ΜΥΥ802 – ΜΕΤΑΦΡΑΣΤΕΣ

ΑΝΑΦΟΡΑ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΗΣ ΑΣΚΗΣΗΣ

Μέλη Ομάδας:

Κεφάλας Γιώργος Α.Μ. : 5252

Παπαδόπουλος Δημήτρης Α.Μ. : 5321

ΙΩΑΝΝΙΝΑ 2025

Εισαγωγή

Η συγκεκριμένη προγραμματιστική άσκηση αφορά την υλοποίηση ενός μεταφραστή για την γλώσσα greek++.

Στην αναφορά που ακολουθεί αναλύουμε σε φάσεις το πως εκπονήθηκε η προγραμματιστική άσκηση, καθώς μετά το τέλος της κάθε φάσης αναφέρουμε ενδεικτικά αποτελέσματα από ορισμένα test.

Οι κεντρικοί άξονες στους οποίους θα κινηθούμε είναι:

- Λεκτικός αναλυτής
- Συντακτικός Αναλυτής
- Ενδιάμεσος Κώδικας
- Πινάκας Συμβόλων
- Τελικός Κώδικας

Λεκτικός αναλυτής

Ο λεκτικός αναλυτής αποτελεί την πρώτη φάση της μεταγλώττισης. Σε αυτό το στάδιο διαβάζεται το αρχικό πρόγραμμα και παράγονται οι λεκτικές μονάδες (token).

Ο λεκτικός αναλυτής υλοποιείται ως μια συνάρτηση , της οποίας πρωταρχική λειτουργία είναι να διαβάζει έναν-έναν τους χαρακτήρες του αρχικού προγράμματος και να επιστρέφει την επόμενη λεκτική μονάδα.

Ο δευτερεύων ρόλος του λεκτικού αναλυτή είναι η αναγνώριση σφαλμάτων, ώστε να πληροφορήσει τον συντακτικό αναλυτή ή να διακόψει τη μεταγλώττιση και να επιστρέψει στον χρήστη ένα μήνυμα λάθους.

Υλοποίηση

Η υλοποίηση του λεκτικού αναλυτή βασίστηκε στην αντιμετώπιση της λεκτικής μονάδας ως ένα αντικείμενο με τρία βασικά πεδία:

- Το string το οποίο αναγνώρισε (recognized string)
- Την κατηγορία στην οποία ανήκει αυτή η συμβολοσειρά (family)
- Ο αριθμός γραμμής στην οποία εντοπίστηκε η συμβολοσειρά (line number)

Ο κώδικας ξεκίνα ορίζοντας τις κατηγορίες στις οποίες πρέπει να ανήκει κάθε λεκτική μονάδα. Αυτές είναι:

- identifier(id) : ονόματα μεταβλητών, συναρτήσεων, διαδικασιών, σταθερών και ό,τι άλλο μπορεί να έχει ονομασία σε ένα πρόγραμμα

- number : αριθμητικές σταθερές

- addOperator: οι αριθμητικοί τελεστές +,-

- minusOperetor : οι αριθμητικοί τελεστές * , /

- relOperator : οι λογικοί τελεστές, π.χ: ==, <

- assignment: τελεστής εκχώρησης (:=)

- delimeter: διαχωριστές , π.χ.: ;, . , ,

- groupOperator: συμβολα ομαδοποίησης , π.χ.: (,) , {,} , [,]

• byreference: Χρησιμοποιείται όταν περνάμε παραμέτρους με αναφορά. Αναγνωρίζεται η λεκτική μονάδα “%”.

• error: Ειδική κατηγορία για την περίπτωση που η λεκτική μονάδα δεν ανήκει σε κάποια από τις παραπάνω κατηγορίες , με αυτό να σημαίνει πως ο συγκεκριμένος χαρακτήρας είναι άγνωστος για την γλώσσα.

• Eof: Κατηγορία ώστε να αναγνωρίζεται ο χαρακτήρας κενού που σηματοδοτεί το τέλος του αρχείου.

• Keyword: Κατηγορία στην οποία ανήκουν όλες οι λέξεις οι οποίες είναι δεσμευμένες από την γλώσσα , ώστε να επιτελεί τις διάφορες λειτουργίες που υποστηρίζει.

Οι δεσμευμένες λέξεις τις greek++ είναι :

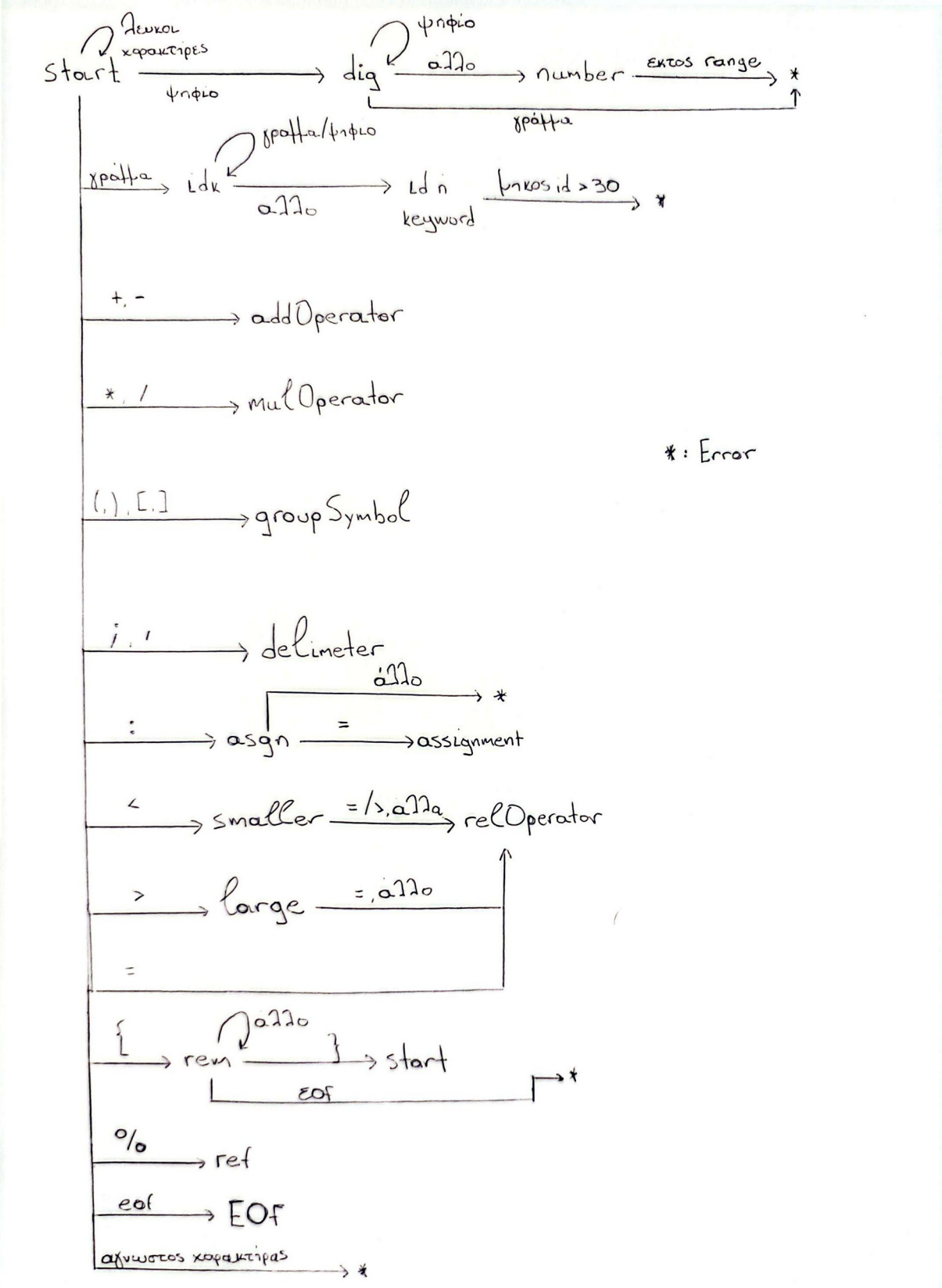
"πρόγραμμα", "εάν" , "επανάλαβε" , "για" , "διάβασε" , "συνάρτηση" , "είσοδος" , "αρχή_συνάρτησης" , "αρχή_διαδικασίας" , "αρχή_προγράμματος", "ή" , "δήλωση" , "τότε" , "μέχρι" , "έως" , "γράψε", "διαδικασία" , "έξοδος" , "και" , "αλλιώς" , "όσο" , "με_βήμα" , "διαπροσωπεία" , "τέλος_συνάρτησης" , "τέλος_διαδικασίας" , "τέλος_προγράμματος", "όχι" , "εκτέλεσε" , "εάν_τέλος", "όσο_τέλος", "για_τέλος"

Έπειτα στον κώδικα ακολουθεί ο ορισμός της κλάσης Token που αντιπροσωπεύει την κάθε λεκτική μονάδα. Η κλάση αυτή έχει τα πεδία που προαναφέρθηκαν (recongnized string, family, line number) και αρχικοποιούνται με αυτά.

Τέλος υπάρχει μια εντός της κλάσης συνάρτηση __repr__ , η οποία επιστρέφει ένα string το οποίο αναπαριστά κάθε φορά το αντικείμενο , δηλαδή την λεκτική μονάδα.

Στη συνέχεια υλοποιείται η συνάρτηση lex() , η οποία αποτελεί τον λεκτικό αναλυτή του μεταφραστή και επιστρέφει κάθε φορά την επόμενη λεκτική μονάδα. Αυτό σημαίνει πως σε κάθε κλήση του, σημειώνει το σημείο στο οποίο σταματάει την ανάγνωση του αρχικού προγράμματος και την επόμενη φορά που καλείται συνεχίζει την ανάγνωση από το σημείο αυτό.

Στη συνάρτηση lex() καλούμαστε να υλοποιήσουμε το παρακάτω αυτόματο που αναγνωρίζει τις λεκτικές μονάδες:



Εντός της συνάρτησης lex υπάρχει ένα while-loop το οποίο υλοποιεί το αυτόματο. Συγκεκριμένα, εντός αυτού του loop υπάρχουν μια σειρά από if-statements , ώστε κάθε χαρακτήρας που διαβάζεται από το αρχείο να κατηγοριοποιείται κατάλληλα.

• Πρώτη κατηγορία που συναντάμε στον κώδικα είναι αυτή του eof που σηματοδοτεί το τέλος του αρχείου. Αυτό το if-statement θα είναι αληθής στην περίπτωση που ο επόμενος χαρακτήρα θα είναι κενός ή άγνωστος (�).

• Επόμενη περίπτωση είναι αυτή του αλφαριθμητικού, οπού .Συγκεκριμένα, εντός αυτής της περίπτωσης αναθέτουμε το υπάρχων χαρακτήρα που έχει διαβαστεί στην global μεταβλήτη token , ώστε να μην χαθεί. Έπειτα ακολουθεί ένα while-loop στο οποίο συνεχίζουμε να διαβάζουμε χαρακτήρες από το αρχείο και στην περίπτωση που δεν είναι αλφαριθμητικό ή ο χαρακτήρας “_” οδηγούμε τον pointer του αρχείου μία θέση πίσω και ο κώδικας συνεχίζει να εκτελείται εκτός του while-loop. Στο σενάριο που ο χαρακτήρας είναι αλφαριθμητικό ή ο χαρακτήρας “_” , τότε τον προσθέτουμε στην μεταβλήτη token για να συμπληρωθεί σταδιακά ολόκληρη η λέξη που διαβάζεται από το αρχείο.

Έχοντας ολόκληρη την λέξη στη μεταβλήτη token , εξετάζουμε πλέον εκτός του while-loop εάν η λέξη αυτή ανήκει στις δεσμευμένες , από τη γλώσσα , λέξεις ή όχι. Στην περίπτωση που ανήκει ο λεκτικός αναλυτής επιστρέφει ένα αντικείμενο Token με το πεδιό family να είναι Reserved, ενώ σε οποιαδήποτε άλλη περίπτωση το αντικείμενο Token που θα επιστραφεί θα έχει για πεδίο family το Id.

Να σημειωθεί πως στην περίπτωση που το Token ανήκει στην κατηγορία id, ακολουθεί έλεγχος για ενδεχόμενο λεκτικό σφάλμα. Συγκεκριμένα άμα η λέξη που κρατά η μεταβλητή token είναι παραπάνω από 30 χαρακτήρες, τότε τυπώνεται μήνυμα σφάλματος.

• Ακολουθεί η περίπτωση που ο χαρακτήρας που διαβάζουμε είναι ψηφίο. Αναλυτικότερα, αρχικοποιούμε όπως και στην περίπτωση του αλφαριθμητικού μία μεταβλήτη token για να συμπληρωθεί σταδιακά ο αριθμός που διαβάζουμε από το αρχείο. Ο αριθμός αυτός συμπληρώνεται με τη βοήθεια ενός while-loop που εντός του διαβάζουμε τον επόμενο χαρακτήρα και διακρίνουμε εάν είναι ψηφίο ή όχι. Αν ο χαρακτήρας αυτός είναι ψηφίο το προσθέτουμε στην μεταβλητή token, ώστε να καταφέρουμε να πάρουμε ολόκληρο τον αριθμό που διαβάσαμε από το αρχείο.

Στην περίπτωση που ο χαρακτήρας δεν είναι ψηφίο, τότε με τον αριθμό που έχει ανατεθεί ήδη στη μεταβλητή token εξετάζουμε ενδεχόμενα λεκτικά σφάλματα. Το πρώτο που εξετάζεται είναι αν ο αριθμός αυτός δεν ανήκει στο πεδίο [-32767,32767] , ενώ ο δεύτερος έλεγχος αφορά την περίπτωση που έχουμε

αλφαριθμητικό μετά από ψηφίο. Γίνεται εύκολα αντιληπτό πως και τα δύο ενδεχόμενα υπάρχουν μηνύματα σφάλματος εάν αποδειχθούν αληθή.

Τέλος εάν ο χαρακτήρας δεν αποτελεί ψηφίο οδηγούμε τον pointer του αρχείου μία θέση πίσω , ώστε στην επόμενη κλήση της συνάρτησης lex να το ξαναδιαβάσει και να το επιστρέψει ως αντικείμενο Token με το ανάλογο family ως πεδίο.

• Οι επόμενες περιπτώσεις που ο χαρακτήρας που διαβάζουμε είναι : “<” ,”>” και “:”. Στις τρεις αυτές περιπτώσεις χρειάζεται να διαβάσουμε τον επόμενο χαρακτήρα προκειμένου να γίνει η σωστή κατηγοριοποίηση των επικείμενων αντικειμένων Token.

Η πρώτη περίπτωση είναι αυτή που διαβάζουμε τον χαρακτήρα “<”. Εδώ εξετάζουμε αρχικά άμα ο επόμενος χαρακτήρας είναι το ”=” επιστρέφοντας αντικείμενο Token με family το LESSEQ(relOperator). Δεύτερο ενδεχόμενο που εξετάζουμε είναι που ο επόμενος χαρακτήρας είναι το “>” , ώστε το ανάλογο Token να έχει για family το ANGBR(relOperator). Στη συνέχεια, αφού ο χαρακτήρας δεν είναι το ”=” ή το ”>” το αντικείμενο Token που επιστρέφεται, έχοντας τον pointer του αρχείου μία θέση πίσω , έχει για family το LESS(relOperator).

Παρόμοια περίπτωση είναι αυτή του χαρακτήρα “>”. Εδώ το μόνο ενδεχόμενο που εξετάζεται είναι αυτή που ο επόμενος χαρακτήρας είναι το ”=” , με το family του Token να είναι το MOREEQ(relOperator). Εναλλακτικά το family του Token είναι το MORE(relOperator) που αντιστοιχεί στον χαρακτήρα “>”.

Τελευταία παρόμοια κατάσταση είναι αυτή του χαρακτήρα ”:”. Εδώ το family του αντικείμενου Token ενδέχεται να είναι το ASSIGN(assignment) εφόσον συναντήσουμε ως επόμενο χαρακτήρα το “=” , ενώ σε άλλη περίπτωση υπάρχει λεκτικό σφάλμα με το αντίστοιχο μήνυμα σφάλματος.

• Ακολουθούν οι περιπτώσεις στις οποίες ο χαρακτήρας είναι ένας από τους ακόλουθους : “+” , “-” , “*” , “/” , “=”, “,” , “;” , “(” , “)” , “[” , “]” , ”%”. Η μοναδική ενέργεια που γίνεται είναι η επιστροφή τους ως αντικείμενα Token με το κατάλληλο πεδίο family συμπληρωμένο.

• Τελευταία ιδιάζουσα περίπτωση είναι αυτή που χειριζόμαστε τα σχόλια που εισάγει ο χρήστης. Συγκεκριμένα μόλις συναντήσουμε τον χαρακτήρα “{” ενεργοποιούμε το flag (rem) που έχουμε ορίσει στην αρχή της συνάρτησης. Στην επόμενη κλήση του λεκτικού αναλυτή και εφόσον το flag είναι True τότε το πρόγραμμα ανακατευθύνεται σε ένα if-statement , χειρίζοντας ορισμένες

καταστάσεις. Στην περίπτωση που ο επόμενος χαρακτήρας είναι το "}" το flag γίνεται false σηματοδοτώντας το κλείσιμο των σχόλιων . Από την άλλη άμα το flag είναι True και ο επόμενος χαρακτήρας είναι το κενό , σημαίνει πως έχουμε φτάσει στο τέλος του αρχείου και τα σχόλια δεν έχουν κλείσει. Το τελευταίο ενδεχόμενο αποτελεί λεκτικό σφάλμα , γι'αυτό εισάγουμε και το αντίστοιχο μήνυμα σφάλματος.

Τέλος , η συνάρτηση lex τελειώνει με ένα else-statement στο οποίο καταλήγουμε εφόσον δεν συναντήσουμε κάποιον από τους παραπάνω χαρακτήρες, οδηγώντας σε ένα άλλο λεκτικό σφάλμα καθώς ο χαρακτήρας είναι άγνωστος ως προς τη γλώσσα τυπώνοντας το αντίστοιχο μήνυμα σφάλματος.

Testing

Ακολουθούν screenshot από την εκτέλεση τριών από τα παραδείγματα που έχουμε επισυνάψει. Συγκεκριμένα, ακολουθούν μηνύματα λάθους στις περιπτώσεις που γίνεται κάποια λάθος ανάθεση, ανάθεση κάποιου μεγάλου αριθμού και ανάθεση κάποιου άκυρου χαρακτήρα.

Λάθος ανάθεση:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_lex_error_3.gr
Lectical Error at line 6:  ':' is an invalid expression. Expected ':='.
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$
```

Ανάθεση κάποιου μεγάλου αριθμού:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_lex_error_5.gr

Lectical Error at line 6: number 120000 is out of range.
```

Ανάθεση κάποιου άκυρου χαρακτήρα:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_lex_error_6.gr

Lectical Error at line 6: '#' doesn't belong as character in Greek++.
```

Συντακτικός αναλυτής

Η συντακτική ανάλυση (syntax analysis) αποτελεί ένα από τα βασικότερα στάδια στη διαδικασία ανάπτυξης ενός μεταφραστή. Σκοπός της είναι να επαληθεύσει ότι η ακολουθία των λεκτικών μονάδων (tokens) που παράγονται από τον λεκτικό αναλυτή συντάσσονται σύμφωνα με τους κανόνες της γραμματικής της γλώσσας που υλοποιείται (της γλώσσας Greek++ στην περίπτωσή μας).

Η διαδικασία της συντακτικής ανάλυσης στηρίζεται στην εφαρμογή μιας τυπικής γραμματικής, η οποία έχει σχεδιαστεί σε μορφή κανόνων παραγωγής. Κάθε κανόνας καθορίζει πώς συντίθενται οι διάφορες συντακτικές δομές της γλώσσας, όπως οι δηλώσεις μεταβλητών, οι εκφράσεις, οι εντολές ελέγχου ροής και τα υποπρογράμματα.

Ο συντακτικός αναλυτής που αναπτύχθηκε βασίζεται στη μέθοδο καθοδικής ανάλυσης και συγκεκριμένα στην τεχνική της αναδρομικής κατάβασης. Η μέθοδος αυτή χαρακτηρίζεται από την αντιστοίχιση κάθε κανόνα της γραμματικής σε μία συνάρτηση στον πηγαίο κώδικα του αναλυτή, η οποία καλείται αναδρομικά ώστε να αναγνωρίσει τις αντίστοιχες δομές στο πρόγραμμα εισόδου. Κάθε συνάρτηση χειρίζεται άμεσα τα tokens, χωρίς τη χρήση ενδιάμεσων βοηθητικών λειτουργιών, ακολουθώντας πιστά τη γραμματική περιγραφή της Greek++ .΄Επιπλέον η αναδρομική κατάβαση διευκολύνει την επεκτασιμότητα του κώδικα, καθώς κάθε συντακτική δομή μπορεί να διαχειρίζεται απομονωμένα μέσα στις αντίστοιχες συναρτήσεις.

Οι 3 βασικές λειτουργίες του συντακτικού αναλυτή είναι οι εξής:

1) Ελέγχει τη σωστή αλληλουχία των tokens που επιστρέφει ο λεκτικός αναλυτής
2) Εξασφαλίζει ότι οι εκφράσεις και οι δηλώσεις συμμορφώνονται με το συντακτικό της γλώσσας
3) Ανιχνεύει συντακτικά σφάλματα και αναφέρει κατάλληλα μηνύματα για τον τελικό χρήστη

Η ανάπτυξη του συντακτικού αναλυτή έγινε με γνώμονα τη μέγιστη αντιστοιχία μεταξύ της περιγραφής της γραμματικής και της κωδικοποίησής της σε συναρτήσεις, εξασφαλίζοντας έτσι σαφήνεια και συνέπεια. Αυτό θα φανεί ιδιαίτερα χρήσιμο για μετέπειτα στάδια του μεταγλωττιστή, όπως η δημιουργία ενδιάμεσου κώδικα, καθώς η εύρεση ενός συγκεκριμένου σημείου της μετάφρασης θα είναι ευκολότερη. Η παραπάνω προσέγγιση διευκολύνθηκε επίσης αρκετά λόγω της απλής και καθαρής γραμματικής της Greek++.

Ακόμα, σε περίπτωση που συναντηθεί μια ακολουθία tokens που δεν συμφωνεί με τη γραμματική, το σύστημα παράγει μήνυμα λάθους και διακόπτει την περαιτέρω ανάλυση. Η έγκαιρη ανίχνευση τέτοιων λαθών είναι κρίσιμη για τη

διασφάλιση της εγκυρότητας του προγράμματος πριν προχωρήσει στα επόμενα στάδια της μετάφρασης.

Η διαδικασία συντακτικής ανάλυσης της Greek++ έχει οργανωθεί με τρόπο που ακολουθεί πιστά τη δομή της γραμματικής. Η ανάλυση ξεκινά από τον κορυφαίο μη-τερματικό κανόνα program, ο οποίος ορίζει τη βασική μορφή ενός πλήρους προγράμματος στη γλώσσα. Από εκεί και πέρα, η ροή ακολουθεί τα εξής βασικά στάδια:

1) Ανάλυση Προγράμματος

Η συνάρτηση που αναλύει το program αναμένει την εμφάνιση της λέξης-κλειδιού “πρόγραμμα”, ακολουθούμενη από ένα αναγνωριστικό (ID) και την έναρξη του κυρίου μπλοκ προγράμματος (programblock). Το programblock περιλαμβάνει αρχικά την ανάλυση δηλώσεων (declarations) και υποπρογραμμάτων (subprograms) και στη συνέχεια το κύριο σώμα του προγράμματος (sequence).

2) Ανάλυση Δηλώσεων Και Υποπρογραμμάτων

Οι δηλώσεις (declarations) αποτελούνται από πολλαπλές προτάσεις “δήλωση”, όπου κάθε μία περιέχει μία λίστα μεταβλητών (varlist). Τα υποπρογράμματα (subprograms) μπορεί να είναι συναρτήσεις (func) ή διαδικασίες (proc), τα οποία με τη σειρά τους έχουν ξεχωριστό μπλοκ δηλώσεων και σώματος.

3) Ανάλυση Σειρών Εντολών

Το sequence αποτελείται από μία ή περισσότερες εντολές (statement), χωρισμένες με ελληνικό ερωτηματικό σημείο “;”. Κάθε statement μπορεί να είναι μία από τις προκαθορισμένες δομές: ανάθεση (assignment_stat), εκτέλεση υπό συνθήκη (if_stat), επανάληψη (while_stat, do_stat, for_stat), εισαγωγή (input_stat), εκτύπωση (print_stat) ή κλήση διαδικασίας (call_stat).

4) Ανάλυση Εκφράσεων και Συνθηκών

Η ανάλυση εκφράσεων (expression) γίνεται ιεραρχικά, αναλύοντας πρώτα το προαιρετικό πρόσημο, κατόπιν τους όρους (term) και τέλος τα αθροίσματα ή αφαιρέσεις (add_oper). Οι λογικές συνθήκες (condition) αναλύονται μέσω λογικών όρων (boolterm) και παραγόντων (boolfactor), υποστηρίζοντας και τις σύνθετες λογικές συνθέσεις με “ή”, “και”, “‘όχι”.

5) Χειρισμός Λίστας Παραμέτρων και Κλήσεων

Στη διαχείριση κλήσεων υποπρογραμμάτων, η idtail διαχωρίζει αν το αναγνωριστικό ακολουθείται από λίστα πραγματικών παραμέτρων (actualpars) ή όχι.

Διαχείριση Συντακτικών Σφαλμάτων

Κατά την εκτέλεση της συντακτικής ανάλυσης, είναι πιθανό το πρόγραμμα εισόδου να μην ακολουθεί πλήρως τη γραμματική της γλώσσας. Σε αυτές τις περιπτώσεις ο συντακτικός αναλυτής μας εντοπίζει το σφάλμα και ενημερώνει τον χρήστη με σαφές μήνυμα και διακόπτει την εκτέλεση της μετάφρασης. Κάθε φορά λοιπόν που ο αναλυτής δέχεται token διαφορετικό από αυτό που περιμένει, εμφανίζεται ένα στοχευμένο μήνυμα στον χρήστη όπου του αναφέρει τον τύπο του σφάλματος (π.χ. “Expected identifier”, “Expected ‘τότε’” κ.ά.) και την γραμμή στην οποία εντοπίστηκε το σφάλμα και έπειτα ολοκληρώνεται το πρόγραμμα.

Testing

Ακολουθούν screenshot από την εκτέλεση τριών από τα παραδείγματα που έχουμε επισυνάψει. Συγκεκριμένα, ακολουθούν μηνύματα λάθους στις περιπτώσεις που η παρένθεση σε μια αριθμητική παράσταση δεν έχει κλείσει, η απουσία της δεσμευμένης λέξης ‘επανάλαβε’ σε ένα loop και την απουσία της δεσμευμένης λέξης ‘τέλος_συνάρτησης’ όταν ορίζεται σε μια συνάρτηση.

Παρένθεση σε μια αριθμητική παράσταση δεν έχει κλείσει:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_syn_error_6.gr
Syntax Error at line 17: Expected ')'
```

Απουσία της δεσμευμένης λέξης ‘επανάλαβε’ σε ένα loop:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_syn_error_3.gr
Syntax Error at line 20: Expected 'επανάλαβε'
```

Απουσία της δεσμευμένης λέξης ‘τέλος_συνάρτησης’ :

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py test_syn_error_1.gr
Syntax Error at line 14: Expected 'τέλος_συνάρτησης'
```

Ενδιάμεσος Κώδικας

‘Ενα πρόγραμμα στην ενδιάμεση γλώσσα αποτελείται από μία σειρά από αριθμημένες τετράδες χρησιμοποιώντας τον αριθμό της ως ετικέτα. Κάθε τετράδα αποτελείται από έναν τελεστή και τρία τελούμενα , με τον τελεστή να περιγράφει την ενέργεια που γίνεται πάνω σε τρία ή λιγότερα τελούμενα.

Η λίστα που δημιουργείται από τις τετράδες αποτελεί το μέσο επικοινωνίας της φάσης της παραγωγής ενδιάμεσου κώδικα της παραγωγής τελικού κώδικα , ο οποίος παράγεται με βάση αυτή τη λίστα και πληροφόρια που αντλεί από τον πίνακα συμβόλων.

Υλοποίηση

Για να παραχθεί ο ενδιάμεσος κώδικας , υλοποιούμε μία κλάση με το όνομα Quad η οποία έχει για πεδία μία λίστα που θα φιλοξένει τις τετράδες, ένα μετρήτη για τις προσωρινές μεταβλητές και έναν μετρήτη και για τις τετράδες.

Η κλάση αυτή έχει κάποιες βοηθητικές συναρτήσεις με την κάθε μία να ορίζει μία κατάλληλη λειτουργία.

Οι βοηθητικές συναρτήσεις είναι:

• **genQuad()** : Συνάρτηση υπεύθυνη για την δημιουργία της τετράδας και την αποθήκευση της στη λίστα που αποτελεί πεδίο της κλάσης. Αυξάνει τον μετρήτη των τετράδων , ώστε να δείχνει πάντα στην επόμενη τετράδα που πρόκειται να δημιουργηθεί και τον επιστρέφει μειωμένο κατά ένα για να συμφωνεί με τον υπάρχων αριθμό των τετράδων.

• **nextQuad()** : Συνάρτηση που επιστρέφει τον μετρητή των τετράδων που είναι πεδιό της κλάσης , ο οποιός χάρη στην genQuad() δείχνει την ετικέτα της επόμενης τετράδας που θα δημιουργηθεί.

• **nextTemp()** : Συνάρτηση που δημιουργεί μία προσωρινή νέα μεταβλητή. Συγκεκριμένα, αυξάνει τον αντίστοιχο μετρήτη και την επιστρέφει στο κατάλληλο format.

• **emptyList()** : Συνάρτηση που δημιουργεί μία κενή λίστα στην οποία θα τοποθετηθούν ετικέτες τετράδων.

• **makeList()** : Συνάρτηση που δημιουργεί και επιστρέφει μία νέα λίστα η οποία έχει ως μοναδικό στοιχείο της την ετικέτα τετράδας που περνά ως παράμετρος.

• **mergeList()** : Συνάρτηση που συνενώνει τις λίστες που περνάνε ως παράμετροι και δημιουργεί μία νέα λίστα.

• **backpatch()** : Συνάρτηση που ενημερώνει τις τετράδες στη λίστα, ώστε να δείχνουν στην ετικέτα που είναι ως παράμετρος. Συγκεκριμένα, όταν συμπληρωθούν οι τετράδες που σημειώνονται προσωρινά με ασαφή προορισμό τροποποιούνται ώστε το πεδίο του προορισμού (operand3) να δείχνει στη σωστή ετικέτα/γραμμή του κώδικα.

• **printQuads()** : Συνάρτηση που εκτυπώνει όλες τις τετράδες.

• **writeQuadsToFile()** : Συνάρτηση που είναι υπεύθυνη να γράφει τις τετράδες που δημιουργούνται σε ένα αρχείο με κατάληξη .int, υπεύθυνο να έχει όλες τις τετράδες του προγράμματος.

Πριν προχωρήσουμε στον συντακτικό αναλυτή , ώστε να περιγράψουμε το σκεπτικό με τον οποίο παράγεται ο ενδιάμεσος κώδικας πρέπει να σχολιάσουμε μία βοηθητική κλάση με το όνομα struct. Η συγκεκριμένη δομή χρειάζεται στην περίπτωση των λογικών συνθηκών όπου πολλές φορές θα γίνουν λογικά άλματα.

Συγκεκριμένα αυτή η κλάση έχει:

• Μία λίστα **true_list** η οποία αποτελείται από όλες εκείνες τις τετράδες που έχουν μείνει ασυμπλήρωτες διότι ο κανόνας δεν μπορεί να συμπληρωθεί. Οι τετράδες συμπληρώνονται με την ετικέτα της τετράδας στην οποία πρέπει να μεταβεί ο έλεγχος αν η λογική συνθήκη ισχυεί.

• Μία λίστα **false_list** η οποία αποτελείται από όλες εκείνες τις τετράδες που έχουν μείνει ασυμπλήρωτες διότι ο κανόνας δεν μπορεί να συμπληρωθεί. Οι τετράδες συμπληρώνονται με την ετικέτα της τετράδας στην οποία πρέπει να μεταβεί ο έλεγχος αν η λογική συνθήκη δεν ισχυεί.

Προχωράμε στον συντακτικό αναλυτή όπου παίρνοντας κάθε μία κατηγορία ξεχωριστά θα περιγράψουμε το σκεπτικό με το οποίο παράγεται ο αντίστοιχος ενδιάμεσος κώδικας.

Αριθμητικές Παραστάσεις

Σύμφωνα με τη γραμματική της γλώσσας, υποστηρίζονται και αριθμητικές πράξεις. Συγκεκριμένα η γραμματική υποστηρίζει προσθέσεις, αφαιρέσεις , πολλαπλασιασμούς , διαιρέσεις και προτεραιότητα με παρενθέσεις. Η γραμματική που ακολουθείται είναι εξής:

E → T(1) (+ T(2)) *

T → F(1) (* F(2)) *

F → (E)

F → ID

Καθένας από τους τέσσερις κανόνες της γραμματικής επιστρέφει στον κανόνα που τον ενεργοποίησε μία μεταβλητή ως αποτέλεσμα.

Αρχίζουμε από τον πρώτο κανόνα και συγκεκριμένα στον κώδικα πηγαίνουμε στον κανόνα expression() που είναι υπεύθυνος για την τέλεση της πρόσθεσης (και αφαίρεσης). Σύμφωνα με την γραμματική το πρώτο πράγμα που κάνουμε είναι να ενεργοποιήσουμε τον κανόνα term() ο οποίος θα μας επιστρέψει μία τιμή που θα αποθηκεύσουμε στην t1place. ‘Επειτα αφού αναγνωριστεί ο τελεστής από τον λεκτικό αναλυτή τότε ενεργοποιείτε εκ νέου ο κανόνας term() και η τιμή που επιστρέφει αποθηκεύεται στην t2place.

‘Ετσι λοιπόν έχοντας τις τιμές από τους δύο κανόνες θα πρέπει να τους προσθέσουμε και να εκχωρήσουμε το αποτέλεσμα στη μεταβλήτη που θα επιστραφεί ως αποτέλεσμα , την eplace. Μέσα στο while-loop δημιουργείται μία προσωρινή μεταβλητή ώστε να εκχωρηθεί το αποτέλεσμα της πράξης και να καλυφθεί η περίπτωση που θα έχουμε περισσότερες από μία προσθέσεις. Δημιουργείται η νέα τετράδα [oper ,t1place, t2place, w] και αναθέτουμε στην t1place το απότελεσμα που κρατά η w για κάποια πιθανή επόμενη πρόσθεση. Τέλος, έχοντας την t1place ως την μεταβλητή που κρατά το τελικό αποτέλεσμα το αναθέτουμε στην eplace, δηλαδή στην μεταβλητή επιστροφής.

Συνεχίζουμε με τον επόμενο κανόνα αυτόν της factor() , ο οποίος επιστρατεύεται στην περίπτωση του πολλαπλασιασμού και της διαίρεσης. Η λογική είναι ακριβώς όπως αυτής του κανόνα expression(), οπότε δεν υπάρχει και ουσιαστικός λόγος για κάποιον περαιτέρω σχολιασμό.

Επόμενος κάνονας είναι αυτός της factor() που περιγράφεται με τους δύο τελευταίους κανόνες που παραπέμπουν σε δύο διαφορετικές περιπτώσεις. Στην πρώτη περίπτωση που δεν περιγράφεται από τους κανόνες καλούμαστε να γυρνάμε παντα τον αριθμο που θα συνάντα ο λεκτικός αναλυτής.

Στη δεύτερη περίπτωση καλούμαστε να διαχειριστούμε τα δεδομένα που έρχονται από τον κανόνα expression() και αυτό γίνεται όταν θα συναντήσουμε το token ‘(‘. Αρχικοποιούμε μια τοπική μεταβλητή με το όνομα eplace που θα περιεχέι το αποτέλεσμα όλων των πράξεων που έχουν δημιουργηθεί από τον κανόνα expression(). Όταν πλέον ο λεκτικός αναλυτής φτάσει στο token ‘)’ αναθέτουμε τη μεταβλητή eplace που περιέχει το συνολικό αποτέλεσμα στη μεταβλήτη επιστροφής , την fplace.

Τρίτη και τελευταία περίπτωση αυτού του κανόνα είναι όταν θα συναντήσουμε κάποιο token που να ανήκει στην οικογένεια id. Βλέποντας τον κανόνα που εκφράζει την περίπτωση αυτή, κατανοούμε πως πρόκειται για κάτι το πολύ απλό αφού το μόνο που έχουμε να κάνουμε είναι να γυρνάμε αυτούσιο το token μέσω της μεταβλητής fplace.

Σχόλιο: Ο αντίστοιχος κώδικας που υλοποιεί την 3η περίπτωση φαίνεται ξεκάθαρα πιο περίπλοκος από ότι σχολιάζεται παραπάνω. Στο συγκεκριμένο κομμάτι υλοποιούνται σημεία της σημασιολογικής ανάλυσης αλλά και παραγωγής ενδιάμεσου κωδικά στην περίπτωση συναρτήσεων. Τα συγκεκριμένα σημεία θα αποτελέσουν αντικείμενα συζήτησης πιο κάτω στην αναφορά συνεπώς θα υπάρξει στιγμή που θα επανέλθουμε στον κανόνα αυτόν.

Λογικές Συνθήκες

Προχωράμε στις λογικές συνθήκες που η υλοποίηση τους , όπως και στις αριθμητικές παραστάσεις, βασίζεται σε έναν αριθμο κανόνων που θα κληθούμε να υλοποιήσουμε. Συνεπώς η λογική της υλοποίησης των αριθμητικών παραστάσεων ακολουθείται και εδώ.

Αξίζει εδώ να σημειωθει πως τα αποτελέσματα δεν είναι μεταβλητές όπως πριν. Επίσης, μόνο σε έναν κανόνα παράγεται ενδιάμεσος κώδικας, ενώ όλοι οι υπόλοιποι κανόνες διαχειρίζονται πληροφορία και την επεξεργάζονται πριν περάσουν το δικό τους αποτέλεσμα προς τα πάνω.

Τέλος, πριν αρχίσει ουσιαστικά ο σχολιασμός της υλοποίησης των λογικών συνθηκών, αξίζει να αναφέρουμε πως εδώ χρησιμοποιείται η κλάση struct() η οποία περιγράψαμε πιο πάνω.

Οι κανόνες που θα χρησιμοποιηθούν είναι οι εξής:

B → Q (or Q)* B → condition

Q→ R (and R)* Q → boolterm

R → not [B] R→ boolfactor
| [B]
| E rel_op E

Ο πρώτος εξ αυτών είναι ο κανόνας condition() που περιγράφει την συνένωση λογικών συνθηκών με τον τελεστή ‘ή’. Το πρώτο πράγμα που κάνουμε είναι η αρχικοποίηση της δομής struct στην μεταβλητή b_str ώστε μέσω της δομής να έχουμε τις δύο κενές λίστες **b_str.true_list** και **b_str.false_lis**t. Στη συνέχεια ενεργοποιούμε τον κανόνα boolterm() ,ο οποίος και αυτός με την σειρά του θα μας επιστρέψει δυο λίστες ,και τους αναθέτουμε στην μεταβλητή q_str1. Πριν συναντήσουμε τον τελεστή “ή” για να μπούμε μέσα στο while-loop αναθέτουμε τις λίστες της q_str σε αυτές τις b_str ώστε στην περίπτωση που δεν ενεργοποιηθεί ο 2ος κανόνας της boolterm να γυρνάμε ανέπαφες τις λίστες που μας παρέδωσε ο 1ος κανόνας boolterm().

Εντός του while-loop η πρώτη ενέργεια είναι να συμπληρώσουμε το label , με τη βοήθεια της backpatch , των τετράδων που ανήκουν στη λίστα **b_str.false_list** με το label της επόμενης τετράδας . Στη συνέχεια καλούμε τον δεύτερο κανόνα της boolterm() δίνοντας μας με τη σειρά της άλλες δυο λίστες. Εφόσον όλες οι τετράδες ,για την περίπτωση που η λογική συνθήκη αποτιμηθεί ως αληθής , κάνουν άλμα στο ίδιο σημείο, συνενώνουμε την **b_str.true_list** με την **q_str2.true_list**. Τέλος, αφού η **b_str.false_list** έγινε backpatch() πρακτικά δεν υπάρχει με αποτέλεσμα το άλμα που θα θέλουμε να κάνουμε όταν η συνθήκη δεν ισχύει να ανήκει στη λίστα **q_str2.false_list**. Συνεπώς αναθέτουμε αυτή τη λίστα στην **b_str.false_list**, και γυρνάμε την μεταβλητή b_str που έχει σωστά συμπληρωμένες τις δύο λίστες στη δομή της.

Οδηγούμαστε στον δεύτερο κανόνα αυτόν της boolterm() , ο οποίος ακολουθεί ακριβώς την ίδια λογική με αυτή του κανόνα condition(). Το μόνο που αξίζει να σχολιάσουμε εδώ είναι πως εντός του while-loop η λίστα που γίνεται backpatch() είναι η αληθής λίστα της q_str (1ος κανόνας του boolfactor()), ενώ η λίστα που μαζεύει τις τετράδες που μας οδηγούν εκτός της λογικής συνθήκης λόγω ότι δεν ισχύει, είναι η ψευδής λίστα της q_str . Αυτό συμβαίνει διότι όταν μια συνθήκη που βρίσκεται αριστερά ενός and αποτυγχάνει τότε ο έλεγχος μεταβαίνει έξω από την συνθήκη χωρίς περισσότεροι έλεγχοι.

Προχωράμε στον τρίτο και τελευταίο κανόνα την boolfactor(). Ο συγκεκριμένος κάνονας αποτελείται από τρεις υπο-κανόνες τους οποίους θα σχολιάσουμε.

Επιλέγουμε να σχολιάσουμε τον τρίτο και τελευταίο υπο-κανόνα:

E rel_op E

ο οποίος είναι και ο μόνος κανόνας ο οποίος δημιουργεί ενδιάμεσο κώδικα. Ουσιαστικά σε αυτόν τον κανόνα εξετάζουμε τη σύγκριση δυο αριθμητικών εκφράσεων και καλούμαστε να παράξουμε την τετραδα που πραγματοποιεί το λογικό άλμα άμα η συνθήκη ισχύει ή όχι. Συνεπώς, αρχικοποιούμε τις μεταβλητές eplace1, real_op και eplace2 με τους αντίστοιχους κανόνες expression() και relational_oper(). Λόγω του ότι δεν προερχόμαστε από άλλον κανόνα θα πρέπει να δημιουργήσουμε δυο λίστες για τα δυο ενδεχόμενα και έπειτα να τα αναθέσουμε στις λίστες της δομής που αρχικοποιήσαμε στην αρχή του κανόνα r_str. Γιαυτό το λόγο χρησιμοποιούμε την βοηθητική συνάρτηση makeList() η οποία έχει ως παράμετρο το nextQuad() και θα κληθεί πριν την genQuad(). Ακολουθεί η δημιουργία της τετράδας [real_op, eplace1, eplace2, “-”] , ενώ στις επόμενες δυο γραμμές ακολουθείται η ίδια διαδικασία για την τετραδα που θα απευθυνθούμε όταν η συνθήκη δεν ισχύει. Συνεπώς δημιουργείται και η τετραδα [“jump”, “_”,”_”,”_”].

Συνεχίζουμε με τον δεύτερο κανόνα:

[B]

οπου δεν δημιουργείται ενδιάμεσος κώδικας και οι τετράδες που έρχονται από τον κανόνα condition() θα μεταφερθούν στην boolfactor(). Συνεπώς, αναθέτουμε στις λίστες τις r_str τις λίστες που θα έχουμε από την δομή - μεταβλητή b_str.

Τέλος έχουμε τον τρίτο κανόνα :

not [B]

οπου η διάφορα είναι όταν η condition() επιστρέφει αληθές, τότε το boolfactor() πρέπει να επιστρέφει ψευδές και όταν η condition() επιστρέφει ψευδές, τότε το boolfactor() πρέπει να επιστρέφει αληθές. Συνεπώς αναθέτουμε στην r_str.true_list την b_str.false_list και στην r_str.false_list την b_str.true_list.

Αρχή και τέλος ενότητας

Πρώτος κανόνας είναι ο programblock όπου όταν συναντάμε την δεσμευμένη λέξη “αρχή_προγράμματος” δημιουργούμε με την κλήση της συνάρτησης genQuad() την τετράδα : [“begin_block” , όνομα_προγράμματος, “-” ,”-”]. Αντίστοιχα όταν συναντάμε την δεσμευμένη λέξη “τέλος_προγράμματος” δημιουργούνται οι τετράδες : [“halt” , “-” , “-” ,”-”] , η οποία τερματίζει το πρόγραμμα και επιστρέφεται ο χώρος στο λειτουργικό σύστημα, και η τετράδα [“end_block” , όνομα_προγράμματος, “-” ,”-”]. Να σημειωθεί πως το όνομα τους προγράμματος ανατίθεται στην global μεταβλητή prog_name , ώστε έτσι ο κάθε κανόνας να έχει άμεση πρόσβαση σε αυτή.

Συνεχίζουμε αρκετά πιο κάτω στον συντακτικό αναλυτή και συγκεκριμένα στον κανόνα funcblock που και εδώ δημιουργούνται οι τετράδες που σηματοδοτούν την αρχή και το τέλος μίας συνάρτησης , οπότε έδω θα έχουμε τις τετράδες : [“begin_block” , όνομα_συνάρτησης, “-” ,”-”] και [“end_block” , όνομα_συνάρτησης, “-” ,”-”]. Το όνομα της συνάρτησης γίνεται προσβάσιμο από τον κανόνα func οπου ορίζεται ως global μεταβλητή. Εντός της funcblock και συγκεκριμένα πριν κληθεί ο κάνονας subprograms κρατάμε σε μια τοπική μεταβλητή current_name το όνομα της μεταβλητής. Ο λόγος που γίνεται αυτό είναι προκειμένου να μπορούμε στην περίπτωση των φωλιασμένων κλήσεων να κρατείται σωστά το όνομα κάθε συνάρτησης.

Τελευταίος κανόνας που θα παράξει τέτοιους είδους τετράδες είναι η procblock(). Δεν αξίζει να γίνει κάποιος περαιτέρω σχολιασμός καθώς οι τετράδες begin_block και end_block δημιουργούνται με τον ίδιο τρόπο με αυτόν του κανόνα funcblock.

Είσοδος και έξοδος δεδομένων

Εδώ καλούμαστε να παράξουμε τετράδες που να σηματοδοτούν 2 εντολές εισόδου και εξόδου την “γράψε” και “διάβασε”.

Για την εντολή “διάβασε” πηγαίνουμε στον κανόνα input_stat και καλούμε εντός του if-statement την κλήση της συνάρτησης genQuad() που θα παράξει την τετραδα: [“out”,token.recongnized_string,”_”,”_”] με το δεύτερο πεδίο να είναι η μεταβλητή την οποία το πρόγραμμα θα διαβάσει.

Για την εντολή “γράψε” θα πάμε στον αντιστοιχο κανόνα print_stat που πριν παραχθεί η τετραδα αρχικοποιούμε μια μεταβλητή με όνομα x ώστε να κράτα το όνομα της μεταβλητής που θα μας δώσει ο κάνονας expression (ουσιαστικά το όνομα θα προέλθει από τον κανόνα factor) και μετέπειτα παράγεται η τετραδα: [“in”,x”_”,”_”]

Κανόνας while_stat

Στον συγκεκριμένο κανόνα θα πρέπει να παράξουμε τις κατάλληλες τετράδες , όταν συναντήσουμε στο πρόγραμμα έναν βρόγχο τύπου while-loop.

Εντός του κώδικα αρχίζουμε με την αρχικοποίηση της τοπικής μεταβλητής condQuad που θα κρατά το label της τετράδας στην οποία θέλουμε να επανέλθουμε για να επανεξετάσουμε τη λογική συνθήκη. Στην επόμενη γραμμή καλούμε τον κανόνα condition η οποία θα μας επιστρέψει δυο λίστες τις οποίες θα κρατήσει η μεταβλητή cond. Οι ασυμπλήρωτες τετράδες της λίστας cond.true_list πρέπει να συμπληρωθούν με την ετικέτα της πρώτης τετράδας της sequence , ώστε να μεταβούν άμα η συνθήκη ισχύει. Γι αυτό καλούμε την backpatch με παράμετρο την nextQuad(). Μετά την κλήση του sequence πρέπει να δημιουργηθεί μια τετραδα η οποία θα κάνει το άλμα στην αρχή της condition , δηλαδή στο condQuad , γι αυτό πριν το τελευταίο backpatch που γίνεται στον κανόνα τοποθετούμε την τετραδα [“jump”,”_”,”_”,condQuad]. Τέλος, οι τετράδες της cond.false_list συμπληρώνονται με την ετικέτα της πρώτης τετράδας μετά τη δομή του βρόχου, και αυτό γίνεται με backpatch(cond.false_list, nextQuad()).

Κανόνας do_stat

Αρχικά, αποθηκεύουμε τη θέση της πρώτης τετράδας του σώματος της επανάληψης στη μεταβλητή seqQuad. Εκτελούμε το σώμα μέσω της sequence(). Στη συνέχεια, όταν αναγνωριστεί η δεσμευμένη λέξη"μέχρι", καλούμε τον κανόνα condition, ο οποίος επιστρέφει τις λίστες true_list και false_list.

Οι τετράδες της cond.true_list συμπληρώνονται με την επόμενη διαθέσιμη ετικέτα (nextQuad()), δηλαδή το σημείο εξόδου από τον βρόχο. Οι τετράδες της cond.false_list συμπληρώνονται με την ετικέτα της αρχής του σώματος (seqQuad), ώστε να επαναληφθεί αν η συνθήκη δεν ισχύει.

Κανόνας if_stat

Παρόμοια και με τον κανόνα while_stat και εδώ θα πρέπει να διαχειριστούμε σωστά τις δυο λίστες που επιστρέφει η condition. Συνεπώς οι τετράδες που ανήκουν στην λίστα cond.true_list θα πρέπει να συμπληρωθούν με το label της πρώτης τετράδας που θα παράξει η sequence. Γιαυτό μετά την κλήση του κανόνα condition γίνεται η κλήση της βοηθητικής συνάρτησης backpatch με παράμετρο την nextQuad().

Στη συνέχεια μεταφερόμαστε εντός του if_statement οπου η πρώτη ενέργεια μας θα είναι μετά την κλήση του κανόνα sequence είναι να συμπληρώσουμε τις ετικέτες των τετράδων που βρίσκονται στην λίστα cond.false_list με το label της πρώτης τετράδας που θα παράξει ο κανόνας elsepart (ουσιαστικά sequence). Αυτό

προφανώς θα γίνει πάλι με τη κλήση της βοηθητικής συνάρτησης backpatch με παράμετρο την nextQuad().

Πριν όμως την εκτέλεση της backpatch θα πρέπει να τοποθετηθούν οι κατάλληλες εντολές ώστε να παραχθούν τετράδες που θα μας οδηγήσουν έξω από το if , με αποτέλεσμα να παρακάμψουμε τις τετράδες που πρόκειται να δημιουργήσει ο κάνονας elsepart. Αυτό που κάνουμε είναι να δημιουργήσουμε μια τετραδα άλματος με άγνωστο το που θα γίνει το άλμα καθώς η αντίστοιχη τετραδα δεν έχει καν δημιουργηθεί. Γιαυτό χρειαζόμαστε την βοηθητική συνάρτηση makeList(), η οποία θα μας επιστρέψει μια λίστα με τη θέση της τετράδας αυτής ώστε να μπορέσουμε να την συμπληρώσουμε αργότερα.

Τέλος, μετά την παραγωγή του elsepart, καλούμε backpatch για τη λίστα που δημιουργήθηκε με makeList(), ώστε να συμπληρωθεί με την πρώτη διαθέσιμη τετράδα μετά το τέλος του if.

Κανόνας for_stat

Ο κανόνας for_stat είναι υπεύθυνος για τη διαχείριση της επαναληπτικής δομής <για .. έως … βήμα … επανάλαβε … για τέλος>, δηλαδή εκτελεί ένα for loop για μια συγκεκριμένη συνθήκη.

Για να παράγουμε τον σωστό ενδιάμεσο κώδικα, αρχικά όταν αναγνωρίσουμε την δεσμευμένη λέξη «για» στην συντακτική ανάλυση, αποθηκεύουμε το identifier της μεταβλητής που θα συναντήσουμε σε μια μεταβλητή με όνομα loop_var. Αφού η συντακτική ανάλυση αναγνωρίσει και την μεταβλητή / αριθμό από την οποία ξεκινάει το for loop, μέσω της συνάρτησης expression(), την αποθηκεύουμε σε μια μεταβλητή start_expr και παράγουμε την πρώτη γραμμή ενδιάμεσου κώδικα [“:=”, start_expr, “_”, loop_var]. Στην συνέχεια αποθηκεύουμε τη θέση της επόμενης τετράδας (με την χρήση της nextQuad()), η οποία αντιστοιχεί στο σημείο όπου θα επιστρέφουμε κάθε φορά για να ελέγξουμε τη συνθήκη της επανάληψης . Αυτή η θέση αποθηκεύεται στη μεταβλητή startQuad.

Αφού αναγνωριστεί η δεσμευμένη λέξη «έως», τότε αποθηκεύουμε σε μία μεταβλητή end_expr, την τελική τιμή του for, χρησιμοποιώντας τον κανόνα expression. Στη συνέχεια αναλύεται και το βήμα της επανάληψης με χρήση του κανόνα step. Ο κανόνας αυτός μπορεί να επιστρέψει είτε θετικό είτε αρνητικό αποτέλεσμα. Με βάση το πρόσημο του βήματος, καθορίζεται η συνθήκη ελέγχου. Αν το βήμα είναι θετικό, η συνθήκη είναι loop_var >= end_expr. Αν το βήμα είναι αρνητικό, η συνθήκη γίνεται loop_var <= end_expr. Παράγεται λοιπόν η κατάλληλη τετράδα για τη λογική συνθήκη και καταγράφεται η θέση της με την μεταβλητή conditionQuad, ώστε να συμπληρωθεί πλήρως αργότερα με το backpatch. Στη συνέχεια μέσα στο σώμα της for_stat, υπάρχει ο κανόνας sequence, ο οποίος είναι υπεύθυνος για την ανάλυση των εντολών του σώματος του βρόχου. Οι τετράδες αυτών των εντολών θα εκτελούνται όσο η συνθήκη ισχύει.
Μετά την εκτέλεση του κανόνα sequence, δημιουργείται μια νέα προσωρινή μεταβλητή και παράγουμε δύο τετράδες. Οι τετράδες αυτές είναι υπεύθυνες για τον υπολογισμό και την ανάθεση της νέας μεταβλητής loop_var, αφού εφαρμόστηκε πάνω στην προηγούμενη το βήμα. Έπειτα παράγεται τετράδα άλματος στο startQuad (“jump”, “_”, “_”, startQuad).

Τέλος, η τετράδα που περιέχει τη λογική συνθήκη πρέπει να γνωρίζει πού θα μεταβεί όταν δεν ισχύει η συνθήκη. Αυτό επιτυγχάνεται με την κλήση της backpatch([conditionQuad], nextQuad()), ώστε να συμπληρωθεί το πεδίο προορισμού με τη θέση της πρώτης τετράδας μετά τη λήξη του βρόχου.

Συναρτήσεις και Διαδικασίες

Πιο πάνω εξηγήθηκε πως παράγονται οι τετράδες των συναρτήσεων και των διαδικασιών τη στιγμή που ορίζονται. Σε αυτή τη φάση θα αναλυθεί το πως παράγονται οι τετράδες τη στιγμή που στο κύριο πρόγραμμα κάποια συνάρτηση ή διαδικασία καλείται.

Αρχικά από τη στιγμή που θα κληθεί μια συνάρτηση ή μια διαδικασία οι πρώτες τετράδες που θα παραχθούν θα αφορούν της παραμέτρους και έπειτα την κλήση της συνάρτησης ή διαδικασίας.

Συγκεκριμένα , στην περίπτωση της συνάρτησης θα έχουμε πρώτα τα ορίσματα με τα οποία καλείται η συνάρτηση , ενώ σαν τρίτη παράμετρος θα εμφανίζεται μια προσωρινή μεταβλητή που θα κρατά το αποτέλεσμα που μας επιστρέφει η συνάρτηση.
Στην περίπτωση της διαδικασίας θα έχουμε μόνο τα ορίσματα που τις δίνουμε.

Η σειρά των κανόνων που διασχίζουμε όταν καλούμε μια συνάρτηση είναι η εξής: **assignment_stat → expression → term → factor → idtail → actualpars → actualparlist→ actualparitem**.

Όταν φτάνουμε στην factor που με τη σειρά της θα καλέσει τον κανόνα idtail επιλέγουμε να φορτώσουμε με το όνομα της συνάρτησης στους επόμενους κανόνες που θα ακολουθήσουν με αποτέλεσμα όταν φτάσουμε στον τελευταίο κανόνα που είναι ο actualparitem να φτιάχνουμε σωστά τις τετράδες. Ο λόγος που γίνεται αυτό θα γίνει πιο ξεκάθαρος στην περιγραφή του τελικού κώδικα.

Συνεπώς στον κανόνα actualparitem θα παράξουμε την τετραδα [“par”,token.recongnized_string,”ref”, caller_name] όταν πρόκειται για μια παράμετρος που περνιέται με αναφορά , ενώ αντίστοιχα για μια παράμετρο που περνιέται με τιμή θα έχουμε την τετραδα : [“par”,token.recongnized_string,”cv”, caller_name]

Στην περίπτωση της συνάρτησης θα πρέπει να παράξουμε άλλη μια τετραδα που θα έχει την παράμετρο που θα κρατά το αποτέλεσμα της συνάρτησης. Αυτό γίνεται στον κανόνα factor με τη βοήθεια ενός flag με το όνομα check_func. Άμα η κλήση του κανόνα idtail ενεργοποιήσει το flag (το κάνει ίσο με 1), τότε αυτό σηματοδοτεί πως πρόκειται για συνάρτηση. Οπότε εντός του if_statement που ελέγχουμε την τιμή του flag δημιουργούμε μια προσωρινή μεταβλητή και με την

βοηθητική συνάρτηση παράγουμε την τετραδα [“par”,w,”ret”,fplace] με fplace το όνομα της συνάρτησης και w την προσωρινή μεταβλητή.
Τέλος, αμέσως μετά παράγουμε την τετραδα [“call”,fplace,”_”,”_”] που δείχνει την κλήση της συνάρτησης.

Τέλος, μας απομένει η περίπτωση της διαδικασίας που η διαδρομή των κανόνων είναι: **assignment_stat → call_stat → idtail → actualpars → actualparlist→ actualparitem**.

Δεν υπάρχει λόγος να γίνει ιδιαίτερη ανάλυση καθώς οι τετράδες που παράγονται με τον ίδιο τρόπο όπως και στην περίπτωση της συνάρτησης στον κανόνα actualparitem.
Όσον αφορά την τετραδα που σηματοδοτεί την κλήση της διαδικασίας, παράγεται απευθείας στον κανόνα actualparitem οπότε θα έχουμε [“call”,pname,”_”,”_”] με pname το όνομα της διαδικασίας.

Testing

Ακολουθούν screenshot από την εκτέλεση 2 προγραμμάτων που έχουμε επισυνάψει και έχουν αναρτηθεί στο teams. Ακολουθεί ο ενδιάμεσος κώδικας που παράχθηκε, καθώς και το αντίστοιχο output από το πρόγραμμα write_to_c.py.

test_teams2:

```
1 : begin_block , τεστ2_teams , _ , _
2 : := , 1 , _ , α
3 : < , α , 10 , 5
4 : jump , _ , _ , 8
5 : + , α , 1 , T@1
6 : := , T@1 , _ , α
7 : jump , _ , _ , 3
8 : out , α , _ , _
9 : := , α , _ , β
10 : + , β , 1 , T@2
11 : := , T@2 , _ , β
12 : - , β , 2 , T@3
13 : := , T@3 , _ , β
14 : = , β , 0 , 16
15 : jump , _ , _ , 10
16 : out , β , _ , _
17 : - , 0 , 1 , T@4
18 : <= , β , T@4 , 20
19 : jump , _ , _ , 23
20 : - , 0 , 1 , T@5
21 : := , T@5 , _ , α
22 : jump , _ , _ , 25
23 : - , 0 , 2 , T@6
24 : := , T@6 , _ , α
25 : out , α , _ , _
26 : < , α , β , 28
27 : jump , _ , _ , 30
28 : := , 1 , _ , α
29 : jump , _ , _ , 31
30 : := , 2 , _ , α
31 : out , α , _ , _
32 : halt , _ , _ , _
33 : end_block , τεστ2_teams , _ , _
```

```c
#include <stdio.h>

int main()
{
int α ;
int T$1 ;
int β ;
int T$2 ;
int T$3 ;
int T$4 ;
int T$5 ;
int T$6 ;

L1:
L2: α=1;
L3: if (α < 10) goto L5;
L4: goto L8;
L5: T$1=α+1;
L6: α=T$1;
L7: goto L3;
L8: printf("%d\n",α);
L9: β=α;
L10: T$2=β+1;
L11: β=T$2;
L12: T$3=β-2;
L13: β=T$3;
L14: if (β == 0) goto L16;
L15: goto L10;
L16: printf("%d\n",β);
L17: T$4=0-1;
L18: if (β <= T$4) goto L20;
L19: goto L23;
L20: T$5=0-1;
L21: α=T$5;
L22: goto L25;
L23: T$6=0-2;
L24: α=T$6;
L25: printf("%d\n",α);
L26: if (α < β) goto L28;
L27: goto L30;
L28: α=1;
L29: goto L31;
L30: α=2;
L31: printf("%d\n",α);
L32:
L33:
}
```

test_teams3:

```
1 : begin_block , τεστ_teams3 , _ , _
2 : in , income , _ , _
3 : <= , income , 1000 , 5
4 : jump , _ , _ , 7
5 : := , 0 , _ , tax
6 : jump , _ , _ , 31
7 : <= , income , 3000 , 9
8 : jump , _ , _ , 13
9 : - , income , 1000 , T@1
10 : / , T@1 , 10 , T@2
11 : := , T@2 , _ , tax
12 : jump , _ , _ , 31
13 : <= , income , 7000 , 15
14 : jump , _ , _ , 22
15 : - , 3000 , 1000 , T@3
16 : / , T@3 , 10 , T@4
17 : - , income , 3000 , T@5
18 : / , T@5 , 5 , T@6
19 : + , T@4 , T@6 , T@7
20 : := , T@7 , _ , tax
21 : jump , _ , _ , 31
22 : - , 3000 , 1000 , T@8
23 : / , T@8 , 10 , T@9
24 : - , 7000 , 3000 , T@10
25 : / , T@10 , 5 , T@11
26 : + , T@9 , T@11 , T@12
27 : - , income , 7000 , T@13
28 : / , T@13 , 2 , T@14
29 : + , T@12 , T@14 , T@15
30 : := , T@15 , _ , tax
31 : out , tax , _ , _
32 : halt , _ , _ , _
33 : end_block , τεστ_teams3 , _ , _
```

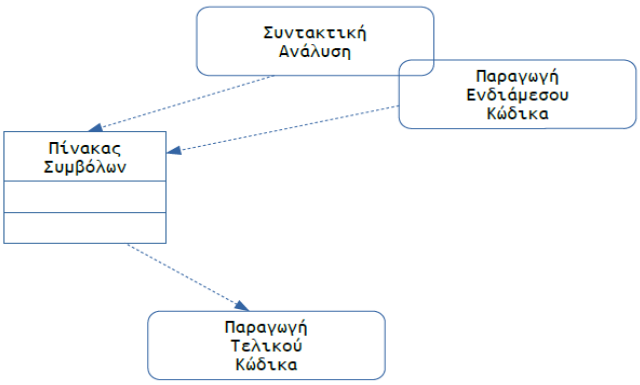
```c
#include <stdio.h>

int main()
{
int income ;
int tax ;
int T$1 ;
int T$2 ;
int T$3 ;
int T$4 ;
int T$5 ;
int T$6 ;
int T$7 ;
int T$8 ;
int T$9 ;
int T$10 ;
int T$11 ;
int T$12 ;
int T$13 ;
int T$14 ;
int T$15 ;

L1:
L2: scanf("%d",&income);
L3: if (income <= 1000) goto L5;
L4: goto L7;
L5: tax=0;
L6: goto L31;
L7: if (income <= 3000) goto L9;
L8: goto L13;
L9: T$1=income-1000;
L10: T$2=T$1/10;
L11: tax=T$2;
L12: goto L31;
L13: if (income <= 7000) goto L15;
L14: goto L22;
L15: T$3=3000-1000;
L16: T$4=T$3/10;
L17: T$5=income-3000;
L18: T$6=T$5/5;
L19: T$7=T$4+T$6;
L20: tax=T$7;
L21: goto L31;
L22: T$8=3000-1000;
L23: T$9=T$8/10;
L24: T$10=7000-3000;
L25: T$11=T$10/5;
L26: T$12=T$9+T$11;
L27: T$13=income-7000;
L28: T$14=T$13/2;
L29: T$15=T$12+T$14;
L30: tax=T$15;
L31: printf("%d\n",tax);
L32:
L33:
}
```

Πίνακας συμβόλων

Ο πίνακας συμβόλων αποτελεί τη βασική δομή μέσω της οποίας οργανώνεται και παρακολουθείται κάθε αναγνωριστικό που εμφανίζεται στο πρόγραμμα προς μετάφραση. Χρησιμοποιείται κατά τη φάση της συντακτικής ανάλυσης και της παραγωγής ενδιάμεσου κώδικα. Τα αναγνωριστικά που αποθηκεύονται στον πίνακα συμβόλων είναι τα εξής:

- Μεταβλητές
- Παράμετροι
- Διαδικασίες / Συναρτήσεις
- Τυπικές Παράμετροι
- Προσωρινές Μεταβλητές

Ο σκοπός του πίνακα συμβόλων είναι διπλός. Αφενός, να αποθηκεύει πληροφορίες για κάθε συμβολικό όνομα του προγράμματος. Αφετέρου, να επιτρέπει την ανάκτησή τους όταν χρειαστεί. Έτσι ο πίνακας εξασφαλίζει την ορθή σύνδεση δηλώσεων και χρήσεων μέσα στο πρόγραμμα, απαραίτητη διαδικασία για την σημασιολογική ανάλυση αλλά και την παραγωγή τελικού κώδικα.



Ο πίνακας συμβόλων που υλοποιήσαμε για την γλώσσα της Greek++ οργανώνεται γύρω από την ιδέα ότι κάθε αναγνωριστικό αντιστοιχεί σε μία συγκεκριμένη κατηγορία συμβόλου η οποία καθορίζει ποια πληροφορία είναι σχετική και πρέπει να αποθηκευτεί. Για τον λόγο αυτό, κάθε τύπος εγγραφής υλοποιείται ως ξεχωριστή δομή, με τα κατάλληλα πεδία που απαιτούνται για τη σωστή επεξεργασία του στο πλαίσιο του μεταφραστή. Ακολουθεί αναλυτική παρουσίαση της κάθε εγγραφής:

1) Μεταβλητή

Αντιπροσωπεύει μια απλή μεταβλητή που δηλώνεται με τη λέξη-κλειδί της γλώσσας ¨δήλωση¨.

Πεδία :

- Name
- Datatype
- Offset

2) Παράμετρος

Αντιπροσωπεύει μια παράμετρο που δηλώνεται στη λίστα παραμέτρων ενός υποπρογράμματος

Πεδία:

- Name
- Mode (by value ή by reference)
- Offset

3) Συνάρτηση

Υποπρόγραμμα που επιστρέφει τιμή

Πεδία:

- Name
- Datatype
- startingQuad
- frameLength
- formalParameters

4) Διαδικασία

Υποπρόγραμμα που δεν επιστρέφει τιμή

Πεδία:

- Name
- startingQuad
- frameLength
- formalParameters

5) Τυπική Παράμετρος

Δεν αποθηκεύεται απευθείας στον πίνακα συμβόλων, αλλά συνδέεται με τις Συναρτήσεις και Διαδικασίες ως βοηθητική πληροφορία

Πεδία:

- Datatype
- Mode

6) Προσωρινή Μεταβλητή

Μεταβλητές που παράγονται εσωτερικά από τον μεταφραστή

Πεδία:

- Name
- Offset

Κανόνες Εμβέλειας

Η σωστή διαχείριση της εμβέλειας των αναγνωριστικών αποτελεί βασική απαίτηση ενός μεταφραστή. Η εμβέλεια ενός αναγνωριστικού ορίζεται ως η περιοχή το υπρογράμματος στην οποία είναι επιτρεπτή η χρήση του. Για παράδειγμα , μια μεταβλητή που δηλώνεται εντός μιας συνάρτησης είναι προσβάσιμη μόνο μέσα σε αυτήν και όχι σε όλο το πρόγραμμα.

Για την υποστήριξη των κανόνων εμβέλειας, ο πίνακας συμβόλων οργανώνεται με τη μορφή στοίβας (scopes), όπου κάθε στοιχείο της στοίβας αντιστοιχεί σε ένα επίπεδο εμφώλευσης. Κάθε φορά που εισερχόμαστε σε ένα νέο υποπρόγραμμα ή μπλοκ δηλώσεων, δημιουργείται ένα νέο scope και τοποθετείται στο πάνω μέρος της στοίβας. Όταν ολοκληρωθεί το μπλοκ, το scope αφαιρείται από τη στοίβα και έτσι απομακρύνονται και όλα τα σύμβολα που δεν πρέπει να είναι πλέον ορατά. ΜΕ αυτό τον τρόπο επιτρέπουμε και τον χειρισμό τοπικών και καθολικών μεταβλητών και δήλωσης συμβόλων με ίδιο όνομα σε διαφορετικά scopes.

Αναζήτηση Αναγνωριστικών

Όταν αναζητείται ένα αναγνωριστικό, η αναζήτηση πραγματοποιείται από το πιο εσωτερικό scope προς τα έξω, δηλαδή από τη κορυφή προς την βάση της στοίβας. Αυτό διασφαλίζει ότι το αναγνωριστικό που βρίσκεται πιο κοντά στον τόπο χρήσης του θα βρεθεί πρώτο, σε περίπτωση που το ίδιο όνομα έχει χρησιμοποιηθεί και σε εξωτερικό scope.

Λειτουργίες Πίνακα Συμβόλων

Ο πίνακας συμβόλων υποστηρίζει ένα σύνολο βασικών λειτουργιών που επιτρέπουν την διαχείριση των scopes, προσθήκη και αναζήτηση συμβόλων και ενημέρωση πληροφοριών. Παρακάτω περιγράφονται οι σημαντικότερες λειτουργίες του πίνακα, όπως υλοποιήθηκαν στην περίπτωσή μας:
1) Προσθήκη και Αφαίρεση Scopes

Add_scope() : η συνάρτηση αυτή προσθέτει ένα νέο επίπεδο στον πίνακα συμβόλων. Κάθε επίπεδο είναι ανεξάρτητο και διατηρεί τη δική του λίστα εγγραφών. Κατά την εισαγωγή, το offset του κάθε επιπέδου αρχικοποιείται στη σταθερή τιμή 12, για να αφήνει χώρο για τις σταθερές θέσεις του εγγραφήματος δραστηριοποίησης (το εγγράφημα δραστηριοποίησης θα εξηγηθεί στο κεφάλαιο παραγωγής του τελικού κώδικα).

Remove_scope(): Όταν ένα scope ολοκληρώνεται, η συνάρτηση το αφαιρεί από τη στοίβα και επιστρέφει το offset που είχε. Αν το scope σχετίζεται με υποπρόγραμμα

(συνάρτηση ή διαδικασία), ενημερώνεται αυτόματα το frameLength της αντίστοιχης εγγραφής.

2) Εισαγωγή και Ενημέρωση Εγγραφών

Add_entry(entry): Η συνάρτηση προσθέτει μια νέα εγγραφή στο τρέχον scope. Πριν την εισαγωγή, γίνεται έλεγχος για διπλή δήλωση του ίδιου ονόματος στο ίδιο επίπεδο, και σε περίπτωση που ισχύει εμφανίζεται σφάλμα σημασιολογικής ανάλυσης (περισσότερα για την σημασιολογική ανάλυση παρακάτω). Ενημερώνει επίσης κατάλληλα το πεδίο offset των συμβόλων που εισάγει.

Update_entry(name, updates): Αναζητεί την εγγραφή με το όνομά της και ενημερώνει τα πεδία της.

Add_formal_parameter(proc_or_func_name, datatype, mode): Προσθέτει μια τυπική παράμετρο σε υπάρχουσα εγγραφή συνάρτησης ή διαδικασίας, η οποία αποθηκεύεται στην αντίστοιχη λίστα formalParameters.

3) Αναζήτηση Εγγραφών

Find_entry(name) : Αναζητεί μία εγγραφή με βάση το όνομα, ξεκινώντας από το πιο πάνω scope και προχωρά μέχρι τη βάση του πίνακα. Σε περίπτωση που δεν βρεθεί η εγγραφή, εμφανίζεται μήνυμα σημασιολογικού σφάλματος.

Find_entry_level(name): Επιστρέφει το επίπεδο στο οποίο βρίσκεται η εγγραφή με το δοσμένο όνομα.

4) Διαχείριση offset

Gent_next_offset(): επιστρέφει την επόμενη διαθέσιμη θέση μνήμης για το τρέχον scope.

Σημασιολογική ανάλυση

Στο πλαίσιο της σημασιολογικής ανάλυσης μπορούμε να εφαρμόσουμε πρόσθετους ελέγχους για την ορθότητα ενός προγράμματος. Ενώ η συντακτική ανάλυση επαληθεύει ότι το πρόγραμμα είναι γραμμένο με σωστή μορφή, η σημασιολογική ανάλυση κάνει ελέγχους που δεν έχουν περιγραφεί από την γραμματική της γλώσσας.

Έλεγχοι που υλοποιούνται:

1) Έλεγχος Αδήλωτων Συμβόλων

Όταν γίνεται αναφορά σε ένα αναγνωριστικό, η σημασιολογική ανάλυση επαληθεύει ότι το σύμβολο έχει δηλωθεί σε κάποιο από τα ενεργά scopes. Αν δεν βρεθεί, εκτυπώνεται ένα ενημερωτικό μήνυμα σημασιολογικού σφάλματος. Αυτός ο έλεγχος υλοποιήθηκε εντός της συνάρτησης find_entry()

2) Έλεγχος Διπλών Δηλώσεων

Εδώ ελέγχουμε αν ένα αναγνωριστικό έχει ήδη δηλωθεί στο ίδιο scope. Σε περίπτωση δεύτερης δήλωσης στο ίδιο scope, εμφανίζουμε ένα αντίστοιχο μήνυμα σημασιολογικού σφάλματος. Αυτός ο έλεγχος υλοποιήθηκε εντός της συνάρτησης add_entry()

3) Έλεγχος τοποθεσίας return

Εδώ ελέγχουμε αν μια εντολή return (στην περίπτωση της Greek++, η ονομασία μιας συνάρτησης στο αριστερό μέρος μιας ανάθεσης) βρίσκεται μέσα σε μία συνάρτηση. Αν δεν βρίσκεται εντός μιας συνάρτησης, εμφανίζουμε αντίστοιχο μήνυμα σημασιολογικού σφάλματος. Ο έλεγχος γίνεται εντός της συνάρτησης assignment_stat() της συντακτικής ανάλυσης.

4) Έλεγχος ύπαρξης τουλάχιστον μίας return

Ελέγχουμε αν υπάρχει τουλάχιστον μία εντολή return εντός μίας συνάρτησης. Αν δεν υπάρχει εμφανίζουμε αντίστοιχο μήνυμα σημασιολογικού σφάλματος. Ο έλεγχος γίνεται εντός της συνάρτησης funcblock() της συντακτικής ανάλυσης.

5) Έλεγχος σωστής κλήσης υποπρογράμματος

Εδώ ελέγχουμε αν μια συνάρτηση που καλείται πράγματι έχει δηλωθεί ως συνάρτηση και αντίστοιχος έλεγχος γίνεται και για την διαδικασία. Αν δεν γίνεται σωστή κλήση για κάποια από τις δύο περιπτώσεις εμφανίζουμε αντίστοιχο μήνυμα σημασιολογικού σφάλματος. Ο έλεγχος γίνεται εντός των συναρτήσεων factor() και call_stat() της συντακτικής ανάλυσης

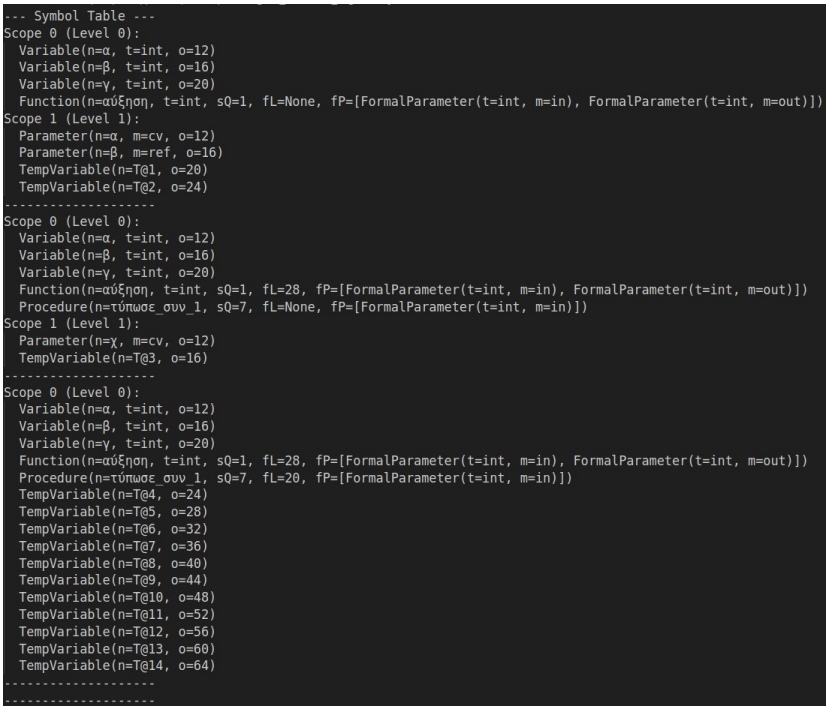
6) Έλεγχος σωστής κλήσης παραμέτρων

Σε αυτό το σημείο ελέγχουμε αν οι παράμετροι που δέχεται έναν υποπρόγραμμα έχουν οριστεί ακριβώς όπως καλούνται. Αν η συνάρτηση ή διαδικασία δεν καλείται με σωστές παραμέτρους τότε εμφανίζουμε αντίστοιχο μήνυμα σημασιολογικού σφάλματος. Ο έλεγχος γίνεται εντός των συναρτήσεων factor() και call_stat() της συντακτικής ανάλυσης.

Όλοι οι παραπάνω έλεγχοι σημασιολογικής ανάλυσης έχουν τεσταριστεί σε .gr προγράμματα που παραδώσαμε.

Testing

Ακολουθούν screenshot από την εκτέλεση ενός πλήρως προγράμματος (paradeigma_ekfonisis_ergasias) που έχουμε επισυνάψει, ενώ ακολουθούν αποτελέσματα από μερικά από τα προγράμματα που αποτυγχάνουν στους έλεγχους της σημασιολογικής ανάλυσης (υπάρχουν περισσότερα τέτοια προγράμματα εντός του αρχείου .zip)



Λάθος πέρασμα παραμέτρου:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py semantic_error4.gr
Semantic Error at line 16: Wrong parameter given for Function 'αύξηση'
```

Λάθος κλήση συνάρτησης:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py semantic_error2-2.gr
Semantic Error at line 24: Entity 'αύξηση' is not a Procedure
```

Απουσία return:

```
dimitris@dimitris-matebook:~/Desktop/μεταφραστες$ python3 greek_5252_5321.py semantic_error5.gr
Semantic Error at line 11: No return call (αύξηση)
```

Παραγωγή Τελικού Κώδικα

Η παραγωγή του τελικού κώδικα είναι το στάδιο της μεταγλώττισης, όπου παράγεται ο κώδικας σε γλώσσα μηχανής. Ο τελικός κώδικας προκύπτει από τον ενδιάμεσο κώδικα με την βοήθεια του πίνακα συμβόλων. Συγκεκριμένα, από κάθε εντολή ενδιάμεσου κώδικα προκύπτει μία σειρά εντολών τελικού κώδικα, η οποία για να παραχθεί ανακτά πληροφορίες από τον πίνακα συμβόλων. Ο τελικός κώδικας μπορεί να λάβει διάφορες μορφές. Στην δικιά μας περίπτωση, θα παραγάγουμε τελικό κώδικα σε συμβολική γλώσσα μηχανής (assembly code) του επεξεργαστή RISC-V. Βέβαια, ο τελικός κώδικας που έχει σχεδιαστεί δεν έχει μεγάλη εξάρτηση από το υλικό και τις ιδιαιτερότητες του RISC-V. Οι εντολές της συμβολικής γλώσσας μηχανής που επιστρατεύσαμε από το διαθέσιμο σύνολο εντολών του RISC-V, υποστηρίζονται από όλους τους επεξεργαστές, με μικρές και μη σημαντικές διαφοροποιήσεις.

Παρακάτω αναφέρονται οι περισσότερες από τις εντολές που θα χρησιμοποιήσουμε:

Εκχώρηση αριθμητικής σταθεράς σε καταχωρητή:

```
li reg,int   # reg = int
   # reg: destination register
   # int: arithmetic integer constant
```

Πράξεις μεταξύ ακεραίων:

```
add reg, reg1, reg2    # reg = reg1 + reg2
sub reg, reg1, reg2    # reg = reg1 - reg2
mul reg, reg1, reg2    # reg = reg1 * reg2
div reg, reg1, reg2    # reg = reg1 / reg2
    # reg: destination register
    # reg1,reg2: registers (operands)
```

Πρόσβαση στη μνήμη:

```
lw reg1,offset(reg2)   # reg1 = [reg2 + offset]
    # reg1: destination register
    # reg2: base register
    # offset: distance from reg2
sw reg1,offset(reg2)   # [reg2 + offset] = reg1
    # reg1: source register
    # reg2: base register
    # offset: distance from reg2
```

Εντολές διακλαδώσεων:

```
b label
    # label: an address

beq reg1,reg2,label    # branch if equal
bne reg1,reg2,label    # branch if not equal
blt reg1,reg2,label    # branch if less than
bgt reg1,reg2,label    # branch if greater than
ble reg1,reg2,label    # branch if less or equal than
bge reg1,reg2,label    # branch if greater or equal than
  # reg1,reg2: the registers to be compared
  # label: the address to jump to

jr reg   # jump [reg]
  # reg: a register
```

Για να κάνουμε ανάγνωση ενός αριθμού χρησιμοποιούμε τις εντολές:

```
li a7,5
ecall
```

Για να εμφανίσουμε έναν αριθμό στην οθόνη πρέπει να τοποθετήσουμε τον ακέραιο που επιθυμούμε στον καταχωρητή a0 και στην συνέχεια να καλέσουμε άλλες δύο εντολές όπως φαίνονται παρακάτω:

```
li a0,44
li a7,1
ecall
```

Για να τερματίσουμε ένα πρόγραμμα καλούμε τις εξής εντολές:

```
li a0,0
li a7,93
ecall
```

Το εγγράφημα δραστηριοποίησης

Κατά την εκτέλεση ενός προγράμματος και κάθε κλήσης υποπρογράμματος δεσμεύεται ένας χώρος στη στοίβα προκειμένου να τοποθετηθούν εκεί δεδομένα, τιμές και διευθύνσεις μεταβλητών. Αυτός ο χώρος που δεσμεύεται ονομάζεται εγγράφημα δραστηριοποίησης και περιλαμβάνει όλα τα απαραίτητα δεδομένα για την ορθή εκτέλεση ενός υποπρογράμματος και την επιστροφή του στο σημείο από το οποίο εκλήθη. Κατά την είσοδο σε ένα υποπρόγραμμα, δημιουργείται νέο εγγράφημα, ενώ με την έξοδο από αυτό, το εγγράφημα αφαιρείται από τη στοίβα.

Στην υλοποίηση μας, κάθε εγγραφή στο εγγράφημα καταλαμβάνει χώρο 4 bytes, αφού χρησιμοποιούμε μόνο ακεραίους, και υπάρχουν κάποιες θέσεις στην αρχή του εγγραφήματος που περιέχουν συγκεκριμένη πληροφορία.

Στα πρώτα 4 bytes του εγγραφήματος, αποθηκεύουμε την διεύθυνση επιστροφής. Είναι η διεύθυνση που πρέπει να μεταβεί το πρόγραμμα μετά την ολοκλήρωση της εκτέλεσης της συνάρτησης ή της διαδικασίας.

Στις θέσεις bytes 4 με 7, αποθηκεύουμε τον σύνδεσμο προσπέλασης. Είναι η διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα του υποπρογράμματος. Μέσα από αυτό το σύνδεσμο το υποπρόγραμμα μπορεί να προσπελάσει δεδομένα που ανήκουν σε προγόνους του.

Στις θέσεις bytes 8 με 11, αποθηκεύουμε την επιστροφή τιμής. Είναι η διεύθυνση της μεταβλητής στην οποία επιθυμούμε να γραφεί το αποτέλεσμα της συνάρτησης. Αν πρόκειται για διαδικασία , η θέση αυτή μένει αχρησιμοποίητη.

Οι θέσεις μετά το byte 12, δεικτοδοτούνται από τον πίνακα συμβόλων, αφού κρατάνε πληροφορίες για όλα τα σύμβολα που αναλύσαμε στην προηγούμενη ενότητα.

Αυτό λοιπόν που περιγράφηκε είναι η δομή του εγγραφήματος. Με την εκκίνηση της εκτέλεσης ενός προγράμματος το λειτουργικό σύστημα του δεσμεύει χώρο ο οποίος θα λειτουργήσει ως στοίβα. Στο σημείο αυτό τοποθετείται ο δείκτης στοίβας sp (stack pointer). Στη συνέχεια αναλαμβάνει ο κώδικας που παρήχθη από τον μεταγλωττιστή. Ο δείκτης στοίβας δείχνει ανά πάσα στιγμή στην αρχή του εγγραφήματος δραστηριοποίησης του υποπρογράμματος που εκείνη τη στιγμή εκτελείται.

Υλοποίηση

Για να παραχθεί ο τελικός κώδικας, χρειάστηκε να υλοποιηθούν κάποιες βοηθητικές συναρτήσεις με την κάθε μια να ορίζει μια κατάλληλη λειτουργία.
Οι βοηθητικές συναρτήσεις είναι:

- **make_final_code():** Συνάρτηση που δημιουργεί ένα αρχείο με κατάληξη .asm και καταγράφει όλες τις assembly εντολές στο αρχείο αυτό.

- **get_risc_instr():** Συνάρτηση που δημιουργεί τις assembly εντολές στο κατάλληλο format. Συγκεκριμένα, παίρνει δυο ορίσματα , op που δείχνει την εκάστοτε assembly εντολή και την terms η οποία αποτελεί μια λίστα στην οποία τοποθετούμε τους εκάστοτε καταχωρητές που θα χρησιμοποιήσουμε.
- **is_local():** Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι τοπική, δηλαδή εάν υπάρχει στο τελευταίο επίπεδο του πίνακα συμβόλων.
- **is_global():** Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι καθολική, δηλαδή εάν υπάρχει στο πρώτο επίπεδο του πίνακα συμβόλων.
- **is_temp_var():** Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι προσωρινή.
- **is_parameter()**: Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι παράμετρος.
- **is_variable()**: Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι μεταβλητή.
- **is_cv_par():** Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι παράμετρος που έχει περαστεί με τιμή.
- **is_ref_par()**: Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα είναι παράμετρος που έχει περαστεί με αναφορά.
- **belongs_ancestor()**: Συνάρτηση που ελέγχει εάν η μεταβλητή που υπάρχει σαν όρισμα βρίσκεται σε κάποια συνάρτηση ή διαδικασία πρόγονο.

Ορίζοντας τις παραπάνω συναρτήσεις μας γίνεται εξαιρετικά πιο εύκολο να ορίσουμε κάποιες επιπλέον βοηθητικές συναρτήσεις οι οποίες μας διευκολύνουν στον σχεδιασμό του τελικού κώδικα.

Αυτές οι συναρτήσεις είναι:

- **gnlvcode()**: Συνάρτηση που μας βοηθά στην προσπέλαση πληροφορίας που βρίσκεται στο εγγράφημα δραστηριοποίησης κάποιου προγόνου ή διαδικασίας.
- **loadvr()**: Συνάρτηση που διαβάζει μια μεταβλητή που είναι αποθηκευμένη στη μνήμη και την μεταφέρει σε έναν καταχωρητή.
- **storerv()**: Συνάρτηση που διαβάζει μια μεταβλητή που είναι αποθηκευμένη σε έναν καταχωρητή και την μεταφέρει στην μνήμη.

Η συνάρτηση gnlvcode()

Στην συνάρτηση αυτή το πρώτο πράγμα που κάνουμε είναι να βρούμε εάν η μεταβλητή βρίσκεται στον πινάκα συμβόλων. Αυτό επιτυγχάνεται με την βοηθητική συνάρτηση find_entry() που ορίσαμε για την υλοποίηση του πινάκα συμβόλων και αποθηκεύουμε το αποτέλεσμα που μας γυρνάει στην τοπική μεταβλητή item. Έπειτα , εφόσον υπάρχει η μεταβλήτη στον πινάκα συμβόλων, βρίσκουμε το επίπεδο στο οποίο βρίσκεται μέσω της συνάρτησης find_entry_level() με το αποτέλεσμα να αποθηκεύεται στην μεταβλητή dest_level. Τελευταία πληροφορία που θα χρειαστεί να αντλήσουμε από τον πίνακα συμβόλων είναι ο τρέχων αριθμός επιπέδων που υπάρχει. Αυτή η πληροφορία θα υπάρχει στη μεταβλητή cur_level μέσω της βοηθητικής συνάρτησης get_nesting_level().

Εφόσον αντλήσαμε όλη την πληροφορία από τον πινάκα συμβόλων πρέπει να συμπεραίνουμε ποσά επίπεδα πάνω στο γενεαλογικό δεντρό της συνάρτησης θα πρέπει να ανεβεί προκείμενου να φτάσει στο εγγράφημα δραστηριοποίησης που έχει την πληροφορία που αναζητά. Έχοντας τον τρέχων αριθμο επίπεδων καθώς και το επίπεδο της μεταβλητής που ψάχνουμε, η διαφορά τους είναι ο αριθμός των επίπεδων που θα χρειαστούμε να ανεβούμε. Αυτός ο αριθμός αποθηκεύεται στη μεταβλητή levels_to_hop.

Το πρώτο βήμα είναι να μεταβούμε στον γονέα της συνάρτησης. Η διεύθυνση του εγγραφήματος δραστηριοποίησης του γονέα βρίσκεται αποθηκευμένη στον σύνδεσμο προσπέλασης της συνάρτησης, δηλαδή στη θέση -4(sp), αφού ο sp δείχνει την αρχή του εγγραφήματος δραστηριοποίησης της συνάρτησης που κάθε στιγμή εκτελείται. Συνεπώς παράγουμε την assembly εντολή:

lw, t0 , -4 , sp

Εφόσον έχουμε συμπεράνει πως πρέπει να ανεβούμε ν επίπεδα, έχοντας μεταβεί στον γονέα πλέον μας απομένουν ν-1 επίπεδα. Γνωρίζοντας πλέον ακριβώς τον αριθμο που θα χρειαστεί να ανεβούμε , έχουμε ένα for-loop στο οποίο επαναλαμβάνουμε τον ίδιο τρόπο όπως και προηγουμένως, δηλαδή θα διαβάσουμε τον σύνδεσμο προσπέλασης από το εγγράφημα δραστηριοποίησης στο οποιο δείχνει ο t0. Ετσι για κάθε επίπεδο που πρέπει να ανεβεί παράγεται η εντολή:

lw, t0 , -4 , t0

Αφού μεταβούμε τόσα επίπεδα όσα είναι απαραίτητα , ο καταχωρητής t0 θα δείχνει στην αρχή του εγγραφήματος δραστηριοποίησης το οποίο ο πίνακας συμβόλων υπέδειξε. Άρα αυτό που απομένει είναι να κατεβεί ο t0 κατά offset θέσεις ώστε να δείξει στη μνήμη την πληροφορία που αναζητούμε. Συνεπώς παράγουμε την assembly εντολή:

addi , t0 , t0 , str(-item.offset)

item.offset: Θέση του στοιχείου που αναζητούμε στο εγγράφημα δραστηριοποίησης.

Η συνάρτηση loadvr():

Στη συνάρτηση αυτή παράγουμε τον κώδικα για να διαβαστεί η τιμή μιας μεταβλητής από τη μνήμη, δηλαδή από μια θέση στη στοίβα και την μεταφέρει σε έναν καταχωρητή.

Πρόκειται να διακρίνουμε ορισμένες περιπτώσεις καθώς βασιζόμαστε στην πληροφορία που επιστρέφει ο πινάκας συμβόλων και ανάλογα με την περίπτωση παράγεται και ο αντίστοιχος κώδικας.

Η πρώτη περίπτωση και η πιο απλή είναι η εκχώρηση αριθμητικής σταθεράς σε έναν καταχωρητή. Στη συγκεκριμένη περίπτωση δεν κάνουμε τίποτα άλλο από το να δημιουργούμε την εξής εντολή:

li, reg, var

Πριν προχωρήσουμε στις υπόλοιπες περιπτώσεις χρειάζεται να αντλήσουμε πληροφορία από τον πινάκα συμβόλων. Το πρώτο πράγμα που κάνουμε είναι με την βοήθεια της συνάρτησης find_entry() να βρίσκουμε την μεταβλητή εντός του πίνακα συμβόλων και αφού βρεθεί να κρατάμε το offset που έχει στο εγγράφημα δραστηριοποίησης που ανήκει.

Συνεχίζοντας , ακολουθεί έλεγχος στον οποιο, με την βοήθεια της συνάρτησης is_local , διακρίνουμε εάν η μεταβλητή είναι τοπική. Στην περίπτωση που είναι ακολουθούν δυο υποκατηγορίες:

- Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή ή προσωρινή μεταβλητή : Η τιμή της μεταβλητής που ζητάμε βρίσκεται αποθηκευμένη στο εγγράφημα δραστηριοποίησης της συνάρτησης που μεταφράζεται. Για να μεταφερθεί μια τέτοια μεταβλητή στον καταχωρητή reg έχουμε την εντολή:

lw reg,-offset(sp)

- Παράμετρος που έχει περαστεί με αναφορά: Στη συγκεκριμένη υποπερίπτωση χρειαζόμαστε ένα ακόμη βήμα καθώς στο εγγράφημα δραστηριοποίησης της συνάρτησης έχει περαστεί η διεύθυνση της. Πρώτα, πρέπει να μεταφερθεί η διεύθυνση της μεταβλητής από τη στοίβα σε έναν καταχωρητή και έπειτα να χρησιμοποιηθεί αυτός ο καταχωρητής ώστε να μεταφερθεί στον reg η τιμή της μεταβλητής. Συνεπώς έχουμε:

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

Στη συνέχεια, εξετάζουμε το ενδεχόμενο εάν η μεταβλητή είναι καθολική. Η συγκεκριμένη περίπτωση είναι αρκετά απλή καθώς η προσπέλαση στις καθολικές μεταβλητές γίνεται εύκολη χρησιμοποιώντας τον καταχωρητή gp. Ετσι όταν γνωρίζουμε ότι η μεταβλητή είναι καθολική και έχει ένα συγκεκριμένο offset , τότε η πρόσβαση σε αυτή γίνεται με την εντολή:

lw reg,-offset(gp)

Οι υπόλοιπες περιπτώσεις που ακολουθούν ανήκουν στο ενδεχόμενο που η μεταβλητή που αναζητούμε προκείμενου να αποθηκεύσουμε την τιμή της ανήκει σε κάποιον πρόγονο. Αυτές οι περιπτώσεις είναι :

- Τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή η οποία ανήκει σε πρόγονο: Χρησιμοποιώντας την βοηθητική συνάρτηση gnlvcode() αναζητούμε την μεταβλητή σε όλα τα εγγραφήματα δραστηριοποίησης των προγόνων της συνάρτησης που βρίσκονται στη στοίβα. Μόλις βρεθεί δεν έχουμε πάρα να διαβάσουμε το περιεχόμενο της θέσης μνήμης. Άρα έχουμε:

  gnlvcode()
  lw reg,(t0)

- Παράμετρος που έχει περαστεί με αναφορά, η οποία ανήκει σε πρόγονο: Παρόμοια και εδώ χρησιμοποιείται η gnlvcode() που θα πάρει ως παράμετρο τη μεταβλητή που θέλουμε να διαβάσουμε και θα τοποθετήσουμε στον reg τη διεύθυνση στην οποία βρίσκεται αποθηκευμένη η διεύθυνση της μεταβλητής. Αφού διαβαστεί το περιεχόμενο της θέσης μνήμης που η gnlvcode() στον reg μπορούμε να φτάσουμε στην τιμή της μεταβλητής χρησιμοποιώντας τη. Εδώ έχουμε τις εντολές:

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

Η συνάρτηση storerv():

Η συνάρτηση storerv() δεν πρόκειται να αναλυθεί όπως έγινε για τις προηγούμενες συναρτήσεις. Ο λόγος βρίσκεται στο γεγονός ότι η υλοποίηση της είναι σε πολύ μεγάλο βαθμό ίδια με αυτή της συνάρτησης loadvr() , με τη διάφορα που αντί για εντολή ανάγνωσης lw, έχουμε εντολή αποθήκευσης sw.

Εντολές άλματος

Για την παρακάτω αντιστοίχιση λογικών αλμάτων στον ενδιάμεσο και στον τελικό κώδικα

cond_jump_int_code	cond_jump_fin_code
==	beq
<>	bne
<	blt
>	bgt
<=	ble
>=	bge

Παράγονται οι εξής κανόνες τελικού κώδικα για το λογικό άλμα cond_jump_intcode, x, y, label

```
loadvr(x,t1)
loadvr(y,t2)
produce('cond_jump_fin_code t1,t2,label')
```

Στην ετικέτα main θα υπάρχουν οι εξής δύο εντολές:

```
main:
    addi sp,sp,framelength_main
    mv gp,sp
```

Όταν θα συναντήσουμε στην τελευταία εντολή, η οποία είναι η halt, ο τερματισμός του προγράμματος θα γίνει με τις εντολές

```
li a0,0
li a7,93
ecall
```

Πέρασμα παραμέτρων, κλήση συναρτήσεων και διαδικασιών στον Τελικό κώδικα

Η κλήση μιας συνάρτησης ή διαδικασίας προϋποθέτει τη δημιουργία ενός νέου εγγραφήματος δραστηριοποίησης και την τοποθέτηση σε αυτό όποιας πληροφορίας απαιτείται για τη λειτουργία του, συμπεριλαμβανομένης και της πληροφορίας που σχετίζεται με το πέρασμα παραμέτρων και επιστροφή αποτελέσματος. Προϋποθέτει επίσης τη μετάβαση του ελέγχου σε αυτήν, αλλά και την επιστροφή του ελέγχου στην καλούσα συνάρτηση, όταν η κληθείσα ολοκληρωθεί.

Πέρασμα Παραμέτρων

Οι παράμετροι μιας συνάρτησης θα τοποθετηθούν στο εγγράφημα δραστηριοποίησης αμέσως πάνω από τα 12 δεσμευμένα bytes. Πριν κάνουμε όμως οποιαδήποτε ενέργεια για το πέρασμα της πρώτης παραμέτρου μιας συνάρτησης, τοποθετούμε τον fp στη θέση του. Προσθέτουμε στον sp τόσα bytes, από όσα αποτελείται το εγγράφημα δραστηριοποίησης της καλούσας. Αν η συνάρτηση ή διαδικασία δεν έχει παραμέτρους παράγουμε την παρακάτω εντολή όταν εμφανιστεί η call της συνάρτησης.

addi fp, sp, framelength

Πέρασμα παραμέτρων με Τιμή

Κατά το πέρασμα της παραμέτρου με τιμή, η τιμή της παραμέτρου αντιγράφεται στο εγγράφημα δραστηριοποίησης της υπό δημιουργία συνάρτησης, στη θέση που έχει δεσμευτεί για την παράμετρο αυτή.
Τοποθετούμε προσωρινά την τιμή της μεταβλητής σε καταχωρητή με τη χρήση της loadvr, υπολογίζουμε το offset στο οποίο θα πρέπει να τοποθετηθεί στο εγγράφημα και αντιγράφουμε την τιμή της από τον καταχωρητή στη στοίβα. Οι εντολές που θα χρησιμοποιήσουμε είναι οι εξής:

loadvr(arg, “t0”)
d = 12 + (i-1) * 4 bytes
sw t0, -d(fp)

Πέρασμα παραμέτρων με αναφορά

Στο πέρασμα παραμέτρων με αναφορά αντιγράφουμε στο εγγράφημα της υπό δημιουργίας συνάρτησης η διεύθυνση της μεταβλητής αυτής. Η πρόσβαση στη μεταβλητή που περάστηκε ως παράμετρος γίνεται μέσω αυτής της διεύθυνσης.
Ακολουθούν οι διαφορετικές περιπτώσεις παραμέτρων με αναφορά:
Παράμετρος με αναφορά, όταν στη στοίβα είναι αποθηκευμένη η τιμή της παραμέτρου

Α) τοπική μεταβλητή ή προσωρινή μεταβλητή ή παράμετρος που έχει περαστεί με τιμή στη συνάρτηση. Παράγουμε τις εξής εντολές:

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

Β) τοπική μεταβλητή ή παράμετρος που έχει περαστεί με τιμή σε συνάρτηση πρόγονο στη οποία ανήκει. Παραγάγουμε τις εξής εντολές:

gnlvcode()
sw t0, -d(fp)

Γ) καθολική μεταβλητή. Παράγουμε τις εξής εντολές:

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

Παράμετρος με αναφορά, όταν στη στοίβα βρίσκεται η διεύθυνση της παραμέτρου.

Α) παράμετρος που έχει περαστεί με αναφορά στην καλούσα συνάρτηση ή διαδικασία. Παράγουμε τις εξής εντολές:

lw t0, -offset(sp)
sw t0, -d(fp)

Β) παράμετρος που έχει περαστεί με αναφορά σε συνάρτηση ή διαδικασία πρόγονο

gnlvcode()
lw t0, (t0)
sw t0, -d(fp)

Επιστροφή τιμής συνάρτησης

Όταν μεταφράζουμε την εντολή επιστροφής
par, x, ret, _
πρέπει να μεταφέρουμε τη διεύθυνση της προσωρινής μεταβλητής x της καλούσας στην Τρίτη θέση του εγγραφήματος δραστηριοποίησης της κληθείσας. Παράγουμε τις εξής εντολές:

addi t0, sp, -offset
sw t0, -8(fp)

Όταν πρέπει να μεταφράσουμε την εντολή ενδιάμεσου κώδικα
ret, z, _, _
τότε πρέπει η τιμή του z να αντιγραφεί, μέσω του δείκτη που είναι αποθηκευμένος στη Τρίτη θέση του εγγραφήματος της κληθείσας, στην προσωρινή μεταβλητή που έχει δημιουργηθεί για τον σκοπό αυτό στην καλούσα. Έτσι παράγουμε τις εξής εντολές:

loadvr(z, "t1")
lw t0, -8(sp)
sw t1(t0)

Κλήση συνάρτησης

Όταν καλείται μια συνάρτηση με την εντολή call προκύπτουν δύο περιπτώσεις.

Αν μια συνάρτηση ή διαδικασία γονέας καλεί τη συνάρτηση ή διαδικασία παιδί της, τότε παράγουμε την εξής εντολή:

sw sp, -4(fp)

Αν μια συνάρτηση καλεί τη συνάρτηση αδελφό της ή τον εαυτό της αναδρομικά τότε παράγουμε τις εξής εντολές:

lw t0, -4(sp)
Sw t0, -4(fp)

Ακόμα, κατά την κλήση μιας συνάρτησης ο δείκτης στοίβας δείχνει το εγγράφημα δραστηριοποίησης του υποπρογράμματος το οποίο εκείνη τη στιγμή εκτελείται. Έτσι πριν την μεταβίβαση ελέγχου ροής, πρέπει να τοποθετηθεί ο δείκτης στοίβας στο εγγράφημα δραστηριοποίησης της κληθείσας με την εντολή:

add sp, sp, frame_length

Για να καλέσουμε τη συνάρτηση χρησιμοποιούμε την jal εντολή, πηγαίνοντας στον αντίστοιχο Label που αρχίζει ο κώδικας της συνάρτησης.
Αντίστοιχα μόλις ολοκληρωθεί η συνάρτηση έχουμε την:

add sp, sp, -frame_length

Testing

Ακολουθούν screenshot από την εκτέλεση ενός πλήρως προγράμματος (21_final_code_from_lecture) που έχουμε επισυνάψει. Ακολουθεί ο τελικός κώδικας που παράχθηκε.

```
    .data
    str_nl: .asciz "\n"
    .text


L0:
    j Lmain

L1:
    sw ra, 0(sp)

L2:
    li t1, 4
    sw t1, -12(sp)

L3:
    lw t1, -16(gp)
    sw t1, -12(gp)

L4:
    lw t0, -4(sp)
    addi t0, t0, -12
    lw t1, 0(t0)
    lw t0, -4(sp)
    addi t0, t0, -16
    lw t0, 0(t0)
    sw t1, 0(t0)

L5:
    lw t0, -4(sp)
    addi t0, t0, -20
    lw t1, 0(t0)
    lw t2, -12(sp)
    add t1, t1, t2
    sw t1, -16(sp)

L6:
    lw t1, -16(sp)
    lw t0, -8(sp)
    sw t1, 0(t0)
```

```
L7:
    lw ra, 0(sp)
    jr ra

L8:
    sw ra, 0(sp)

L9:
    li t1, 3
    sw t1, -20(sp)

L10:
    lw a0, -12(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L11:
    lw a0, -16(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L12:
    addi fp, sp, 20
    addi t0, sp, -24
    sw t0, -8(fp)

L13:
    sw sp, -4(fp)
    addi sp, sp, 20
    jal L1
    addi sp, sp, -20
```

```
L14:
    lw t1, -24(sp)
    sw t1, -16(gp)

L15:
    lw a0, -12(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L16:
    lw a0, -16(gp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L17:
    lw ra, 0(sp)
    jr ra

L18:
Lmain:
    addi sp, sp, 20
    mv gp, sp

L19:
    li t1, 1
    sw t1, -12(sp)

L20:
    li t1, 2
    sw t1, -16(sp)
```

```
L21:
    addi fp, sp, 28
    lw t0, -12(sp)
    sw t0, -12(fp)

L22:
    addi t0, sp, -16
    sw t0, -16(fp)

L23:
    sw sp, -4(fp)
    addi sp, sp, 28
    jal L8
    addi sp, sp, -28

L24:
    lw a0, -12(sp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L25:
    lw a0, -16(sp)
    li a7, 1
    ecall
    la a0, str_nl
    li a7, 4
    ecall

L26:
    li a0, 0
    li a7, 93
    ecall

L27:
```