Τμήμα Μηχανικών Η/Υ & Πληροφορικής

ΜΥΥ802 Μεταφραστές

Κατασκευή Μεταγλωττιστή για τη Γλώσσα Greek++

Θεοφάνης Τομπόλης 4855

Αθανάσιος Φυτιλής 5381

Περιεχόμενα

1.Εισαγωγή

Η εργασία εκπονήθηκε στα πλαίσια του μαθήματος "Μεταφραστές" του Τμήματος Μηχανικών Η/Υ & Πληροφορικής του Πανεπιστημίου Ιωαννίνων. Κύριος στόχος της εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός πλήρους μεταγλωττιστή για την εκπαιδευτική γλώσσα προγραμματισμού Greek++.

1.1. Σκοπός της Εργασίας

- Κατανόηση των βασικών εννοιών της θεωρίας των μεταφραστών.
- Σχεδιασμός και σύνταξη γραμματικής για μια νέα γλώσσα προγραμματισμού.
- Υλοποίηση λεκτικού, συντακτικού και σημασιολογικού αναλυτή.
- Παραγωγή ενδιάμεσου κώδικα και κατανόηση της σημασίας του.
- Παραγωγή τελικού κώδικα για μια συγκεκριμένη αρχιτεκτονική (RISC-V).
- Εξοικείωση με τη διαχείριση του πίνακα συμβόλων και των πλαισίων ενεργοποίησης.
- Ανάπτυξη δεξιοτήτων στην ομαδική εργασία και συνεργασία για την ανάπτυξη κώδικα.

1.2. Η Γλώσσα Προγραμματισμού greek++

Η greek++ είναι μια μικρή, διαδικαστική γλώσσα προγραμματισμού, σχεδιασμένη με εκπαιδευτικούς σκοπούς, υποστηρίζοντας κυρίως ακέραιους αριθμούς και βασικές δομές. Τα κύρια χαρακτηριστικά της greek++ περιλαμβάνουν:

- Δηλώσεις μεταβλητών
- Αριθμητικές και λογικές εκφράσεις.
- Εντολές ανάθεσης.
- Δομές ελέγχου ροής:
    - Απόφασης (εάν-τότε-αλλιώς).
    - Επανάληψης (όσο, επανάλαβε-μέχρι, για).
- Υποπρογράμματα:
    - Συναρτήσεις
    - Διαδικασίες
- Μηχανισμούς περάσματος παραμέτρων
- Αναδρομικές κλήσεις υποπρογραμμάτων.

2. Λεκτική Ανάλυση

Η λεκτική ανάλυση αποτελεί την πρώτη φάση του μεταγλωττιστή. Ο ρόλος της είναι να διαβάσει τον πηγαίο κώδικα της Greek++ ως μια ακολουθία χαρακτήρων και να τον ομαδοποιήσει σε μια ακολουθία από λεκτικές μονάδες (tokens). Κάθε λεκτική μονάδα αντιστοιχεί σε ένα βασικό δομικό στοιχείο της γλώσσας, όπως μια λέξη-κλειδί, ένα αναγνωριστικό, μια σταθερά, ή ένας τελεστής.

2.1 Λεκτικές Μονάδες της greek++

Ο λεκτικός αναλυτής που υλοποιήθηκε για την greek++ αναγνωρίζει τις ακόλουθες κατηγορίες λεκτικών μονάδων, σύμφωνα με τις προδιαγραφές της γλώσσας:

- **Λέξεις-Κλειδιά (Keywords):** Ένα πεπερασμένο σύνολο λέξεων με προκαθορισμένη σημασία στη γλώσσα, οι οποίες δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά. Παράδειγμα: πρόγραμμα, δήλωση, εάν, τότε, αλλιώς, όσο, επανάλαβε, για, συνάρτηση, διαδικασία, και, ή, όχι, εκτέλεσε κ.ά. Το πλήρες σύνολο ορίζεται στη σταθερά KEYWORDS του πηγαίου κώδικα του μεταγλωττιστή..

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

- **Αναγνωριστικά (Identifiers/Variables):** Ονόματα που αποδίδονται από τον προγραμματιστή σε οντότητες όπως μεταβλητές, συναρτήσεις και διαδικασίες. Σύμφωνα με τους κανόνες της Greek++, ένα αναγνωριστικό πρέπει να ξεκινά με ελληνικό ή λατινικό γράμμα ή με τον χαρακτήρα της κάτω παύλας (_), και μπορεί να ακολουθείται από οποιονδήποτε συνδυασμό γραμμάτων, αριθμητικών ψηφίων ή κάτω παύλας. Το μέγιστο επιτρεπτό μήκος για ένα αναγνωριστικό είναι 30 χαρακτήρες.

```
# --- Identifiers and Keywords ---
if char.isalpha() or char == '_':
    identifier = ''
    while i < length and (code[i].isalnum() or code[i] == '_'):
        identifier += code[i]
        i += 1
    if identifier in KEYWORDS:
        tokens.append(('KEYWORD', identifier, line))
    elif len(identifier) <= 30:
        tokens.append(('VARIABLE', identifier, line))
    else:
        tokens.append(('MISMATCH', f"Identifier '{identifier}' exceeds 30 characters", line))
    continue
```

- **Αριθμητικές Σταθερές (Numbers):**
  - Ακέραιοι (Integers): Ακολουθίες ενός ή περισσότερων ψηφίων. Ο υλοποιημένος λεκτικός αναλυτής επιπλέον ελέγχει αν η τιμή του ακεραίου βρίσκεται εντός του επιτρεπτού εύρους τιμών από -32768 έως 32767.
  - Πραγματικοί (Floats): Ακολουθίες ψηφίων που περιέχουν ακριβώς μία υποδιαστολή.(Τα οποία στην συνέχεια δεν χρησιμοποιήθηκαν αφού τελικά η Greek++ υποστηρίζει μόνο ακαίρους).

```
# --- Numbers (Integer and Float) ---
if char.isdigit() or (char == '.' and i + 1 < length and code[i + 1].isdigit()):
    number = ''
    dot_count = 0
    while i < length and (code[i].isdigit() or code[i] == '.'):
        if code[i] == '.':
            dot_count += 1
        number += code[i]
        i += 1

    if dot_count > 1 or number == '.':
        tokens.append(('MISMATCH', f"Invalid number format '{number}'", line))
    else:
        num_type = "FLOAT" if dot_count == 1 else "INTEGER"
        try:
            num_value = float(number) if num_type == "FLOAT" else int(number)
            if num_type == "INTEGER" and (num_value > 32767 or num_value < -32768):
                tokens.append(('MISMATCH', f"Integer '{number}' out of allowed range (-32768 to 32767)", line))
            tokens.append((num_type, num_value, line))
        except ValueError:
            tokens.append(('MISMATCH', f"Could not convert number '{number}'", line))
        continue
```

- **Τελεστές (Operators):** Σύμβολα που δηλώνουν την εκτέλεση αριθμητικών, σχεσιακών πράξεων ή πράξεων ανάθεσης. Το σύνολο των τελεστών περιλαμβάνει
  - Αριθμητικούς: +, -, *, /.
  - Σχεσιακούς: =, ==, <, >, <=, >=, <>, <-.
  - Ανάθεσης: :=.

```
# --- Operators and Punctuation ---
# Check for two-character operators first
if i + 1 < length:
    two_char = code[i:i + 2]
    if two_char in OPERATORS:
        token_type = 'ASSIGNMENT' if two_char == ASSIGNMENT else 'OPERATOR'
        tokens.append((token_type, two_char, line))
        i += 2
        continue
# Check for single-character operators and punctuation
if char in OPERATORS:
    tokens.append(('OPERATOR', char, line))
    i += 1
    continue
if char in PUNCTUATION:
    if char not in '{}':
        tokens.append(('PUNCTUATION', char, line))
    i += 1
    continue
if char == PARAMETER_PASS:
    tokens.append(('PARAMETER_PASS', char, line))
    i += 1
    continue
```

```
OPERATORS = {'+', '-', '*', '/', '=', '==', '<', '>', '<=', '>=', '<>', ':=', '<-'}
RELATIONAL_OPERATORS = {'=', '==', '<', '>', '<=', '>=', '<>', '<-'}
LOGICAL_OPERATORS = {'και', 'ή', 'όχι'}
ARITHMETIC_OPERATORS = {'+', '-', '*', '/'}
ASSIGNMENT = ':='
PUNCTUATION = {';', ',', '(', ')', '{', '}', '[', ']'}
PARAMETER_PASS = '%'
```

- **Διαχωριστικά/Σημεία Στίξης (Punctuation):** Σύμβολα που συμβάλλουν στη συντακτική δομή του προγράμματος, όπως το ελληνικό ερωτηματικό, το κόμμα , οι παρενθέσεις , και οι αγκύλες.
- **Σύμβολο Περάσματος Παραμέτρου (Parameter Pass Specifier):** Ο ειδικός χαρακτήρας % χρησιμοποιείται για να υποδηλώσει το πέρασμα μιας παραμέτρου με αναφορά κατά την κλήση υποπρογράμματος.
- **Σχόλια (Comments):** Κείμενο που προορίζεται για ανθρώπινη ανάγνωση και αγνοείται από τον μεταγλωττιστή. Τα σχόλια στην greek++ περικλείονται από τους χαρακτήρες ‘{‘ και ‘}’. Ο λεκτικός αναλυτής υποστηρίζει και την αναγνώριση φωλιασμένων σχολίων.

2.2 Σχεδιασμός και Υλοποίηση του Λεκτικού Αναλυτή

Η προσέγγιση που ακολουθήθηκε βασίζεται στην επαναληπτική σάρωση του πηγαίου κώδικα, χαρακτήρα προς χαρακτήρα, και στην εφαρμογή κανόνων για την αναγνώριση των προτύπων (patterns) που αντιστοιχούν στις διάφορες κατηγορίες λεκτικών μονάδων. Η λογική αυτή προσομοιώνει την λειτουργία ενός πεπερασμένου αυτόματου.

1. **Αρχικοποίηση:** Ένας δείκτης (i) αρχικοποιείται στην αρχή του πηγαίου κώδικα και ένας μετρητής γραμμών (line) αρχικοποιείται στην τιμή 1.
2. **Επανάληψη:** Ο αναλυτής επαναλαμβάνεται όσο ο δείκτης δεν έχει φτάσει στο τέλος του κώδικα.
3. **Παράλειψη Λευκών Χαρακτήρων:** Εάν ο τρέχων χαρακτήρας είναι κενό ή tab, ο δείκτης προχωρά. Εάν είναι αλλαγή γραμμής, ο μετρητής γραμμών αυξάνεται και ο δείκτης προχωρά.
4. **Αναγνώριση Σχολίων:**
    - Εάν ο τρέχων χαρακτήρας είναι '{', ξεκινά η αναγνώριση σχολίου. Ένας μετρητής επιπέδου (comment_level) χρησιμοποιείται για την υποστήριξη φωλιασμένων σχολίων.
    - Ο αναλυτής καταναλώνει χαρακτήρες, αυξάνοντας τον μετρητή γραμμών για κάθε νέα γραμμή (\n), μέχρι ο comment_level να μηδενιστεί (δηλαδή, να βρεθεί ο αντίστοιχος κλείνοντας χαρακτήρας '}').
    - Εάν το τέλος του αρχείου (EOF) επιτευχθεί πριν κλείσουν όλα τα επίπεδα σχολίων, καταγράφεται σφάλμα.
5. **Αναγνώριση Τελεστών και Σημείων Στίξης:**
    - Ελέγχεται πρώτα η πιθανότητα αναγνώρισης τελεστή δύο χαρακτήρων (π.χ., :=, ==, <=, >=, <>, <-) εξετάζοντας τον τρέχοντα και τον επόμενο χαρακτήρα.
    - Εάν δεν ταιριάζει, ελέγχεται αν ο τρέχων χαρακτήρας ανήκει στο σύνολο των μονοχαρακτηριακών τελεστών ή των σημείων στίξης (εξαιρουμένων των '{' και '}' που χειρίζονται τα σχόλια). Ελέγχεται επίσης ο ειδικός χαρακτήρας '%' για το πέρασμα παραμέτρων.

6. **Αναγνώριση Αναγνωριστικών και Λέξεων-Κλειδιών:**
    - Εάν ο τρέχων χαρακτήρας είναι γράμμα (ελληνικό ή λατινικό) ή κάτω παύλα, ξεκινά η συλλογή μιας ακολουθίας από αλφαριθμητικούς χαρακτήρες και κάτω παύλες.
    - Η συλλεγόμενη συμβολοσειρά συγκρίνεται με το σύνολο των KEYWORDS. Εάν ανήκει σε αυτό, χαρακτηρίζεται ως KEYWORD.
    - Διαφορετικά, ελέγχεται το μήκος της. Αν είναι μικρότερο ή ίσο των 30 χαρακτήρων, χαρακτηρίζεται ως VARIABLE. Αλλιώς, καταγράφεται σφάλμα υπέρβασης μήκους.
7. **Αναγνώριση Αριθμών:**
    - Εάν ο τρέχων χαρακτήρας είναι ψηφίο, ή αν είναι υποδιαστολή (.) και ακολουθείται από ψηφίο, ξεκινά η συλλογή ενός αριθμού.
    - Κατά τη συλλογή, καταμετρώνται οι υποδιαστολές. Εάν βρεθούν περισσότερες από μία, ή αν ο "αριθμός" αποτελείται μόνο από μια υποδιαστολή, καταγράφεται σφάλμα.
    - Ανάλογα με την παρουσία (μίας) υποδιαστολής, ο αριθμός κατηγοριοποιείται ως FLOAT ή INTEGER. Για τους ακεραίους, πραγματοποιείται έλεγχος ώστε η τιμή τους να μην υπερβαίνει το εύρος [-32768, 32767].
8. **Αναφορά Σφάλματος:** Εάν ο τρέχων χαρακτήρας δεν μπορεί να αντιστοιχιστεί σε καμία από τις παραπάνω κατηγορίες, καταγράφεται ως MISMATCH (μη αναμενόμενος χαρακτήρας).

```
            continue

        # --- Mismatched/Unexpected Character ---
        tokens.append(('MISMATCH', f"Unexpected character '{char}'", line))
        i += 1

    return tokens
```

Στην περίπτωση που εντοπιστεί κάποιο από τα παραπάνω σφάλματα, ο λεκτικός αναλυτής δεν διακόπτει άμεσα τη λειτουργία του, αλλά καταγράφει το σφάλμα προσθέτοντας μια ειδική λεκτική μονάδα τύπου MISMATCH στη λίστα εξόδου. Αυτή η μονάδα περιέχει μια περιγραφή του σφάλματος και τον αριθμό γραμμής όπου αυτό συνέβη. Η παρουσία έστω και μίας MISMATCH μονάδας σηματοδοτεί την αποτυχία της λεκτικής ανάλυσης, οδηγώντας συνήθως στη διακοπή της διαδικασίας μεταγλώττισης πριν την έναρξη των επόμενων φάσεων. Στην περίπτωση μη κλεισμένου σχολίου, η λεκτική ανάλυση τερματίζεται αμέσως μετά την καταγραφή του σφάλματος.

2.3 Διαχείριση Λεκτικών Σφαλμάτων

Ο λεκτικός αναλυτής είναι επιφορτισμένος με τον εντοπισμό σφαλμάτων που αφορούν την εσφαλμένη δόμηση των στοιχειωδών λεκτικών μονάδων. Τα σφάλματα που ανιχνεύονται περιλαμβάνουν:

1. **Μη Κλεισμένο Σχόλιο:** Όταν το τέλος του αρχείου (EOF) επιτυγχάνεται ενώ ένα ή περισσότερα επίπεδα σχολίων παραμένουν ανοιχτά.
2. **Μη Αναμενόμενος Χαρακτήρας:** Όταν συναντάται ένας χαρακτήρας ο οποίος δεν ανήκει στο ορισμένο αλφάβητο της Greek++ ούτε μπορεί να σχηματίσει μέρος κάποιας έγκυρης λεκτικής μονάδας.
3. **Υπέρβαση Μήκους Αναγνωριστικού:** Όταν ένα αναγνωριστικό αποτελείται από περισσότερους από 30 χαρακτήρες.
4. **Λανθασμένη Μορφή Αριθμού:** Για παράδειγμα, ένας αριθμός με πολλαπλές υποδιαστολές (π.χ., 3.1.4) ή ένας αριθμός που αποτελείται αποκλειστικά από μία υποδιαστολή (.).
5. **Ακέραιος Εκτός Εύρους:** Όταν μια ακέραια σταθερά είναι μικρότερη από -32768 ή μεγαλύτερη από 32767.

2.4 Διεπαφή με τον Συντακτικό Αναλυτή

Η έξοδος του λεκτικού αναλυτή είναι μια λίστα από πλειάδες (tuples), όπου κάθε πλειάδα αντιπροσωπεύει μια λεκτική μονάδα και έχει την ακόλουθη δομή.

(TYPE, VALUE, LINE_NUMBER):

- **Type:** Μια συμβολοσειρά που περιγράφει την κατηγορία της λεκτικής μονάδας (π.χ., 'KEYWORD', 'VARIABLE', 'INTEGER', 'OPERATOR', 'PUNCTUATION', 'MISMATCH').
- **VALUE:** Η πραγματική συμβολοσειρά ή η αριθμητική τιμή που αναγνωρίστηκε (π.χ., 'πρόγραμμα', 'counter', 123, 3.14, '+'). Για λεκτικές μονάδες τύπου MISMATCH, αυτό το πεδίο περιέχει το μήνυμα σφάλματος.
- **Line_Number:** Ο αριθμός της γραμμής στον πηγαίο κώδικα όπου εντοπίστηκε η αρχή της συγκεκριμένης λεκτικής μονάδας. Αυτή η πληροφορία είναι κρίσιμη για την ακριβή αναφορά σφαλμάτων από τις επόμενες φάσεις του μεταγλωττιστή.

3. Συντακτική Ανάλυση (Syntax Analysis)

Μετά την επιτυχή ολοκλήρωση της λεκτικής ανάλυσης και τη μετατροπή του πηγαίου κώδικα σε μια ακολουθία λεκτικών μονάδων (tokens), ο συντακτικός αναλυτής αναλαμβάνει να ελέγξει εάν αυτή η ακολουθία ακολουθεί τους γραμματικούς κανόνες της

γλώσσας Greek++. Ουσιαστικά, εξετάζει την ιεραρχική δομή του προγράμματος, επιβεβαιώνοντας ότι οι λεκτικές μονάδες συνδυάζονται με τρόπους που επιτρέπονται από τη γραμματική. Πέρα από τον έλεγχο ορθότητας, ο συντακτικός αναλυτής που υλοποιήθηκε παράγει μια ενδιάμεση αναπαράσταση του προγράμματος, γνωστή ως Αφηρημένο Συντακτικό Δέντρο (Abstract Syntax Tree - AST), η οποία διευκολύνει τις επόμενες φάσεις της μεταγλώττισης.

3.1 Γραμματική της Greek++

Η συντακτική δομή της Greek++ ορίζεται από μια τυπική γραμματική χωρίς συμφραζόμενα. Όπως φαίνεται πιο κάτω.

Γραμματική της greek++

```
program          :  'πρόγραμμα' ID programblock
                 ;

programblock     :  declarations subprograms
                       'αρχή_προγράμματος' sequence 'τέλος_προγράμματος'
                 ;

declarations     :  ( 'δήλωση' varlist )*
                 |
                 ;

varlist          :  ID ( ',' ID )*
                 ;

subprograms      :  ( func | proc )*
                 ;

func             :  'συνάρτηση' ID '(' formalparlist  ')' funcblock
                 ;

proc             :  'διαδικασία' ID '(' formalparlist  ')' procblock
                 ;

formalparlist    :  varlist
                 |
                 ;

funcblock        :  'διαπροσωπεία' funcinput funcoutput declarations
                       subprograms
                       'αρχή_συνάρτησης' sequence 'τέλος_συνάρτησης'
                 ;

procblock        :  'διαπροσωπεία' funcinput funcoutput declarations
                       subprograms
                       'αρχή_διαδικασίας' sequence 'τέλος_διαδικασίας'
                 ;

funcinput        :  'είσοδος' varlist
                 |
                 ;

funcoutput       :  'έξοδος' varlist
                 |
                 ;
```

```
sequence            :   statement ( ';' statement )*
                    ;

statement           :   assignment_stat
                        |   if_stat
                        |   while_stat
                        |   do_stat
                        |   for_stat
                        |   input_stat
                        |   print_stat
                        |   call_stat
                        ;

assignment_stat     :   ID ':=' expression
                    ;

if_stat             :   'εάν' condition 'τότε' sequence elsepart 'εάν_τέλος'
                    ;

elsepart            :   'αλλιώς' sequence
                    |
                    ;

while_stat          :   'όσο' condition 'επανάλαβε' sequence 'όσο_τέλος'
                    ;

do_stat             :   'επανάλαβε' sequence 'μέχρι' condition
                    ;

for_stat            :   'για' ID ':=' expression 'έως' expression  step
                          'επανάλαβε' sequence 'για_τέλος'
                    ;

step                :   'με_βήμα' expression
                    |
                    ;

print_stat          :   'γράψε' expression
                    ;

input_stat          :   'διάβασε' ID
                    ;

call_stat           :   'εκτέλεσε' ID idtail
                    ;
```

```
idtail              :   actualpars
                    |
                    ;

actualpars          :   '(' actualparlist ')'
                    ;

actualparlist       :   actualparitem ( ',' actualparitem )*
                    |
                    ;

actualparitem       :   expression |  '%'ID
                    ;

condition           :   boolterm ( 'ή' boolterm )*
                    ;

boolterm            :   boolfactor ( 'και' boolfactor )*
                    ;

boolfactor          :   'όχι' '[' condition ']'
                    |   '[' condition ']'
                    |   expression relational_oper expression
                    ;

expression          :   optional_sign term ( add_oper term )*
                    ;

term                :   factor  ( mul_oper factor )*
                    ;

factor              :   INTEGER
                    |   '(' expression ')'
                    |   ID idtail
                    ;

relational_oper     :   '=' | '<=' | '>=' | '<>' | '<'  | '>'
                    ;

add_oper            :   '+' | '-'
                    ;

mul_oper            :   '*' | '/'
                    ;

optional_sign       :   add_oper
                    |
                    ;
```

Η γραμματική περιλαμβάνει κανόνες παραγωγής για όλες τις δομές της γλώσσας, όπως η συνολική δομή ενός προγράμματος, οι δηλώσεις μεταβλητών, ο ορισμός συναρτήσεων και διαδικασιών, οι παράμετροι, οι διάφορες εντολές (ανάθεσης, εαν, οσο, για, διάβασε, γράψε, εκτέλεσε) και οι εκφράσεις (αριθμητικές, λογικές).

3.2. Σχεδιασμός και Υλοποίηση του Συντακτικού Αναλυτή

Για την υλοποίηση του συντακτικού αναλυτή επιλέχθηκε η μέθοδος της Αναδρομικής Κατάβασης (Recursive Descent Parsing). Αυτή η προσέγγιση είναι ιδιαίτερα κατάλληλη για γραμματικές LL.

Η λογική λειτουργίας του αναλυτή είναι η εξής:

- **Αντιστοίχιση Μη-Τερματικών σε Συναρτήσεις:** Κάθε συνάρτηση είναι υπεύθυνη για την αναγνώριση της ακολουθίας λεκτικών μονάδων που αντιστοιχεί στο μη-τερματικό σύμβολο που αντιπροσωπεύει.
- **Κατανάλωση Τερματικών Συμβόλων:** Όταν ένας κανόνας παραγωγής απαιτεί ένα συγκεκριμένο τερματικό σύμβολο (π.χ., μια λέξη-κλειδί), η αντίστοιχη συνάρτηση «parsing» χρησιμοποιεί τη μέθοδο «expect» για να επιβεβαιώσει την παρουσία του αναμενόμενου «token» και να το καταναλώσει, προχωρώντας τον δείκτη στην επόμενη λεκτική μονάδα. Εάν το τρέχον «token» δεν είναι το αναμενόμενο, η «expect» προκαλεί συντακτικό σφάλμα.
- **Επιλογή μεταξύ Εναλλακτικών Κανόνων:** Όταν ένα μη-τερματικό σύμβολο έχει πολλαπλούς κανόνες παραγωγής (π.χ., ‘η’ statement μπορεί να είναι assignment_stat | if_stat | ...), η συνάρτηση parsing εξετάζει την τρέχουσα λεκτική μονάδα (lookahead) για να αποφασίσει ποιον κανόνα θα ακολουθήσει. Για παράδειγμα, αν το τρέχον token είναι η λέξη-κλειδί 'εάν', καλείται η συνάρτηση «if_statement».
- **Αναδρομικές Κλήσεις:** Όταν ένας κανόνας παραγωγής περιέχει ένα μη-τερματικό σύμβολο στο δεξί του μέλος, η τρέχουσα συνάρτηση «parsing» καλεί αναδρομικά τη συνάρτηση που αντιστοιχεί σε εκείνο το μη-τερματικό σύμβολο.
- **Διαχείριση Επαναληπτικών και Προαιρετικών Δομών της Γραμματικής:**
    - **Διαχείριση Επανάληψης:** Όταν ένας κανόνας περιλαμβάνει ένα στοιχείο που μπορεί να επαναληφθεί μηδέν ή περισσότερες φορές υλοποιούνται συνήθως με βρόχους while. Ο βρόχος συνεχίζει να εκτελείται και να καταναλώνει τα επαναλαμβανόμενα στοιχεία (π.χ., το κόμμα και το επόμενο ID).

- **Προαιρετικών Στοιχείων:** Όταν ένα τμήμα ενός κανόνα είναι προαιρετικό (π.χ., το [αλλιώς statement_sequence] στην εντολή if, ή το [με_βήμα expression] στην εντολή for), αυτό υλοποιείται με μια δομή if. Ο αναλυτής ελέγχει αν το τρέχον token αντιστοιχεί στην αρχή του προαιρετικού τμήματος. Εάν ναι, τότε εκτελείται ο κώδικας που αναλύει αυτό το τμήμα. Εάν όχι, το τμήμα παραλείπεται και η ανάλυση συνεχίζεται με το επόμενο μέρος του κανόνα, χωρίς να προκληθεί σφάλμα, καθώς το στοιχείο ήταν προαιρετικό.

- **Δημιουργία Abstract Syntax Tree:** Καθώς ο αναλυτής αναγνωρίζει τις διάφορες συντακτικές δομές, κατασκευάζει κόμβους ενός Αφηρημένου Συντακτικού Δέντρου. Κάθε κόμβος στο AST αντιπροσωπεύει μια συντακτική κατασκευή (π.χ., 'PROGRAM', 'ASSIGNMENT', 'IF', 'FUNCTION_CALL') και περιέχει ως παιδιά τους κόμβους που αντιστοιχούν στα συστατικά της. Για παράδειγμα, ένας κόμβος 'IF' θα έχει παιδιά για τη συνθήκη, το then block και το (προαιρετικό) else block. Αυτό το AST χρησιμοποιείται από την επόμενη φάση, την παραγωγή ενδιάμεσου κώδικα.

Αλληλεπίδραση με τον Πίνακα Συμβόλων:
Κατά τη συντακτική ανάλυση, και ειδικά κατά την επεξεργασία δηλώσεων (μεταβλητών, συναρτήσεων, παραμέτρων), ο parser αλληλεπιδρά στενά με τον Πίνακα Συμβόλων:

- **Εισαγωγή Συμβόλων:** Όταν συναντάται μια δήλωση, οι πληροφορίες για το νέο σύμβολο (π.χ., όνομα, τύπος, είδος οντότητας) εισάγονται στον Πίνακα Συμβόλων στο τρέχον scope.
- **Άνοιγμα/Κλείσιμο Scope:** Ο parser διαχειρίζεται το άνοιγμα νέων scopes (π.χ., κατά την είσοδο σε μια συνάρτηση ή διαδικασία) και το κλείσιμό τους (κατά την έξοδο).
- **Έλεγχοι Ορθότητας (Σημασιολογικοί):** Ο parser, σε συνεργασία με τον Πίνακα Συμβόλων, εκτελεί βασικούς σημασιολογικούς ελέγχους, όπως:
    - Έλεγχος για διπλή δήλωση αναγνωριστικού στο ίδιο scope.

- Έλεγχος ότι ένα αναγνωριστικό έχει δηλωθεί πριν από τη χρήση του σε μια έκφραση ή ανάθεση (μέσω της μεθόδου lookup του Πίνακα Συμβόλων).
- Έλεγχος της ορθότητας των παραμέτρων κατά την κλήση συναρτήσεων/διαδικασιών (αριθμός και τρόπος περάσματος, π.χ. χρήση '%' για REF).

Παράδειγμα AST:

```
--- AST ---
PROGRAM
  αρνητικο
  ['x']
  []
  ('ASSIGNMENT', 'x', [('OPERATOR', '-', 4), ('INTEGER', 3, 4)])
```

3.3. Διαχείριση Συντακτικών Σφαλμάτων

Όταν ο συντακτικός αναλυτής συναντήσει μια ακολουθία λεκτικών μονάδων που δεν ταιριάζει με κανέναν έγκυρο κανόνα της γραμματικής (με βάση το τρέχον token και το lookahead), προκαλείται συντακτικό σφάλμα. Η μέθοδος raise_error χρησιμοποιείται για να σηματοδοτήσει το σφάλμα, παρέχοντας ένα μήνυμα που περιγράφει το πρόβλημα, το token που προκάλεσε το σφάλμα και τον αριθμό γραμμής.

Ο υλοποιημένος αναλυτής ακολουθεί μια απλή στρατηγική διαχείρισης σφαλμάτων: με την ανίχνευση του πρώτου συντακτικού σφάλματος, η διαδικασία μεταγλώττισης τερματίζεται. Δεν επιχειρείται ανάκαμψη από το σφάλμα (error recovery) για τη συνέχιση της ανάλυσης, καθώς αυτό θα αύξανε σημαντικά την πολυπλοκότητα του parser.

4. Πίνακας Συμβόλων (Symbol Table)

Ο πίνακας συμβόλων αποτελεί ένα από τα πιο κρίσιμα υποσυστήματα ενός μεταγλωττιστή. Στην υλοποίηση του μεταγλωττιστή για τη γλώσσα **Greek++**, ο πίνακας συμβόλων αναλαμβάνει να αποθηκεύσει και να διαχειριστεί όλες τις πληροφορίες που σχετίζονται με τις δηλώσεις μεταβλητών, συναρτήσεων, παραμέτρων και προσωρινών μεταβλητών, καθώς και τη δομή των scopes (εμβέλεια).

Η αντίστοιχη υλοποίηση στον κώδικά μας περιλαμβάνει:

- Τη δημιουργία και διαχείριση (scopes), για την υποστήριξη φωλιασμένων συναρτήσεων/διαδικασιών.
- Την αποθήκευση για κάθε entity (π.χ. μεταβλητή, συνάρτηση, παράμετρος) πληροφοριών όπως: όνομα, τύπος, offset στη στοίβα, τρόπος περάσματος παραμέτρου, κ.ά.
- Την αποθήκευση για κάθε οντότητα (entity) – όπως μεταβλητή, συνάρτηση, διαδικασία, παράμετρος ή προσωρινή μεταβλητή – ενός συνόλου κρίσιμων πληροφοριών. Αυτές περιλαμβάνουν τουλάχιστον το όνομα, τον τύπο της οντότητας (entry_type), το επίπεδο εμβέλειας (scope_level), τον τύπο δεδομένων (π.χ.,'integer'), το offset της στη στοίβα για την αντίστοιχη περιοχή δεδομένων του πλαισίου ενεργοποίησης, και το μέγεθός της (size). Για συναρτήσεις και διαδικασίες, καταγράφονται επιπλέον οι παράμετροί τους (με τον τρόπο περάσματος 'CV'/'REF' και τη σειρά τους) και η αρχική ετικέτα του κώδικά τους (start_quad). Μια ειδική σημαία (is_return_var) χρησιμοποιείται για την αναγνώριση της μεταβλητής που αντιστοιχεί στην τιμή επιστροφής μιας συνάρτησης.Τη δυνατότητα αναζήτησης μεταβλητών με βάση το όνομά τους, ξεκινώντας από το πιο εσωτερικό scope και προχωρώντας προς τα έξω.
- Τη διαγραφή scopes και την τελική απόδοση του απαιτούμενου μεγέθους για το εγγράφημα δραστηριοποίησης κάθε block κώδικα.

Η χρήση του πίνακα συμβόλων εξασφαλίζει ότι η γλώσσα Greek++ μπορεί να ελέγχει τη σωστή δήλωση και χρήση των identifiers, να διαχειρίζεται σωστά την περιοχή μνήμης κάθε scope, και να υποστηρίζει λειτουργίες όπως οι παράμετροι με αναφορά ή τιμή. Όλα τα παραπάνω είναι απαραίτητα για τη δημιουργία σωστού ενδιάμεσου και τελικού κώδικα.

Παράδειγμα:

```
--- Symbol Table State AFTER PARSING ---
--- Scope Level 0 (Parser View) ---
{ 'x': { 'data_type': 'integer',
         'entry_type': 'variable',
         'name': 'x',
         'offset': 12,
         'scope_level': 0,
         'size': 4},
  'αρνητικο': { 'entry_type': 'program',
                'frame_offsets_info': {0: 16},
                'name': 'αρνητικο',
                'scope_level': 0}}

--- Frame Offset Tracking AFTER PARSING ---
{0: 16}


--- Symbol Table State AFTER ICG (includes temporaries) ---
--- Scope Level 0 (ICG View) ---
{ 't@1': { 'data_type': 'integer',
           'entry_type': 'temp_variable',
           'name': 't@1',
           'offset': 16,
           'scope_level': 0,
           'size': 4},
  'x': { 'data_type': 'integer',
         'entry_type': 'variable',
         'name': 'x',
         'offset': 12,
         'scope_level': 0,
         'size': 4},
  'αρνητικο': { 'entry_type': 'program',
                'frame_offsets_info': {0: 20},
                'name': 'αρνητικο',
                'scope_level': 0}}
```

4.1. Ανάλυση της δομής του κώδικα

Δημιουργία Πίνακα Συμβόλων

Ο πίνακας συμβόλων ξεκινάει με ένα κενό global scope. Για κάθε νέο scope (π.χ. συνάρτηση), δημιουργείται νέο λεξικό, και το offset ξεκινά από τα 12 bytes (όπως στην θεωρία του εγγραφήματος δραστηριοποίησης).(init)

Άνοιγμα Νέου Scope

Όταν ξεκινάμε νέα συνάρτηση ή διαδικασία, ανοίγεται νέο scope. Αρχικοποιείται νέος πίνακας για τα σύμβολα αυτού του επιπέδου και καθορίζεται offset από το οποίο ξεκινούν οι τοπικές μεταβλητές/παράμετροι.(open_scope)

Κλείσιμο Scope – Υπολογισμός Frame Size

Όταν ολοκληρώνεται η ανάλυση μιας συνάρτησης ή διαδικασίας, η μέθοδος close_scope() υπολογίζει το συνολικό μέγεθος του χώρου που απαιτείται στο πλαίσιο ενεργοποίησης για τις παραμέτρους, τις τοπικές και τις προσωρινές μεταβλητές του συγκεκριμένου scope. Αυτός ο υπολογισμός βασίζεται στο τελικό offset που έχει καταγραφεί για το scope αυτό (στο εσωτερικό λεξικό frame_offsets), μετρώντας από ένα βασικό offset έναρξης (ST_OFFSET_BASE = 12 bytes). Το υπολογιζόμενο μέγεθος επιστρέφεται και είναι απαραίτητο για τη σωστή δέσμευση χώρου στη στοίβα από την Παραγωγή Τελικού Κώδικα.(close_scope)

Εισαγωγή Νέου Entity (insert)

Η insert() προσθέτει νέες μεταβλητές, παραμέτρους ή προσωρινές μεταβλητές στον πίνακα συμβόλων και τους αποδίδει offset στον χώρο μνήμης του αντίστοιχου scope. Για functions/procedures αποθηκεύει επιπλέον πληροφορίες (π.χ. parameters, start_quad).(insert)

Αναζήτηση Συμβόλου (lookup)

Η lookup() βρίσκει το σύμβολο που αντιστοιχεί σε ένα όνομα (π.χ. μεταβλητή) ξεκινώντας από το εσωτερικό scope προς τα έξω — υποστηρίζοντας σωστή ορατότητα (visibility) σύμφωνα με τη θεωρία.(lookup)

Παράδειγμα χρήσης – Δήλωση παραμέτρων συνάρτησης

Κατά την ανάλυση παραμέτρων συνάρτησης, καλείται insert() για κάθε παράμετρο ώστε να καταχωρηθεί ως parameter στο ενεργό scope. Περιλαμβάνει τον τρόπο περάσματος (CV ή REF) και τη σειρά.

Παράδειγμα χρήσης – Προσωρινές μεταβλητές

Δημιουργούνται προσωρινές μεταβλητές κατά τη διάρκεια της παραγωγής τετράδων (ενδιάμεσος κώδικας). Αυτές καταχωρούνται στον πίνακα συμβόλων ώστε να υπολογιστεί σωστά ο απαιτούμενος χώρος μνήμης.(new_temp)

5.Ενδιάμεσος κώδικας

Κατά την υλοποίηση του ενδιάμεσου κώδικα, δημιουργήσαμε την κλάση IntermediateCodeGenerator, η οποία παράγει τετράδες (quadruples) για κάθε σύνθετη εντολή της γλώσσας Greek++. Η βασική λειτουργία emit() δημιουργεί μία τετράδα με έναν τελεστή και έως τρία τελοούμενα και την αριθμεί αυτόματα. Οι προσωρινές μεταβλητές που απαιτούνται σε αριθμητικές και λογικές εκφράσεις δημιουργούνται με την new_temp(), η οποία διασφαλίζει μοναδικότητα και τις καταχωρεί στον πίνακα συμβόλων.

Για τη διαχείριση ροής εκτέλεσης (όπως συνθήκες if, επαναλήψεις while, κλπ.), χρησιμοποιούμε τεχνικές **backpatching**, μέσω των makelist(), mergelist() και backpatch(), οι οποίες δημιουργούν και ενημερώνουν λίστες με ετικέτες τετράδων για άλματα που θα καθοριστούν αργότερα.

Με αυτόν τον τρόπο, ο ενδιάμεσος κώδικας που παράγεται είναι **πλήρως ανεξάρτητος από την αρχιτεκτονική υλοποίησης** και **διευκολύνει τη μετάβαση σε τελικό κώδικα**, ενώ μας παρέχει μια ενδιάμεση αναπαράσταση που μπορούμε να ελέγξουμε, να τροποποιήσουμε ή να βελτιστοποιήσουμε

Η επεξεργασία εκφράσεων στον ενδιάμεσο κώδικα πραγματοποιείται με χρήση **recursive descent parser** και μηχανισμό προτεραιότητας τελεστών. Η μέθοδος **evaluate_expression()** αναλαμβάνει να αναλύσει τις εκφράσεις και να παραγάγει τις κατάλληλες τετράδες. Καλεί τη **_parse_expr_prec()** για να διαχειριστεί τελεστές όπως +, *, <, και, ή και φροντίζει ώστε να δημιουργηθούν **ενδιάμεσα προσωρινά αποτελέσματα** όταν χρειάζεται.

Οι λογικές εκφράσεις αναπαρίστανται με την τεχνική **των truelist/falselist** και υλοποιούνται με **backpatching**. Η μέθοδος **_ensure_boolean()** μετατρέπει απλές αριθμητικές εκφράσεις σε λογικές όπου απαιτείται. Οι κλήσεις συναρτήσεων εντός εκφράσεων υποστηρίζονται πλήρως με παραγωγή **par**, **call** και **par RET.**

Η προσέγγιση αυτή μας επιτρέπει να εκφράσουμε με ακρίβεια πολύπλοκες αριθμητικές και λογικές εκφράσεις στον ενδιάμεσο κώδικα με **σαφήνεια, ευελιξία και σωστή διαχείριση τιμών και συνθηκών**.

5.1. Ανάλυση της δομής του κώδικα

Εκπομπή Τετράδας (emit)

Η emit παράγει μια τετράδα στη μορφή (op, arg1, arg2, result) και τη προσθέτει στη λίστα quads. Κάθε τετράδα αριθμείται αυτόματα. Αυτή είναι η βασική λειτουργία παραγωγής ενδιάμεσου κώδικα.

Αρίθμηση Τετράδων (nextquad)

Επιστρέφει τον αριθμό της επόμενης τετράδας που θα παραχθεί. Είναι χρήσιμο για έλεγχο ροής, δημιουργία jumps και για το **backpatching**.

Δημιουργία Προσωρινών Μεταβλητών (new_temp)

Παράγει μοναδικά ονόματα προσωρινών μεταβλητών (π.χ. t@1, t@2, κ.ο.κ.) και τις καταχωρεί στον πίνακα συμβόλων. Αυτές οι μεταβλητές αποθηκεύουν ενδιάμεσα αποτελέσματα υπολογισμών.

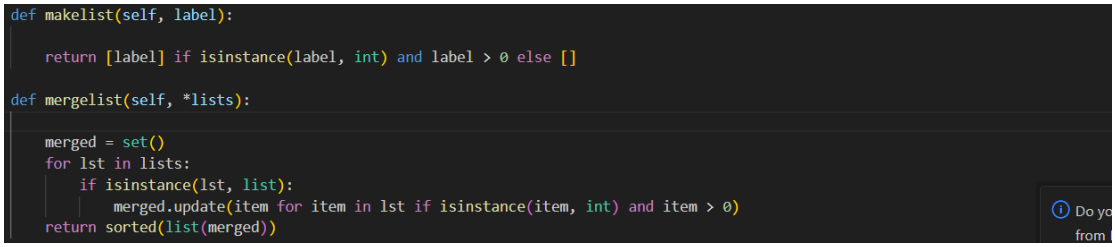
Υπορουτίνες Backpatching

makelist

Χρησιμοποιείται για δημιουργία λίστας με έναν δείκτη τετράδας, χρήσιμη σε λογικές εκφράσεις.

Mergelist

Συγχωνεύει λίστες από labels τετράδων, π.χ. σε or/and συνθήκες



Backpatch

Συμπληρώνει το result των τετράδων που έχουν “τρύπα” (placeholder ? ή _), π.χ. άλματα σε if, while όταν γνωρίζουμε την τελική ετικέτα.

evaluate_expression(expr_tokens)

Κύρια μέθοδος που καλείται από το parser για να αξιολογήσει εκφράσεις και να παραγάγει τις αντίστοιχες τετράδες.

- Αν η έκφραση είναι απλή: καλεί _init_expr_parser() και _parse_expr_prec()
- Αν είναι standalone κλήση συνάρτησης: παράγει par, call, par ret, αποθηκεύει σε temp

_parse_expr_prec(min_precedence)

Υλοποιεί **προτεραιότητα τελεστών** με τεχνική **precedence climbing**. Διαχειρίζεται όλους τους τελεστές (αρ. + λογικούς + συγκριτικούς).

Ενδεικτικό παράδειγμα:

- x + y * z → παράγει * y z t1, + x t1 t2
- x > y και z < 5 → παράγει:
    - x > y ? → truelist1, falselist1
    - backpatching + merge με z < 5 → truelist, falselist

_parse_atom()

Αναλύει **στοιχειώδεις μονάδες** έκφρασης:

- αριθμοί, μεταβλητές
- εμφωλευμένες εκφράσεις με ()
- **εμφωλευμένες κλήσεις συναρτήσεων** (σημαντικό!)
- unary operators: -, +, όχι

Αν αναγνωρίσει συνάρτηση: παράγει τις par, call, par RET, και αποδίδει temp ως result.

_place_from_eval_result()

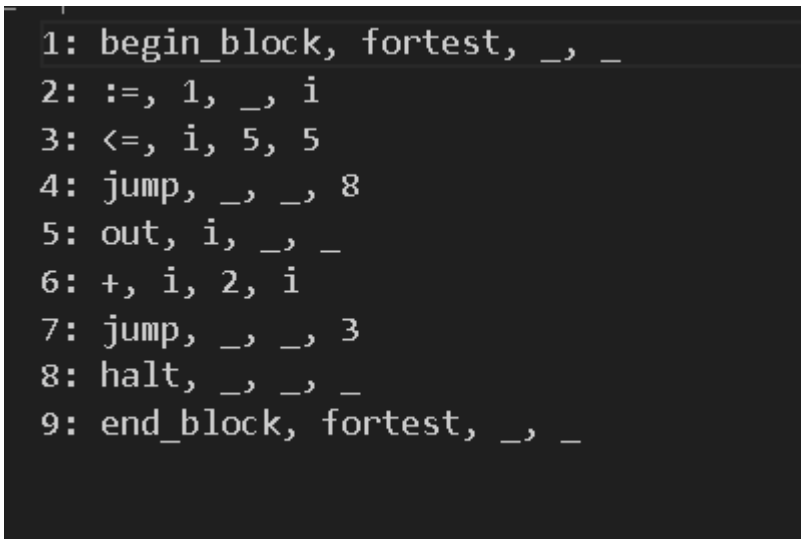
Επιστρέφει place από αποτέλεσμα έκφρασης. Αν είναι boolean (truelist/falselist), δημιουργεί τετράδες που αποθηκεύουν 1 ή 0 σε νέο temp.

Ειδικά για περιπτώσεις όπως: x := (a > b) + 3

_ensure_boolean()

Εξασφαλίζει ότι ένα αποτέλεσμα είναι boolean. Αν έχει μόνο place, δημιουργεί != place 0 ? για να παράγει truelist/falselist.

Παράδειγμα for:



6.Παραγωγή Τελικού Κώδικα

Στο τελευταίο στάδιο του μεταγλωττιστή Greek++ πραγματοποιείται η παραγωγή τελικού κώδικα για την αρχιτεκτονική RISC-V. Σε αυτό το στάδιο, οι τετράδες του ενδιάμεσου κώδικα μεταφράζονται σε αντίστοιχες εντολές μηχανής, επιτυγχάνοντας έτσι τη μετάβαση από μια ενδιάμεση αναπαράσταση σε χαμηλού επιπέδου κώδικα έτοιμο για εκτέλεση ή προσομοίωση.

Η υλοποίηση βασίζεται στην κλάση **FinalCodeGenerator**, η οποία αναλαμβάνει:

- την ερμηνεία και επεξεργασία κάθε τετράδας του ενδιάμεσου κώδικα,
- την παραγωγή εντολών σε μορφή RISC-V όπως**: li, mv, lw, sw, add, sub, beq, bne, jal, jr, ecall** κ.ά.,
- τη διαχείριση των παραμέτρων και της επιστροφής τιμών στις συναρτήσεις,
- τη σωστή τοποθέτηση μεταβλητών στη μνήμη μέσω των βοηθητικών συναρτήσεων **loadvr**, **storerv** και **gnvlcode**,
- την τήρηση του πρωτοκόλλου stack frame, αξιοποιώντας καταχωρητές όπως **zero**,**sp,s0, gp, ra, a0** και **a7**, και **t0** έως **t3**.

Η παραγωγή του τελικού κώδικα υλοποιείται μέσω βασικών μεθόδων, όπως:

- **generate()** για την επεξεργασία όλων των τετράδων και την παραγωγή των αντίστοιχων RISC-V εντολών,
- **loadvr()** και **storerv()** για φόρτωση και αποθήκευση τιμών από/προς τη μνήμη,
- **gnvlcode()** για προσπέλαση μη τοπικών μεταβλητών σε φωλιασμένες συναρτήσεις,
- επιμέρους μεθόδους όπως **handle_arithmetic(), handle_branch(), handle_function_call()** για την εξειδικευμένη μετάφραση αριθμητικών πράξεων, συνθηκών και κλήσεων.

Η έξοδος είναι μία μορφή **assembly-like** κώδικα, κατάλληλη για εισαγωγή σε RISC-V assembler ή προσομοιωτή.

```
.data

.text
.globl main
main:
    j L_arnitiko_entry                  # Initial jump to αρνητικο entry
LQ1:
# Quad 1: begin_block, αρνητικο, _, _
L_arnitiko_entry:
    addi sp, sp, -8                     # allocate frame for main (8 bytes)
    mv gp, sp                           # set Global Pointer (GP = main frame base)
    mv s0, sp                           # set Frame Pointer S0 for main
LQ2:
# Quad 2: -, 0, 3, t@1
    li t0, 0                            # load literal value 0
    li t1, 3                            # load literal value 3
    sub t2, t0, t1                      # t@1 = 0 - 3
    addi t3, gp, -16                    # addr of global t@1 (-16 from gp)
    sw t2, 0(t3)                        # store value to t@1
LQ3:
# Quad 3: :=, t@1, _, x
    addi t3, gp, -16                    # addr of global t@1 (-16 from gp)
    lw t0, 0(t3)                        # load value for t@1
    addi t3, gp, -12                    # addr of global x (-12 from gp)
    sw t0, 0(t3)                        # store value to x
LQ4:
# Quad 4: halt, _, _, _
    li a0, 0                            # exit code 0
    li a7, 93                           # syscall: exit
    ecall
LQ5:
# Quad 5: end_block, αρνητικο, _, _
    addi sp, sp, 8                      # deallocate main's frame (8 bytes)
```

(Σημείωση: Η αρχιτεκτονική RISC-V ορίζει περισσότερους προσωρινούς καταχωριτές (t0-t6) και καταχωριτές ορισμάτων (a0-a7). Στην παρούσα υλοποίηση, έγινε επιλεκτική χρήση αυτών που κρίθηκαν απαραίτητοι για τις ανάγκες του μεταγλωττιστή της Greek++.)

6.1. Ανάλυση της δομής του κώδικα

1. generate(self, quads)

Σκοπός: Κεντρική μέθοδος που διατρέχει όλες τις τετράδες και παράγει ισοδύναμες εντολές RISC-V.

- Καλεί βοηθητικές μεθόδους όπως handle_arithmetic, handle_assignment, handle_branch, handle_function_call, κ.ά.
- Εντοπίζει το είδος κάθε τετράδας με βάση το quad['op'] και δημιουργεί τις αντίστοιχες εντολές στο self.final_code.

2. loadvr(v, r)

Σκοπός: Μεταφέρει την τιμή της μεταβλητής v στον καταχωρητή r.

Περιλαμβάνει πολλαπλές περιπτώσεις:

- Αν v είναι σταθερά: li r, v
- Αν είναι global μεταβλητή: lw r, -offset(gp)
- Αν είναι τοπική/προσωρινή: lw r, -offset(sp)
- Αν είναι παραμετρος με αναφορά: lw t0, -offset(sp) → lw r, (t0)
- Αν είναι μη τοπική: gnvlcode(v) → lw r, (t0)

3. storerv(r, v)

Σκοπός: Αποθηκεύει την τιμή του καταχωρητή r στη μεταβλητή v.

Ανάλογες περιπτώσεις με τη loadvr, αλλά για sw:

- sw r, -offset(sp) για τοπικές/temps
- sw r, -offset(gp) για global
- gnvlcode(v) → sw r, (t0) για μη τοπικές
- Αν v είναι reference: lw t0, -offset(sp) → sw r, (t0)

4. gnvlcode(v)

Σκοπός: Εντοπίζει την **διεύθυνση μιας μη τοπικής μεταβλητής** και την τοποθετεί στον t0.

Ακολουθεί τους συνδέσμους πρόσβασης (-4(sp), -4(t0), ...) τόσες φορές όσο το βάθος φωλιάσματος, και τελικά:

Riscv:

```
addi t0, t0, -offset
```

---

5. handle_arithmetic(quad)

Σκοπός: Δημιουργεί εντολές RISC-V για αριθμητικές τετράδες όπως +, -, *, /.

Παράδειγμα:

- Για +, x, y, z:

Riscv:

```
loadvr x, t1
loadvr y, t2
add t1, t1, t2
storerv t1, z
```

---

6. handle_assignment(quad)

Σκοπός: Μεταφράζει τετράδες ανάθεσης :=.

Παράδειγμα:

Riscv:

```
loadvr x, t1
storerv t1, y
```

---

7. handle_branch(quad)

Σκοπός: Χειρίζεται τετράδες συνθηκών (<, >, =, !=, <=, >=) και jump.

Παράδειγμα:

Riscv:

loadvr x, t1

loadvr y, t2

beq t1, t2, Lz

ή απλό άλμα:

Riscv:

b Lz

8. handle_function_call(quad)

Σκοπός: Παράγει εντολές για call, par, ret, begin_block, end_block.

Παραδείγματα:

- Για par x, CV, _:

  loadvr x, t0

  sw t0, -offset(fp)

- Για call f, _, _:

Riscv:

addi sp, sp, framelength

jal f

addi sp, sp, -framelength

Για retv _, _, x:

loadvr x, t1

lw t0, -8(sp)

sw t1, (t0)

9. get_final_code()

Σκοπός: Επιστρέφει τη λίστα εντολών RISC-V που έχουν παραχθεί.

7.Συμπέρασμα

Η ανάπτυξη του μεταγλωττιστή για τη γλώσσα Greek++ κάλυψε όλα τα βασικά στάδια που απαιτούνται για τη μετάφραση ενός υψηλού επιπέδου προγράμματος σε τελικό εκτελέσιμο κώδικα. Ξεκινώντας από τη λεκτική και συντακτική ανάλυση, προχωρήσαμε στη σημασιολογική επεξεργασία με τη χρήση πίνακα συμβόλων, καταγράφοντας πληροφορίες για μεταβλητές, παραμέτρους και συναρτήσεις, ανά επίπεδο εμβέλειας. Στη συνέχεια, παράχθηκε ενδιάμεσος κώδικας με τετράδες, αξιοποιώντας τεχνικές όπως backpatching για την υλοποίηση συνθηκών και ελέγχου ροής, ενώ στην τελική φάση παράχθηκε χαμηλού επιπέδου κώδικας RISC-V που μπορεί να εκτελεστεί σε προσομοιωτή ή πραγματικό σύστημα.

Η υλοποίηση αυτή ανέδειξε τόσο τις θεωρητικές αρχές της μεταγλώττισης όσο και τις πρακτικές δυσκολίες στη σωστή διαχείριση της μνήμης, της στοίβας και των scopes. Μέσω του modular σχεδιασμού, επιτεύχθηκε σαφής διαχωρισμός ευθυνών ανάμεσα στα επιμέρους υποσυστήματα (parser, intermediate code, final code), γεγονός που επέτρεψε την ευκολότερη επέκταση, δοκιμή και συντήρηση του κώδικα. Συνολικά, το project αυτό αποτέλεσε μια πλήρη και πρακτική εφαρμογή των εννοιών που διδάσκονται στο μάθημα των μεταφραστών, παρέχοντας μια λειτουργική υλοποίηση μεταγλωττιστή που μεταφράζει μια ελληνόφωνη γλώσσα σε κώδικα RISC-V.