lw), έχουμε την εντολή αποθήκευσης (sw).

Αρχικά, καλούμε την *searchEntity()* για να βρούμε σε ποιο scope βρίσκεται η μεταβλητή αυτή. Αναλόγως το επίπεδο που βρέθηκε διακρίνουμε 3 περιπτώσεις :

- Επίπεδο = 0: Σε αυτή την περίπτωση η v είναι καθολική μεταβλητή, αφού βρίσκεται στο Εγγράφημα Δραστηριοποίησης του κυρίως προγράμματος. Η τιμή της μεταβλητή v που είναι αποθηκευμένη στον καταχωρητή *reg* αποθηκεύεται στην μνήμη *offset* θέσεις πάνω από τον δείκτη *gp*.
  - *sw reg, -offset(gp)*
- Επίπεδο = Ανώτερο Επίπεδο (top scope) : Σε αυτήν την περίπτωση η v έχει δηλωθεί σε υποπρόγραμμα που αυτή την στιγμή εκτελείται. Αν είναι **τοπική μεταβλητή ή προσωρινή μεταβλητή ή τυπική παράμετρος που περνάει με τιμή**, τότε η τιμή της μεταβλητής θα πρέπει να αποθηκευτεί offset θέσεις παραπάνω από την αρχή του Εγγραφήματος του υποπρογράμματος που αυτή τη στιγμή μεταφράζεται. Έτσι, παράγεται ο παρακάτω τελικός κώδικας:
  - *sw reg, -offset(sp)* , όπου offset το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.

  Αν είναι **τυπική παράμετρος που περνάει με αναφορά**, τότε στο Εγγράφημα Δραστηριοποίησης του υποπρογράμματος θα πρέπει να τοποθετηθεί η διεύθυνσή της. Άρα για να αποθηκευτεί η τιμή του καταχωρητή reg στην μνήμη, απαιτείται ένα ακόμα βήμα. Πρέπει πρώτα να μεταφερθεί η διεύθυνση της εγγραφής από την στοίβα σε έναν καταχωρητή (π.χ. t0) και μετά σε αυτόν τον καταχωρητή είναι που θα αποθηκεύσουμε την τιμή του reg. Άρα, παράγεται ο παρακάτω κώδικας σε assembly:

- *lw t0, -offset(sp)* , όπου offset το offset της μεταβλητής που επιστρέφει ο πίνακας συμβόλων.
- *sw reg, (t0)*

▪ Επίπεδο < Ανώτερου Επιπέδου: Σε αυτή την περίπτωση η v έχει δηλωθεί σε κάποιο πρόγονο. Αυτό σημαίνει ότι η πληροφορία που θέλουμε να αποθηκεύσουμε στη μνήμη βρίσκεται σε κάποιο Εγγράφημα Δραστηριοποίησης, στο οποίο μπορούμε να έχουμε πρόσβαση μέσα από μια σειρά ανακτήσεων συνδέσμων προσπέλασης. Οπότε στην αρχή καλούμε την *gnlvcode()*, όπου θα τοποθετήσει στον t0 την διεύθυνση στην οποία βρίσκεται αποθηκευμένη η εγγραφή. Αν η εγγραφή αυτή είναι **τοπική μεταβλητή ή τυπική παράμετρος που περνάει με τιμή** τότε αποθηκεύουμε την τιμή του reg στη θέσης μνήμης t0. Ο τελικός κώδικας που παράγεται είναι ο εξής:

- *gnlvcode(v)*
- *sw reg, (t0)*

Αν είναι **τυπική παράμετρος που περνάει με αναφορά** και σε αυτή την περίπτωση θα καλέσουμε την *gnlvcode()* όπου θα αποθηκεύσει στο t0 τη διεύθυνση στην οποία είναι αποθηκευμένη η διεύθυνση της μεταβλητής. Στην συνέχεια ακολουθούμε την ίδια λογική όπως αναφέρθηκε και πιο πάνω παράγοντας τον εξής κώδικα σε assembly:

- *gnlvcode(v)*
- *lw t0, (t0)*
- sw reg, t0

❖ *produce*(): Σε αυτή την συνάρτηση για κάθε τετράδα ενδιάμεσου κώδικα δημιουργούμε μια νέα εντολή τελικού κώδικα. Ο κώδικας που παράγουν οι εντολές αυτές αποθηκεύεται κατευθείαν σε ένα αρχείο assembly (test.asm). Παρακάτω, παραθέτουμε τις διαφορετικές περιπτώσεις τετράδων που μπορούμε να συναντήσουμε και τον κώδικα που γράφεται στο αρχείο για κάθε μία από αυτές τις περιπτώσεις:

- Άλμα: δημιουργείται η εντολή διακλάδωσης **b** που μας μεταφέρει στην Lx ετικέτα.
  - *b Lx*

- Λογικοί Τελεστές:

relop, x, y, z

όπου relop ένας από τους τελεστές

=, >, <, <>, >=, <=

Οι τιμές των x, y φορτώνονται αντίστοιχα στους καταχωρητές t1 και t2. Το z αντιστοιχεί στο label που θα γίνει το άλμα αν ισχύει η σύγκριση. Οπότε έχουμε:

  - για «=» δημιουργείται η εντολή **beq**
  - *beq t1, t2, label*
  - για «<>» δημιουργείται η εντολή **bne**
  - *bne t1, t2 , label*
  - για «>» δημιουργείται η εντολή **bgt**
  - *bgt t1, t2 , label*
  - για «<» δημιουργείται η εντολή **blt**
  - *blt t1, t2 , label*
  - για «>=» δημιουργείται η εντολή **bge**
  - *bge t1, t2 , label*
  - για «<=» δημιουργείται η εντολή **ble**
  - *ble t1, t2 , label*

- Εκχώρηση:

:=, x, _, z

Φορτώνουμε στον καταχωρητή t1 την τιμή του x την οποία στην συνέχεια την αποθηκεύουμε στο z.

- Αριθμητικοί Τελεστές:

```
op, x, y, z     # z = x op y
# op: +, -, * , /
# x, y, z: variables
```

Οι τιμές των x, y φορτώνονται αντίστοιχα στους καταχωρητές t1 και t2. Το z αντιστοιχεί στο αποτέλεσμα της πράξης και αποθηκεύεται στον t1 και μετά στην μνήμη Οπότε έχουμε:

- για «+» δημιουργείται η εντολή **add**
  - *add t1, t1, t2*
- για «-» δημιουργείται η εντολή **sub**
  - sub t1, t1, t2
- για «/» δημιουργείται η εντολή **div**
  - *div t1, t1, t2*
- για «*» δημιουργείται η εντολή **mul**
  - *mul t1, t1, t2*

- <u>Εντολή return:</u>

```
ret, x, _, _
```

Φορτώνεται η τιμή της μεταβλητής x από την μνήμη και αποθηκεύεται στον καταχωρητή t1. Έπειτα φορτώνουμε την 3η θέση του Εγγραφήματος Δραστηριοποίησης (στον t0) και αποθηκεύουμε το αποτέλεσμα της συνάρτησης εκεί.

- *lw t0,-8(sp)*
- *sw t1, (t0)*

- <u>Είσοδος</u>: δημιουργεί τις εντολές:

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

- *li a7, 5*
- *ecall*
- *mv t1, a0*

Η τιμή που πήρε από το πληκτρολόγιο αποθηκεύτηκε στην μεταβλητή x και μετά στον t1. Αποθηκεύουμε την τιμή του t1 στην στοίβα.

- Έξοδος:

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

Φορτώνουμε την τιμή του print στον t1 για να την περάσω στον a0. Δημιουργεί τις εντολές:

- *li a0, t1*
- *li a7, 1*
- *ecall*

- Πέρασμα Παραμέτρου: χρειαζόμαστε αρχικά να βρούμε το frame length του υποπρογράμματος, για αυτό ψάχνουμε αρχικά το συγκεκριμένο entity που το περιέχει. Δημιουργείται η εντολή, όπου τοποθετούμε τον fp να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί:
  - *addi fp, sp, framelength*

Γνωρίζουμε ότι το πέρασμα παραμέτρων γίνεται με τρεις διαφορετικούς τρόπους που τους μελετάμε ξεχωριστά.

Έτσι, για πέρασμα με τιμή φορτώνουμε την τιμή της παραμέτρου στον t0 και δημιουργείται η εντολή :

- *sw t0, -x(fp) ,* x ο αύξων αριθμός της παραμέτρου.

Για πέρασμα με αναφορά διακρίνουμε δύο περιπτώσεις. Βρίσκουμε αρχικά το entity της μεταβλητής x. Αν η x είναι στο ίδιο βάθος φωλιάσματος με την καλούσα συνάρτηση, τότε ελέγχουμε σε δεύτερο επίπεδο αν η x είναι τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή. Σε αυτήν την περίπτωση δημιουργούνται οι εντολές:

- *addi t0, sp, -y,* y να είναι το offset της μεταβλητής x
- *sw t0, -z(fp)*, z να είναι ο αύξων αριθμός της παραμέτρου.

Αν ωστόσο η x είναι παράμετρος που έχει περαστεί με αναφορά, τότε δημιουργούνται οι εντολές:

- *lw t0, -y(sp),* y να είναι το offset της μεταβλητής x
- *sw t0, -z(fp),* z να είναι ο αύξων αριθμός της παραμέτρου.

Στην συνέχεια ελέγχουμε την περίπτωση όπου το βάθος φωλιάσματος της καλούσας συνάρτησης είναι διαφορετικό από αυτό της μεταβλητής x. Διακρίνουμε ξανά αν η x είναι τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή και τότε έχουμε:

- *gnlvcode(x)*
- *sw t0, -z (fp*), z να είναι ο αύξων αριθμός της παραμέτρου.

Ενώ εάν η x είναι παράμετρος που έχει περαστεί με αναφορά, τότε δημιουργούνται οι εντολές:

- *gnlvcode(x)*
- *lw t0, t0*
- *sw t0, -z(fp)***,** z να είναι ο αύξων αριθμός της παραμέτρου.

Τέλος, έχουμε το πέρασμα με <u>επιστροφή τιμής</u>, όπου γεμίζουμε το 3^{ο} πεδίο του Εγγραφήματος Δραστηριοποίησης της κληθείσας συνάρτησης με την διεύθυνση της προσωρινής μεταβλητής στην οποία θα επιστραφεί η τιμή. Έτσι, δημιουργούνται οι εντολές:

- *addi t0, sp, -y*, y το offset της μεταβλητής που επιστρέφει η συνάρτηση.
- *sw t0, -8(fp)*

- <u>Κλήση Συνάρτησης</u>: Αρχικά γεμίζουμε το 2^{ο} πεδίο του Εγγραφήματος Δραστηριοποίησης της κληθείσας συνάρτησης, τον σύνδεσμο προσπέλασης, με την διεύθυνση του Εγγραφήματος του γονέα της. Αυτό γίνεται έτσι ώστε, η κληθείσα να γνωρίζει που να κοιτάξει αν χρειαστεί να προσπελάσει μια μεταβλητή την οποία έχει δικαίωμα να προσπελάσει , αλλά δεν της ανήκει. Αν η καλούσα και η κληθείσα βρίσκονται στο ίδιο βάθος φωλιάσματος τότε δημιουργούνται οι εντολές:

  - *lw t0, -4(sp)*
  - *sw t0, -4(fp)*

  ενώ αν είναι σε διαφορετικό βάθος φωλιάσματος τότε δημιουργείται η εντολή:

  - *sw sp, -4(fp)*

Στην συνέχεια μεταφέρουμε τον δείκτη στοίβας στην κληθείσα, καλούμε την συνάρτηση και όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα.

- *addi sp , sp, framelength*
- *jal f*
- *addi sp , sp, -framelength*

- Έναρξη Συνάρτησης / Διαδικασίας: Στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του Εγγραφήματος την διεύθυνση επιστροφής της, την οποία έχει τοποθετήσει στον ra η jal :
  - *sw ra, (sp)*
- Κλείσιμο Συνάρτησης / Διαδικασίας: Στο τέλος κάθε συνάρτησης, παίρνουμε την διεύθυνση επιστροφής της συνάρτησης και την βάζουμε πάλι στο ra . Τέλος, μέσω του ra επιστρέφουμε στην καλούσα:
  - *lw ra, (sp)*
  - *b ra*
- main: δημιουργεί την εντολή η οποία κάνει jump στην ετικέτα main
  - *b main* , στην αρχή του αρχείου test.asm .

  Αυτό γίνεται γιατί το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται. Στην συνέχεια, στο τέλος του αρχείου κατεβάζουμε τον sp κατά framelength θέσεις της main.Τέλος, σημειώνουμε στον gp το Εγγράφημα της main ώστε να έχουμε εύκολη πρόσβαση στις global μεταβλητές:
  - *main: addi sp, sp, framelength*
  - *mv gp, sp*
- κλείσιμο προγράμματος: δημιουργεί τις εντολές εφόσον αναγνωρίσει την τετράδα που ξεκινάει με “halt”.
  - *li a0, 0*
  - *li a7, 93*

- *ecall*

ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ: Αποτέλεσμα αρχείου test.asm με αρχείο εισόδου *_HappyDay.ci. :*

```
1     .text
2  L0:
3     b main
4
5  L1:
6     sw ra,(sp)
7
8  L2:
9     lw t1, -12(sp)
10    li t2, 0
11    bgt, t1, t2, L4
12
13 L3:
14    b L9
15
16 L4:
17    lw t1, -16(sp)
18    lw t2, -20(sp)
19    bgt, t1, t2, L6
20
21 L5:
22    b L9
23
24 L6:
25    lw t1, -24(sp)
26    li t2, 1
27    add, t1, t1, t2
28    sw t1, -28(sp)
29
30 L7:
31    lw t1, -28(sp)
32    sw t1, -24(sp)
33
34 L8:
35    b L2
36    lw t1, -20(sp)
37    li t2, 1
38    blt, t1, t2, L11
39
40 L10:
41    b L14
42
43 L11:
44    lw t1, -20(sp)
45    li t2, 1
46    add, t1, t1, t2
47    sw t1, -32(sp)
48
49 L12:
50    lw t1, -32(sp)
51    sw t1, -20(sp)
52
53 L13:
54    b L9
55
56 L14:
57    lw t1, -20(sp)
58    li t2, 1
59    sub, t1, t1, t2
60    sw t1, -36(sp)
61
62 L15:
63    lw t1, -36(sp)
64    sw t1, -20(sp)
65
66 L16:
67    lw t1, -16(sp)
68    lw t2, -20(sp)
69    bgt, t1, t2, L18
70
71 L17:
72    b L20
73
74 L18:
75    lw t1, -16(sp)
76    li a0, t1
77    li a7, 1
78    ecall
79
```

```
80   L19:
81       b L22
82
83   L20:
84       lw t1, -16(sp)
85       lw t2, -20(sp)
86       add, t1, t1, t2
87       sw t1, -40(sp)
88
89   L21:
90       lw t1, -40(sp)
91       li a0, t1
92       li a7, 1
93       ecall
94
95   L22:
96       li t1, 0
97       sw t1, -44(sp)
98
99   L23:
100      lw t1, -20(sp)
101      lw t2, -16(sp)
102      bge, t1, t2, L25
103
104  L24:
105      b L28
106
107  L25:
108      lw t1, -20(sp)
109      lw t2, -16(sp)
110      add, t1, t1, t2
111      sw t1, -48(sp)
112
113  L26:
114      lw t1, -48(sp)
115      sw t1, -20(sp)
116
117  L27:
118      li t1, 1
119      sw t1, -44(sp)
120      lw t1, -12(sp)
121      lw t2, -16(sp)
122      blt, t1, t2, L30
123
124  L29:
125      b L33
126
127  L30:
128      lw t1, -12(sp)
129      lw t2, -16(sp)
130      sub, t1, t1, t2
131      sw t1, -52(sp)
```

```
133  L31:
134      lw t1, -52(sp)
135      sw t1, -12(sp)
136
137  L32:
138      li t1, 1
139      sw t1, -44(sp)
140
141  L33:
142      lw t1, -44(sp)
143      li t2, 1
144      beq, t1, t2, L22
145
146  L34:
147      lw t1, -12(sp)
148      lw t2, -16(sp)
149      add, t1, t1, t2
150      sw t1, -56(sp)
151
152  L35:
153      lw t1, -56(sp)
154      lw t2, -20(sp)
155      add, t1, t1, t2
156      sw t1, -60(sp)
157
158  L36:
159      lw t1, -24(sp)
160      lw t2, -12(sp)
161      div, t1, t1, t2
162      sw t1, -64(sp)
163
164  L37:
165      lw t1, -64(sp)
166      lw t2, -16(sp)
167      mul, t1, t1, t2
168      sw t1, -68(sp)
169
170  L38:
171      lw t1, -68(sp)
172      lw t2, -20(sp)
173      mul, t1, t1, t2
174      sw t1, -72(sp)
175
176  L39:
177      lw t1, -72(sp)
178      lw t2, -24(sp)
179      mul, t1, t1, t2
180      sw t1, -76(sp)
181
182  L40:
183      lw t1, -60(sp)
184      lw t2, -76(sp)
```

```
184         lw t2, -76(sp)
185         add, t1, t1, t2
186         sw t1, -80(sp)
187
188     L41:
189         lw t1, -80(sp)
190         lw t0,-8(sp)
191         sw t1, (t0)
192
193     L42:
194         lw ra,(sp)
195         jr ra
196
197     L43:
198         sw ra,(sp)
199
200     L44:
201         lw t1, -24(gp)
202         lw t2, -12(gp)
203         add, t1, t1, t2
204         sw t1, -12(sp)
205
206     L45:
207         lw t1, -12(sp)
208         lw t0,-8(sp)
209         sw t1, (t0)
210
211     L46:
212         lw ra,(sp)
213         jr ra
214
215     L47:
216         sw ra,(sp)
217
218     L48:
219         lw t1, -24(gp)
220         li t2, 1
221         add, t1, t1, t2
222         sw t1, -12(sp)
223
224     L49:
225         lw t1, -12(sp)
226         sw t1, -24(gp)
227
228     L50:
229         lw ra,(sp)
230         jr ra
231
232     main:
233         addi sp, sp, 32
234         mv gp, sp
235
```

```
236     L52:
237         li a7, 5
238         ecall
239         mv t1, a0
240         sw t1, -12(gp)
241
242     L53:
243         li a7, 5
244         ecall
245         mv t1, a0
246         sw t1, -16(gp)
247
248     L54:
249         li a7, 5
250         ecall
251         mv t1, a0
252         sw t1, -20(gp)
253
254     L55:
255         li a7, 5
256         ecall
257         mv t1, a0
258         sw t1, -24(gp)
259
260     L56:
261         lw t0,-4(sp)
262         sw t0, -4(fp)
263         addi sp, sp, 16
264         jal  L48
265         addi sp, sp, -16
266
267     L57:
268         addi fp, sp, 84
269         lw t0, -12(gp)
270         sw t0, -12(fp)
271
272     L58:
273         addi fp, sp, 84
274         lw t0, -16(gp)
275         sw t0, -16(fp)
276
277     L59:
278         addi fp, sp, 84
279         lw t0, -20(gp)
280         sw t0, -20(fp)
281
282     L60:
283         addi fp, sp, 84
284         lw t0, -24(gp)
285         sw t0, -24(fp)
286
```

```
287   L61:
288       addi fp, sp, 84
289       addi t0, sp, -28
290       sw t0, -8(fp)
291
292   L62:
293       lw t0,-4(sp)
294       sw t0, -4(fp)
295       addi sp, sp, 84
296       jal  L2
297       addi sp, sp, -84
298
299   L63:
300       lw t1, -28(gp)
301       li a0, t1
302       li a7, 1
303       ecall
304
305   L64:
306       li a0, 0
307       li a7, 93
308       ecall
309
```