

## Αναφορά: Εργασία στους μεταφραστές

### Ο Μεταγλωττιστής της γλώσσας C-imple

- Λεκτική & συντακτική ανάλυση:

Η λειτουργία που επιτελεί ο λεκτικός αναλυτής είναι η εξής, δοθέντος του αρχείου που περιέχει τον πηγαίο κώδικα στη γλώσσα C-imple, κάθε φορά που καλείται επιστρέφει την επόμενη λεκτική μονάδα (πχ δεσμευμένη λέξη, όνομα μεταβλητής, χαρακτήρες διαχωρισμού, άνοιγμα-κλείσιμο σχολίων, λογικούς τελεστές, αριθμητικούς τελεστές, τελεστές ανάθεσης καθώς και σύμβολα ομαδοποίησης). Στην λεκτική ανάλυση μπορεί να γίνει και ο έλεγχος κάποιων συντακτικών σφαλμάτων όπως,  $\text{number} > 2^{32}-1$  ή  $\text{number} < -2^{32}-1$ , αλφαριθμητικό με συνολικό άθροισμα χαρακτήρων  $> 30$  και άνοιγμα σχολίων χωρίς αυτά να έχουν «κλείσει».

Βασική συνάρτηση του λεκτικού αναλυτή είναι η **lex()**, η οποία κάθε φορά που εκτελείται επιστρέφει ένα αντικείμενο κλάσης **Token**, που περιέχει όλες τις απαραίτητες πληροφορίες που χρειάζεται ο συντακτικός αναλυτής (τύπος του token, αλφαριθμητικό που διαβάστηκε και αριθμός γραμμής στην οποία ανήκει). Όσον αφορά την υλοποίηση του λεκτικού αναλυτή, έχει χρησιμοποιηθεί η τεχνική του πεπερασμένου αυτομάτου, όπου αρχίζει από μια κατάσταση 'start', και όταν η κατάσταση γίνει η 'ok', επιστρέφει το αντικείμενο τύπου Token, ενώ όταν η κατάσταση γίνει 'error' επιστρέφεται μήνυμα λάθους καθώς υπάρχει κάποιο σφάλμα που μπόρεσε να αναγνωρίσει ο λεκτικός αναλυτής.

Στη συνέχεια, από την γραμματική της γλώσσας C-imple, γίνεται η κατασκευή του συντακτικού αναλυτή, όπου γίνεται ο έλεγχος εάν το πηγαίο πρόγραμμα ανήκει ή όχι στη γλώσσα. Ο τρόπος κατασκευής του συντακτικού αναλυτή είναι με αναδρομική κατάβαση. Η κατασκευή του βασίζεται σε LL(1) γραμματική (left to right, leftmost derivation, one look-ahead symbol), δηλαδή, αναγνωρίζει από αριστερά προς τα δεξιά. Την αριστερότερη δυνατή παραγωγή και όταν βρίσκεται σε δίλημμα για το ποιόν κανόνα να ακολουθήσει, της αρκεί να κοιτάξει το αμέσως επόμενο σύμβολο.

Πιο συγκεκριμένα, για κάθε έναν από τους κανόνες της γραμματικής, φτιάχνουμε ένα υποπρόγραμμα. Όταν συναντάμε μη τερματικό σύμβολο καλούμε και το αντίστοιχο υποπρόγραμμα. Όταν συναντάμε τερματικό σύμβολο, εάν και ο λεκτικός αναλυτής επιστρέφει λεκτική μονάδα που αντιστοιχεί στο τερματικό σύμβολο έχουμε αναγνωρίσει επιτυχώς τη λεκτική μονάδα, αλλιώς εάν ο λεκτικός αναλυτής δεν επιστρέψει τη λεκτική μονάδα που περιμένει ο συντακτικός αναλυτής, τότε επιστρέφουμε το αντίστοιχο μήνυμα λάθους. Όταν αναγνωριστεί και η τελευταία

λέξη του πηγαίου προγράμματος, τότε η συντακτική ανάλυση έχει στεφθεί με επιτυχία.

- **Παραγωγή Ενδιάμεσου Κώδικα:**

Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες που αποτελείται από μία θέση για τον τελεστή και τρεις για τα τελούμενα (π.χ. +, a, b, c). Η κάθε τετράδα χαρακτηρίζεται και από μία μοναδική ετικέτα. Μόλις τελειώσει η εκτέλεση μιας τετράδας, εκτελείται η τετράδα με τον αμέσως μεγαλύτερο αριθμό ετικέτας, εκτός και αν πρόκειται για εντολή άλματος (jump, \_, \_, z).

Οι τελεστές διακρίνονται στις παρακάτω κατηγορίες:

- Τελεστές αριθμητικών πράξεων: (op, x, y, z)  
Εφαρμόζεται ο τελεστής op στα τελούμενα x και y και το αποτέλεσμα τοποθετείται στο τελούμενο z.
- Τελεστής εκχώρησης: (:=, x, \_, z)  
Αντιστοιχεί στην εκχώρηση  $z := x$ .
- Τελεστής άλματος χωρίς συνθήκη: (jump, \_, \_, z)  
Μεταπήδηση χωρίς όρους στην τετράδα με ετικέτα z.
- Τελεστής άλματος υπό συνθήκη: (relop, x, y, z)  
Μεταπήδηση στην τετράδα με ετικέτα z, εάν ισχύει  $x \text{ relop } y$ , όπου relop είναι ένας από τους λογικούς τελεστές (=, <, >, <=, >=, <>).
- Αρχή και τέλος ενότητας: (begin\_block, name, \_, \_ && end\_block, name, \_, \_)  
Υπονοεί την αρχή και το τέλος ενός υποπρογράμματος με το όνομα name. Ο διαχωρισμός μεταξύ του block ενός υποπρογράμματος και του block του κυρίως προγράμματος είναι ότι το τέλος του κυρίως σηματοδοτείται και με την επιπλέον τετράδα (halt, \_, \_, \_) πριν το end\_block.
- Τελεστές συναρτήσεων/διαδικασιών:  
(par, x, m, \_): όπου x είναι παράμετρος συνάρτησης και m είναι ο τρόπος μετάδοσης (CV: με τιμή, REF: με αναφορά και RET: επιστροφή τιμής συνάρτησης)  
(call, name, \_, \_): κλήση συνάρτησης/διαδικασίας με το όνομα name.
- Τελεστές εισόδου/εξόδου:  
(inp, x, \_, \_): διαβάζεται η είσοδος από το χρήστη και τοποθετείται στην μεταβλητή x.  
(out, x, \_, \_): εμφανίζεται η τιμή της μεταβλητής x στην οθόνη.

Επιπλέον, για να γίνει πιο κατανοητός ο κώδικας, έχουν χρησιμοποιηθεί και οι παρακάτω βοηθητικές υπορουτίνες:

**nextquad():** επιστρέφει τον αριθμό ετικέτας της επόμενης τετράδας που πρόκειται να παραχθεί.

**genquad(op, x, y, z):** δημιουργεί την επόμενη τετράδα (op, x, y, z).

**newtemp():** δημιουργεί και επιστρέφει μια νέα προσωρινή μεταβλητή της μορφής T\_1, T\_2...

**emptylist():** δημιουργεί μια κενή λίστα ετικετών τετράδων.

**makelist(x):** δημιουργεί μια λίστα ετικετών τετράδων που περιέχει μόνο τον αριθμό ετικέτας x.

**mergelist(list1, list2):** δημιουργεί μια λίστα ετικετών τετράδων από την συνένωση των list1 και list2.

**backpatch(list, z):** η list αποτελείται από ετικέτες τετράδων στις οποίες το τελευταίο τελούμενο είναι ασυμπλήρωτο. Η backpatch επισκέπτεται και σε συμπληρώνει αυτές τις τετράδες με την ετικέτα z.

Με τη βοήθεια των παραπάνω υπορουτινών, καλώντας αυτές τις μεθόδους, στα κατάλληλα σημεία, μέσα στα υποπρογράμματα που έχουν δημιουργηθεί για την συντακτική ανάλυση, δημιουργούνται οι τετράδες του ενδιάμεσου κώδικα.

#### Ερμηνεία τρόπου παραγωγής ενδιάμεσου κώδικα

Θεωρώντας τους όρους της γραμματικής της γλώσσας ως τα υποπρογράμματα που δημιουργήσαμε στη συντακτική ανάλυση, προκύπτουν τα παρακάτω:

- Αριθμητικές εκφράσεις:

expression : optionalSign {p1} term1 ( ADD\_OP term2 {p2}) \* {p3}

{p1}: if optionalSign != null:

w = newtemp()

genquad(optionalSign, T1.place, \_, w)

{p2}: w = newtemp()

genquad(ADD\_OP, T1.place, T2.place, w)

T1.place = w

{p3}: E.place = T1.place

Στο {p1} ελέγχουμε αν υπάρχει κάποιο προαιρετικό πρόσημο στην έκφρασή μας και εάν υπάρχει το αποθηκεύουμε σε μία προσωρινή μεταβλητή. Η τιμή της T1.place είναι αυτή που επιστρέφεται από τη μέθοδο term1. Στο {p2} καθόσον 'βλέπουμε' αθροιστικό τελεστή, δημιουργούμε μια προσωρινή μεταβλητή που θα κρατήσει το μέχρι στιγμής αποτέλεσμα. Έπειτα, παράγουμε την τετράδα που προσθέτει στην προσωρινή μεταβλητή, την τιμή που έχει επιστρέψει το term2 (T2.place). Το μέχρι στιγμής αποτέλεσμα τοποθετείται στο T1.place, έτσι ώστε να χρησιμοποιηθεί σε περίπτωση που υπάρχει και κάποιο ακόμα term2. Στο {p3}, το T1.place γνωρίζουμε ότι έχει τη μεταβλητή που περιέχει το τελικό αποτέλεσμα και την επιστρέφουμε ως την τιμή αποτελέσματος του expression (E.place).

Η λογική και για τους υπόλοιπους τελεστές αριθμητικών πράξεων είναι ανάλογη με την παραπάνω.

- Λογικές Παραστάσεις:

Ο λογικός τελεστής OR:

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

$\{P_1\}$ :  $B.true = Q^1.true$

$B.false = Q^1.false$

$\{P_2\}$ :  $backpatch(B.false, nextquad())$

$\{P_3\}$ :  $B.true = merge(B.true, Q^2.true)$

$B.false = Q^2.false$

Στο  $\{p1\}$  μεταφέρουμε τις τιμές που έχει επιστρέψει η κλήση της  $Q1$  στις λίστες  $B.true$  και  $B.false$ . Στο  $\{p2\}$  πλέον γνωρίζουμε ότι υπάρχει κάποιο  $Q2$  ακόμα και συνεπώς η εσφαλμένη αποτίμηση του  $Q1$  ( $B.false$ ) θα πρέπει να μας οδηγήει πάνω σε αυτό. Στο  $\{p3\}$ , στη λίστα  $B.true$  προσθέτουμε και τις αληθείς τιμές του  $Q2$ , όμως εδώ δεν γνωρίζουμε που θα μας οδηγήσει η αληθής αποτίμηση των παραστάσεων και συνεπώς οι τετράδες της  $B.true$  παραμένουν ασυμπλήρωτες. Τέλος, θα πρέπει να αναθέσουμε στο  $B.false$  το  $Q2.false$  γιατί στην περίπτωση που δεν υπάρχει κάποιο άλλο  $Q2$  (και συνεπώς ο βρόχος δεν θα συνεχίσει να εκτελείται), θα πρέπει, το  $B.false$ , να μας οδηγήει στο κατάλληλο σημείο του κώδικα.

Με παρόμοια λογική σκεφτόμαστε και για τον τελεστή AND, αφού λάβουμε υπόψιν την ιδιαιτερότητά του σε σχέση με το OR. Στο AND, οι αληθείς συνθήκες είναι πλέον αυτές που θα μας οδηγήσουν στην επόμενη συνθήκη (στο  $Q2$  του προηγούμενου παραδείγματος).

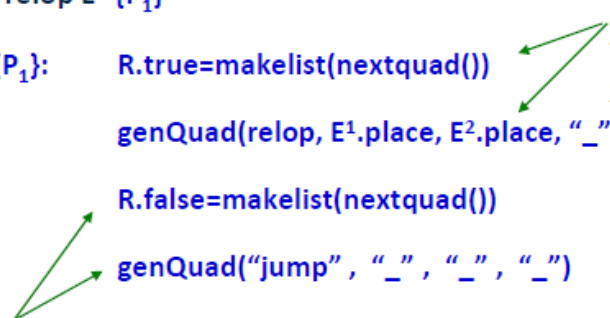
$R \rightarrow E^1 \text{ relop } E^2 \{P_1\}$

$\{P_1\}$ :  $R.true = makelist(nextquad())$

$genQuad(relop, E^1.place, E^2.place, "_")$

$R.false = makelist(nextquad())$

$genQuad("jump", "_", "_", "_")$



Σημειώνουμε την ασυμπλήρωτη τετράδα ( $relop, E1.place, E2.place, \_$ ) που δημιουργείται έτσι ώστε, όταν θα γνωρίζουμε το σημείο που θα μας οδηγήει η αληθής αποτίμησης της, να την συμπληρώσουμε κατάλληλα μέσω της λίστας  $R.true$ . Έτσι δημιουργούμε και μία ασυμπλήρωτη τετράδα άλματος που την κρατάμε στην  $R.false$ .

- Κλήση υπορουτινών:  
Στην κλήση υπορουτινών (π.χ. function (in a, in b)), πρώτα δημιουργούμε τις παραμέτρους της κληθείσας συνάρτησης και, εφόσον η συνάρτηση επιστρέφει κάποια τιμή και χρειαζόμαστε να κρατήσουμε κάπου την τιμή επιστροφής της, δημιουργούμε και μία προσωρινή μεταβλητή που θα είναι παράμετρος τύπου RET. Τέλος, ακολουθεί ο τελεστής κλήσης της συνάρτησης (call, function, \_, \_).
- Μελέτη περιπτώσεων statements (while, if, ...):  
Η δομή while:

$S \rightarrow \text{while } \{P1\} B \text{ do } \{P2\} S^1 \{P3\}$

```
{P1}:    Bquad:=nextquad()
{P2}:    backpatch(B.true,nextquad())
{P3}:    genquad("jump","_","_",Bquad)
         backpatch(B.false,nextquad())
```

Αρχικά, στο {p1} σημειώνουμε την συνθήκη που ελέγχεται κάθε φορά στο while για να γνωρίζουμε το σημείο επιστροφής μετά την εκτέλεση των εντολών του while. Στο {p2} πλέον γνωρίζουμε ότι η λίστα B.true που περιέχει ασυμπλήρωτες τετράδες (που δεν γνωρίζαμε που πρέπει να μας οδηγήσουν προηγουμένως, στις λογικές παραστάσεις), θα πρέπει να δείχνει στις συνθήκες προς εκτέλεση που περιέχονται στο βρόχο while. Αντίστοιχα, στο {p3}, προσθέτουμε εντολή άλματος που μας οδηγεί ξανά στην αρχή του while, καθώς επίσης, γνωρίζουμε πλέον ότι η ψευδής αποτίμηση της συνθήκης B (B.false), θα μας οδηγήσει εκτός βρόχου, και συνεπώς στην αμέσως επόμενη εντολή που πρόκειται να παραχθεί.

Η δομή if ... else:

$S \rightarrow \text{if } B \text{ then } \{P1\} S^1 \{P2\} \text{ TAIL } \{P3\}$

```
{P1}:    backpatch(B.true,nextquad())
{P2}:    ifList=makelist(nextquad())
         genquad("jump","_","_",Bquad)
         backpatch(B.false,nextquad())
{P3}:    backpatch(ifList,nextquad())
```

$\text{TAIL} \rightarrow \text{else } S^2 \mid \text{TAIL} \rightarrow \epsilon$

Με την ίδια λογική, αφού πλέον γνωρίζουμε ότι η αληθής αποτίμηση της συνθήκης B θα μας οδηγήσει στην εκτέλεση των εντολών εντός του if, στο {p1} συμπληρώνουμε τις ασυμπλήρωτες τετράδες του B.true να δείχνουν σε αυτό το σημείο. Αμέσως μετά, στο {p2} πρέπει αρχικά να θέσουμε την ψευδής αποτίμηση της συνθήκης B, να δείχνει στο else, εάν αυτό υπάρχει

βέβαια. Επίσης, δημιουργούμε μια ασυμπλήρωτη τετράδα άλματος της οποίας την ετικέτα και κρατάμε στο ifList, και στο {p3}, όταν δηλαδή γνωρίζουμε το τέλος του if ... else statement, την βάζουμε να δείχνει σε εκείνο το σημείο. Με αυτόν τον τρόπο επιτυγχάνουμε να αποφευχθεί η εκτέλεση των εντολών του else, εφόσον έχουν εκτελεστεί αυτές του if.

Η δομή switch:

S -> switch {P1}

( (cond): {P2} S<sup>1</sup> break {P3} )\*

default: S<sup>2</sup> {P4}

```
{P1}: exitlist = emptylist()
{P2}: backpatch(cond.true,nextquad())
{P3}: e = makelist(nextquad())
      genquad('jump', '_', '_', '_')
      mergelist(exitlist,e)
      backpatch(cond.false,nextquad())
{P4}: backpatch(exitlist,nextquad())
```

Η δομή switch, ελέγχει τις συνθήκες cond, μόλις μία από αυτές βρεθεί αληθής, τότε εκτελούνται οι αντίστοιχες S<sup>1</sup> statements. Μετά ο έλεγχος μεταβαίνει έξω από την switch. Αν, κατά το πέρασμα καμία από τις cond δεν ισχύσει, τότε ο έλεγχος μεταβαίνει στην default όπου και εκτελούνται οι αντίστοιχες S<sup>2</sup>. Στη συνέχεια ο έλεγχος μεταβαίνει έξω από την switch.

Στο {p1}, κρατάμε όλες τις λίστες που θα μας οδηγήσουν εκτός βρόχου. Στο {p2}, η αληθής αποτίμηση της cond, θα μας οδηγήσει στις εντολές εντός του cond και συνεπώς οι τετράδες του cond.true συμπληρώνονται κατάλληλα για να δείχνουν σε αυτό το σημείο. Στη συνέχεια, στο {p3} προσθέτουμε στην exitlist μία ακόμα ετικέτα τετράδας, που σε περίπτωση που οι εντολές cond έχουν αποτιμηθεί ως αληθείς θα πρέπει να μεταβούμε έξω από το βρόχο μέσω του jump στο κατάλληλο σημείο. Επίσης, σε περίπτωσης εσφαλμένης αποτίμησης των συνθηκών γνωρίζουμε ότι είτε θα μεταβούμε στην επόμενη συνθήκη, αν υπάρχει, είτε θα μεταβούμε στο default statement. Επομένως, αυτό είναι το κατάλληλο σημείο για να συμπληρώσουμε τις τετράδες του cond.false. Τέλος, στο {p4}, μετά το default, θα πρέπει να δείχνουν οι εντολές του exitlist, έτσι ώστε να μας οδηγήσουν εκτός δομής.

Η δομή incase:

```
S ->   incase {P1}
        ( when (condition) do {P2}
          sequence {P3}
          end do ) *
        endincase {P4}
{P1}:   w=newTemp()
        p1Quad=nextquad()
        genquad(":=","1","_",w)
{P2}:   backpatch(cond.true,nextquad())
        genquad(":=","0","_",w)
{P3}:   backpatch(cond.false,nextquad())
{P4}:   genquad("=", w,O,p1quad)
```

Η δομή incase, ελέγχει τις συνθήκες condition. Για κάθε μία για τις οποίες η αντίστοιχη συνθήκη condition ισχύει εκτελούνται οι εντολές sequence που την ακολουθούν. Ο έλεγχος μεταβαίνει στην αρχή του incase εάν έστω και ένα sequence να έχει εκτελεστεί, αλλιώς μεταβαίνει εκτός της δομής.

Η ιδέα είναι να χρησιμοποιήσουμε μια νέα προσωρινή μεταβλητή ως σημαία που θα αρχικοποιείται στην τιμή 1. Όταν, έστω και μία συνθήκη έχει αποτιμηθεί ως αληθής, η τιμή της προσωρινής μεταβλητής γίνεται 0. Με αυτόν τον τρόπο όταν έχουν ακολουθηθεί όλες οι συνθήκες του Incase ελέγχουμε την τιμή του flag, που θα μας δείχνει αν θα πρέπει να μεταβούμε στην αρχή της δομής (flag = 0) ή ο έλεγχος θα μεταβεί εκτός του incase (flag = 1).

Αξίζει να σημειωθεί ο τρόπος με τον οποίο ο μεταγλωττιστής αναγνωρίζει συναρτήσεις της μορφής :

max(in max(in a, in b), in max(in c, in d)).

Συγκεκριμένα, για το λόγο αυτό χρησιμοποιούνται οι global μεταβλητές,

isFunction: μας λέει αν μια συνάρτηση έχει μια άλλη συνάρτηση ως όρισμα και αρχικοποιείται στο 0.

nest: αρχικοποιείται στο 0 και μας υποδεικνύει το βάθος που έχει μια συνάρτηση μέσα στο όρισμα μιας άλλης συνάρτησης.

args: είναι ένας πίνακας, στον οποίο αποθηκεύονται οι προσωρινές μεταβλητές που περιέχουν την τιμή επιστροφής των εμφωλευμένων συναρτήσεων. Αυτό γίνεται, έτσι ώστε οι συναρτήσεις-ορίσματα της αρχικής συνάρτησης, να εμφανιστούν στο σωστό σημείο των τετράδων του ενδιαμέσου κώδικα.

Πιο συγκεκριμένα η τιμή του nest αυξάνεται κατά 1, όταν 'εισερχόμαστε' στην actualparlist() και μειώνεται κατά 1, όταν 'εξερχόμαστε'. Στη συνέχεια, μέσα στην factor(), το token μας θα έχει την μορφή ID idtail(), αν το idtail() επιστρέψει 1

σημαίνει ότι πρόκειται για συνάρτηση (και το `isFunction` γίνεται 1), αν επιστρέψει 0 το token μας είναι μια μεταβλητή με το όνομα ID. Στην περίπτωση που το `idtail()` επιστρέψει 1, πρέπει να γίνει και ο παρακάτω έλεγχος,

Αν `nest >= 1`, τότε σημαίνει ότι είμαστε σε εμφωλευμένη συνάρτηση και βάζουμε την τετράδα τιμής επιστροφής (`par`, `w`, `RET`, `_`) και κλήσης (`call`, `f`, `_`, `_`) στις τετράδες του αποτελέσματος, `programQuads`. Επίσης, μέσα στην `actualparitem()` αφού το `isFunction` είναι 1, δεν τυπώνουμε κάποια παράμετρο στις τετράδες αλλά κρατάμε στο πίνακα `args` το όρισμα.

Αν `nest = 0`, τότε είμαστε στην αρχική συνάρτηση (την πιο εξωτερική) και μαζί με τις παραμέτρους επιστροφής και κλήσης συνάρτησης, βάζουμε στο αποτέλεσμα και τις παραμέτρους που κρατούνται στο πίνακα `args` (είναι οι τιμές που έχουν επιστρέψει οι εμφωλευμένες συναρτήσεις).

Τέλος, η μέθοδος `generate_int_file()` δημιουργεί το αρχείο `code.int` που περιέχει τις τετράδες του ενδιαμέσου κώδικα και η `generate_c_file()` δημιουργεί το αρχείο `code.c` (αν χρειάζεται). Υπάρχει και η βοηθητική μέθοδος `find_variables()`, που βρίσκει όλες τις μεταβλητές στο αρχείο `code.int` για να αρχικοποιηθούν στο αρχείο `code.c`.

- **Πίνακας συμβόλων:**

Για την αναπαράσταση του Πίνακα Συμβόλων, έχουν χρησιμοποιηθεί οι κλάσεις `Entity`, που αναπαριστά μία οντότητα του πίνακα συμβόλων (δηλ. `Variable`, `function/procedure`, `parameter` και `tmpvar`), `Argument`, που είναι όρισμα μιας συνάρτησης και `Scope`, που είναι μια εγγραφή στο πίνακα συμβόλων (δηλαδή η εμβέλεια μιας συνάρτησης με όλες τις οντότητές της). Να σημειωθεί, ότι κάθε οντότητα του πίνακα συμβόλων έχει και ένα πεδίο `offset`, που μας υποδεικνύει την απόσταση που έχει η οντότητα από το δείκτη στοίβας του εγγράφηματος δραστηριοποίησης στο οποίο βρίσκεται. Το εγγράφημα δραστηριοποίησης δημιουργείται για κάθε συνάρτηση από αυτήν που την καλεί, περιέχει πληροφορίες που χρησιμεύουν στην εκτέλεση και τον τερματισμό της συνάρτησης καθώς και πληροφορίες που σχετίζονται με τις μεταβλητές που χρησιμοποιεί. Η κάθε θέση στο εγγράφημα δραστηριοποίησης καταλαμβάνει 4 bytes και οι πρώτες 3 θέσεις είναι δεσμευμένες για:

- 1) την διεύθυνση επιστροφής (η διεύθυνση στην οποία θα μεταβεί η ροή του προγράμματος όταν ολοκληρωθεί η εκτέλεση της συνάρτησης)
- 2) τον σύνδεσμο προσπέλασης (δείχνει στο εγγράφημα δραστηριοποίησης που πρέπει να αναζητηθούν μεταβλητές οι οποίες δεν είναι τοπικές αλλά η συνάρτηση έχει δικαίωμα να χρησιμοποιήσει)
- 3) την επιστροφή τιμής (η διεύθυνση στην οποία θα γραφεί το αποτέλεσμα της συνάρτησης όταν αυτό υπολογιστεί)



Δεδομένου του παραπάνω, το offset κάθε οντότητας εκκινεί από την τιμή 12 και αυξάνεται κατά 4 για κάθε οντότητα που προστίθεται. Δηλαδή οι θέσεις που βρίσκονται μετά το 12<sup>ο</sup> Byte στο εγγράφημα δραστηριοποίησης, είναι υπεύθυνες για τις προσωρινές μεταβλητές, τις τοπικές μεταβλητές αλλά και τις τυπικές παραμέτρους που μπορεί να έχει μια συνάρτηση.

Η μοντελοποίηση του πίνακα συμβόλων γίνεται ως εξής:

Στην αρχή της μεταγλώττισης προστίθεται ένα νέο Score, με βάθος = 0. Για κάθε μία μεταβλητή που συναντάμε, προσθέτουμε μια καινούργια οντότητα στη λίστα οντοτήτων του τωρινού επιπέδου. Όταν συναντήσουμε συνάρτηση προσθέτουμε νέο Score και λειτουργούμε ανάλογα κρατώντας όσες πληροφορίες χρειαζόμαστε και αυξάνουμε το nesting ώστε να δείχνει στη σωστή τιμή του βάθους φωλιάσματος. Για τις τυπικές παραμέτρους της συνάρτησης, πρέπει να κρατήσουμε ένα 'σημαδάκι' (μεταβλητή place στην formalparameter()), έτσι ώστε να ξέρουμε σε ποιο σημείο του προηγούμενου score βρίσκεται η συνάρτηση που επεξεργαζόμαστε. Στο τέλος, αφού πλέον γνωρίζουμε το μέγεθος της στοίβας της συνάρτησης ενημερώνουμε την μεταβλητή framelength, κάνουμε τον έλεγχο αν κάθε οντότητα σε αυτό το score είναι μοναδική (με την συνάρτηση **is\_unique(name)**: όπου name, το όνομα της συνάρτησης και χρειάζεται έτσι ώστε να τυπώσουμε κατάλληλο μήνυμα σφάλματος) και μετά αφαιρούμε το score της συνάρτησης από τον πίνακα συμβόλων. Σημαντικό επίσης είναι και το σημείο που θα πρέπει να ενημερώσουμε την μεταβλητή startQuad μιας οντότητας συνάρτησης (ή διαδικασίας). Αυτό θα πρέπει να είναι ακριβώς μετά την εντολή `genquad("begin_block", function_name "_", "_")` μέσα στο block της συνάρτησης, έτσι ώστε να μην υπάρχει πρόβλημα στον πίνακα συμβόλων στην περίπτωση που μια συνάρτηση έχει οριστεί μέσα σε μια άλλη και αυτή δεν είναι η main function. Τελικά, ελέγχουμε αν κάθε οντότητα είναι μοναδική και στο Score της main με την `is_unique()`, και έπειτα διαγράφουμε από τη μνήμη το συγκεκριμένο score.

Επιπλέον, έχει υλοποιηθεί η συνάρτηση **search\_entity(name)**, που επιστρέφει ένα πίνακα της μορφής [nestingLevel, Entity], όπου nestingLevel είναι το βάθος φωλιάσματος της οντότητας με όνομα name (την πρώτη φορά που συναντάται στον πίνακα συμβόλων) και Entity, είναι το αντικείμενο τύπου Entity που περιέχει τις πληροφορίες για την οντότητα αυτή. Εδώ γίνεται και ο σημασιολογικός έλεγχος, κατά τον οποίο αν έχει διαβαστεί όλος ο πίνακας συμβόλων και δεν έχει βρεθεί η οντότητα τυπώνεται μήνυμα σφάλματος.

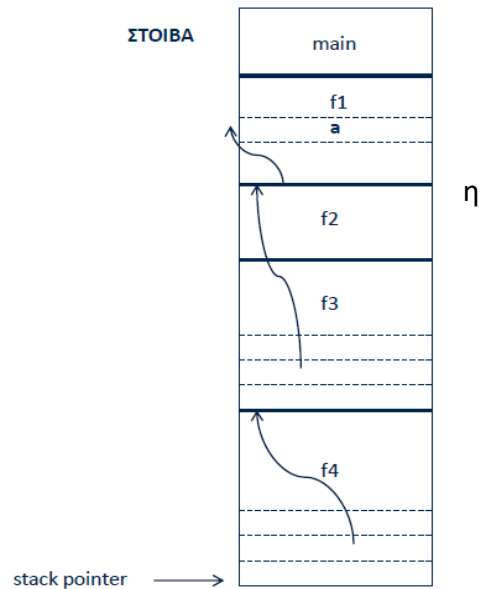
Ο σημασιολογικός έλεγχος ολοκληρώνεται με την συνάρτηση `check_function_parameters()` που ελέγχει αν οι πραγματικές παράμετροι μέσα στην κλήση μιας συνάρτησης ταιριάζουν με τις πραγματικές παραμέτρους της.

- Παραγωγή τελικού κώδικα:

Κατά τη φάση αυτή, από κάθε εντολή του ενδιάμεσου κώδικα, παράγουμε τις αντίστοιχες εντολές του τελικού κώδικα για τον επεξεργαστή MIPS.

Έχουν χρησιμοποιηθεί κάποιες βοηθητικές υπορουτίνες, που είναι οι παρακάτω:

**gnlvcode(x)** : ψάχνει στον πίνακα συμβόλων για την μη τοπική μεταβλητή με όνομα x, την μεταφέρει στον καταχωρητή \$t0, αφού έχει βρει από τον πίνακα συμβόλων πόσα επίπεδα επάνω βρίσκεται μη τοπική μεταβλητή x, την εντοπίζει μέσα από τον σύνδεσμο προσπέλασης.



**loadvr(v, r)**: ψάχνει για την οντότητα με όνομα v στον πίνακα συμβόλων, βρίσκει την διεύθυνση της στη μνήμη και την μεταφέρει στον καταχωρητή r (που είναι ένας εκ των, \$t0, &t1, \$t2). Αν το v πρόκειται να είναι κάποια σταθερά, απλά καταχωρείται η τιμή της στον καταχωρητή r. Ανάλογα με τον τύπο οντότητας της v διακρίνουμε περιπτώσεις που θα δούμε παρακάτω.

**storerv(r, v)**: μεταφέρει δεδομένα από τον καταχωρητή r στη μνήμη και συγκεκριμένα στην μεταβλητή v. Ομοίως και εδώ διακρίνουμε περιπτώσεις.

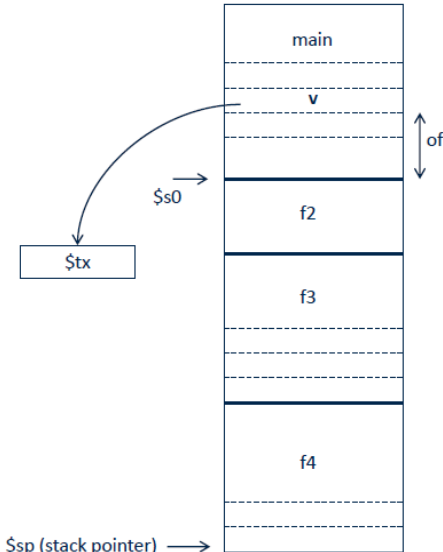
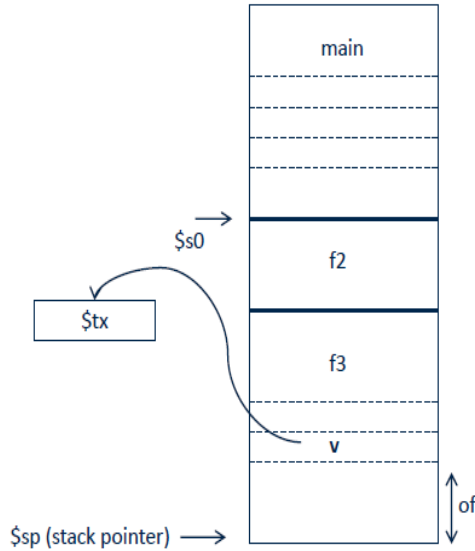
Παρακάτω παρουσιάζεται ο τρόπος κατασκευής της μεθόδου **gnlvcode(x)**, αλλά και της **loadvr(v, r)**. Η λογική κατασκευής της **storerv(r, v)** είναι παρόμοια με αυτή της **loadvr(v, r)**.

Η λογική κατασκευής της gnlvcode(x):

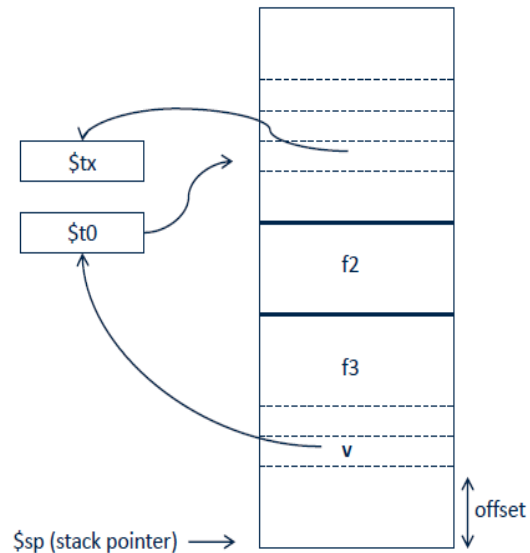
Αρχικά, μεταφέρουμε στον καταχωρητή \$t0 την στοίβα του γονέα παράγοντας την εντολή (`lw $t0, -4($sp)`). Έπειτα, όσες φορές χρειαστεί, σύμφωνα και με την διαφορά επιπέδων από το τρέχον επίπεδο που έχει η μη τοπική μεταβλητή, προσθέτουμε την εντολή (`lw $t0, -4($t0)`), δηλαδή μεταφέρουμε στον \$t0, τη στοίβα του προγόνου που έχει τη μεταβλητή. Στο τέλος, αφού έχουμε βρεί το επίπεδο που ανήκει η μη τοπική μεταβλητή, απλά μεταφέρουμε στον \$t0 τη διεύθυνση που έχει αυτή στη στοίβα (`addi $t0, $t0, -offset`).

### Η λογική κατασκευής της loadvr(v, r):

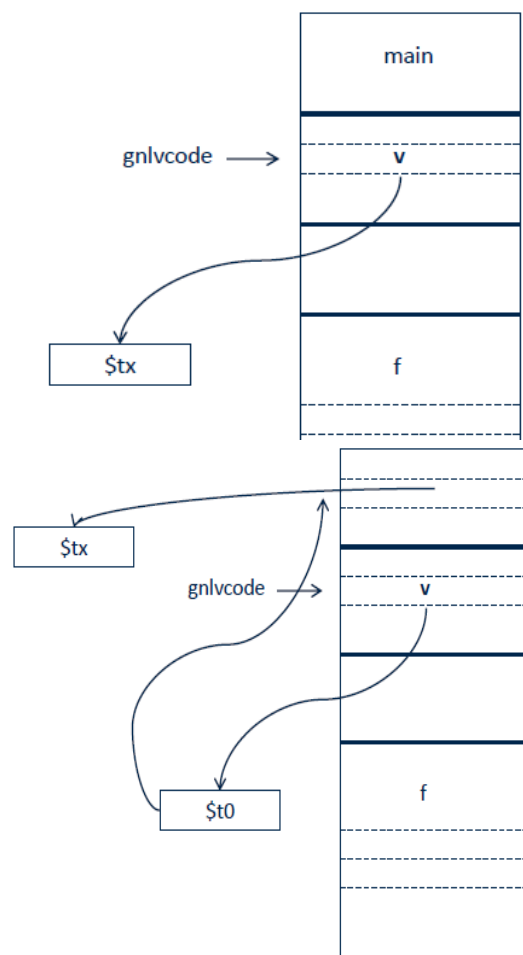
Όπως αναφέρθηκε, οφείλουμε να διακρίνουμε περιπτώσεις. Αυτές είναι:

- Η v είναι σταθερά: τότε απλά μεταφέρουμε την τιμή της σταθεράς v, στον καταχωρητή με την εντολή (li \$tr, v)
- Η v είναι καθολική μεταβλητή:  
βρίσκουμε από τον πίνακα συμβόλων το offset της μεταβλητής και παράγουμε την εντολή (lw \$tr, -offset(\$s0)). Να σημειωθεί ότι κατά την εκκίνηση της μεταγλώττισης, κρατάμε στον καταχωρητή \$s0 το εγγράφημα δραστηριοποίησης της main για εύκολη πρόσβαση στις global μεταβλητές.
- Η v έχει δηλωθεί στη συνάρτηση που αυτή τη στιγμή εκτελείται και είναι τοπική μεταβλητή, ή τυπική παράμετρος που περνάει με τιμή, ή προσωρινή μεταβλητή: όταν ισχύει κάποιο από τα παραπάνω, γνωρίζουμε ότι η διεύθυνση της v βρίσκεται κάπου στη στοίβα της συνάρτησης που βρίσκεται υπό μεταγλώττιση. Επομένως, βρίσκουμε από τον πίνακα συμβόλων το offset της και παράγουμε την εντολή (lw \$tr, -offset(\$sp)).

- Η  $v$  έχει δηλωθεί στη συνάρτηση που αυτή τη στιγμή εκτελείται και είναι τυπική παράμετρος που περνάει με αναφορά: εφόσον πρόκειται για παράμετρο που έχει περαστεί με αναφορά, γνωρίζουμε ότι η διεύθυνσή της βρίσκεται στη στοίβα της συνάρτησης που μεταγλωττίζεται. Συνεπώς, πρωτίστως βάζουμε στον  $\$t0$  τη διεύθυνση της παραμέτρου (`lw  $\$t0$ , -offset( $\$sp$ )`) και ακολούθως τοποθετούμε στον καταχωρητή αποτελέσματος την τιμή που υπάρχει σε αυτή τη διεύθυνση (`lw  $\$tr$ , ( $\$t0$ )`).



- Η  $v$  έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι είτε τοπική μεταβλητή είτε τυπική παράμετρος που περνάει με τιμή: οφείλουμε να κάνουμε τις κατάλληλες ενέργειες που περιγράφηκαν στην `gnlvcode(x)`, έτσι ώστε να βρούμε τη διεύθυνση της μη τοπικής μεταβλητής στον πρόγονο (κλήση της μεθόδου `gnlvcode(v)`), και έπειτα, γνωρίζοντας ότι ο  $\$t0$  περιέχει την επιθυμητή διεύθυνση, βάζουμε στον  $\$tr$  την τιμή της μεταβλητής (`lw  $\$tr$ , ( $\$t0$ )`).



- Η  $v$  έχει δηλωθεί σε κάποιο πρόγονο και εκεί είναι τυπική παράμετρος που περνάει με

αναφορά: θα πρέπει να καλέσουμε την `gblncode(v)`, όμως λόγω του περάσματος με αναφορά, ο `$t0` θα περιέχει τη διεύθυνση της μεταβλητής στη στοίβα του επιπέδου όπου ανήκει, που θα είναι και πάλι διεύθυνση. Συνεπώς, μετά την κλήση της `gblncode(v)`, παράγουμε τις 2 παρακάτω εντολές για να βάλουμε την τιμή της μεταβλητής στον επιθυμητό καταχωρητή, (`lw $t0, ($t0)`) και (`lw $tr, ($t0)`).

Έχοντας υλοποιήσει τις παραπάνω βοηθητικές μεθόδους, πλέον η παραγωγή του τελικού κώδικα καθίσταται αρκετά πιο εύκολη και κατανοητή. Η βασική ιδέα είναι από κάθε εντολή που έχουμε παράγει για τον ενδιαμέσο κώδικα, να εκτελέσουμε τις κατάλληλες ενέργειες και κλήσεις των παραπάνω μεθόδων, για την παραγωγή του τελικού κώδικα MIPS.

Παρακάτω παρουσιάζεται η βασική ιδέα αυτής της μετατροπής με βάση των τελεστή στις τετράδες του ενδιαμέσου κώδικα.

- **(relop x, y, z)**, `relop` μπορεί να είναι οποιοσδήποτε από (<, >, =, >=, <=, <>)  
    `loadvr(x, $t1)`: βάλε στον `$t1` την τιμή του `x`  
    `loadvr(y, $t2)`: παρομοίως  
    `branch $t1, $t2, z`: εκτέλεσε την επιθυμητή λογική πράξη,  
    `branch` μπορεί να είναι {`blt`:<, `bgt`:>, `ble`:<=, `bge`:>=, `beq`:=, `bne`:<>}
- **(:=, x, \_, z)**,  
    `loadvr(x, $t1)`  
    `storerv($t1, z)`
- **(out, x, \_, \_)**,  
    `(li $v0, 1)`  
    `loadvr(x, $a0)`  
    `syscall`
- **(retv, x, \_, \_)**,  
    `loadvr(x, $t1)`  
    `(lw $t0, -8($sp))`  
    `(sw $t1, ($t0))`  
    Ο `x` αποθηκεύεται στη διεύθυνση που είναι αποθηκευμένη στην 3<sup>η</sup> θέση του εγγραφήματος δραστηριοποίησης (που είναι υπεύθυνο για την επιστροφή τιμής).

Ενδιαφέρον παρουσιάζει η μέθοδος που ακολουθούμε όταν έχουμε να μετατρέψουμε παραμέτρους κάποιας συνάρτησης.

- Πριν κάνουμε τις ενέργειες ώστε να γίνει το πέρασμα της πρώτης παραμέτρου, τοποθετούμε τον `$fp` να δείχνει στην στοίβα της συνάρτησης που θα δημιουργηθεί (`addi, $fp, $sp, framelength`)
- Στη συνέχεια, για κάθε παράμετρο και ανάλογα με το αν περνά με τιμή ή αναφορά κάνουμε τις εξής ενέργειες:
  1. **(par, x, CV, \_)**:

loadvr(x, \$t0)  
(sw \$t0, -(12+4i)(\$fp)), όπου i ο αύξων αριθμός της παραμέτρου (0...)

2. (par, x, REF, \_):

- Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
**(addi \$t0, \$sp, -offset)**  
**(sw \$t0, -(12+4i)(\$fp))**
- Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν το ίδιο βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά:  
**(lw \$t0, -offset(\$sp))**  
**(sw \$t0, -(12+4i)(\$fp))**
- Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή:  
**gnlvcode(x)**  
**(sw \$t0, -(12+4i)(\$fp))**
- Αν η καλούσα συνάρτηση και η μεταβλητή x έχουν διαφορετικό βάθος φωλιάσματος, η παράμετρος x είναι στην καλούσα συνάρτηση παράμετρος που έχει περαστεί με αναφορά:  
**gnlvcode(x)**  
**(lw \$t0, (\$t0))**  
**(sw \$t0, -(12+4i)(\$fp))**

3. (par, x, RET, \_):

Γεμίζουμε το 3<sup>ο</sup> πεδίο του εγγραφύηματος δραστηριοποίησης της κληθείσας συνάρτησης με τη διεύθυνση της προσωρινής μεταβλητής στην οποία θα επιστραφεί η τιμή.

**(addi \$t0, \$sp, -offset)**  
**(sw \$t0, -8(\$sp))**

Στη συνέχεια, αφού έχουμε λάβει όλες τις απαραίτητες ενέργειες για τις παραμέτρους της συνάρτησης, ακολουθεί η κλήση της:

**(call, f, \_, \_):**

Αρχικά γεμίζουμε το 20 πεδίο του εγγραφύηματος δραστηριοποίησης της κληθείσας συνάρτησης, τον σύνδεσμο προσπέλασης, με την διεύθυνση του εγγραφύηματος

δραστηριοποίησης του γονέα της, ώστε η κληθείσα να γνωρίζει που να κοιτάξει αν χρειαστεί να προσπελάσει μία μεταβλητή την οποία έχει δικαίωμα να προσπελάσει, αλλά δεν της ανήκει.

- Αν καλούσα και κληθείσα έχουν το ίδιο βάθος φωλιάσματος, τότε έχουν τον ίδιο γονέα:  
**(lw \$t0, -4(\$sp))**  
**(sw \$t0, -4(\$sp))**
- Αν καλούσα και κληθείσα έχουν διαφορετικό βάθος φωλιάσματος, τότε η καλούσα είναι ο γονέας της κληθείσας:  
**(sw \$sp, -4(\$fp))**

Έπειτα, μεταφέρουμε τον δείκτη της στοίβας στην κληθείσα:

**(addi \$sp, \$sp, framelength)**

Καλούμε την συνάρτηση:

**(jal f)**

Όταν επιστρέψουμε παίρνουμε πίσω τον δείκτη στοίβας στην καλούσα:

**(addi \$sp, \$sp, -framelength)**

Σημαντική σημείωση:

Στην αρχή κάθε συνάρτησης αποθηκεύουμε στην πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της, την οποία έχει τοποθετήσει στον \$ra η jal:

**(sw \$ra, (\$sp))**

Στο τέλος κάθε συνάρτησης κάνουμε το αντίστροφο, παίρνουμε από την πρώτη θέση του εγγραφήματος δραστηριοποίησης την διεύθυνση επιστροφής της συνάρτησης και την βάζουμε στον \$ra. Έπειτα επιστρέφουμε στη καλούσα:

**(lw \$ra, (\$sp))**

**(jr \$ra)**

Επειδή το κυρίως πρόγραμμα δεν είναι το πρώτο πράγμα που μεταφράζεται, πρέπει στην αρχή του προγράμματος να βάλουμε μια εντολή άλματος που να οδηγεί στην πρώτη ετικέτα του κυρίως προγράμματος: **(j Lmain)**

Στη συνέχεια κατεβάζουμε τον \$sp κατά framelength της main:

**(addi, \$sp, \$sp, framelength)**

Η μεταγλώττιση του κώδικα γίνεται με την εντολή:

**python3 cimple\_3330.py program\_name.ci**

όπου program\_name.ci είναι το όνομα του αρχείου σε πηγαίο κώδικα C-imple.

Κατά την μεταγλώττιση παράγονται τα εξής αρχεία:

1. **code.int:** περιέχει τις παραγόμενες εντολές κατά τη φάση του ενδιάμεσου κώδικα
2. **code.c:** παράγεται μόνο όταν ο πηγαίος κώδικας **δεν** περιέχει κάποια άλλη συνάρτηση εκτός της main και είναι το ισοδύναμο αρχείο σε c, για πιο εύκολο έλεγχο της ορθότητας του παραγόμενου κώδικα.
3. **code.asm:** περιέχει τις εντολές του τελικού κώδικα που έχει παράγει ο μεταγλωττιστής και είναι προς εκτέλεση στο εργαλείο MARS του επεξεργαστή MIPS.
4. **Scopes.txt:** περιέχει μια πρόχειρη μορφή του πίνακα συμβόλων για να διαπιστωθεί η ορθότητά.