Greek++ Compiler

Χρήστος Δημητρέσης
4351
cs04351@uoi.gr

Μάιος 2025

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Ο σκοπός της παρούσας αναφοράς είναι να παρουσιάσει τη διαδικασία υλοποίησης του μεταγλωττιστή (compiler) για τη γλώσσα Greek++

Συγκεκριμένα, στόχος είναι να αναδειχθούν οι σχεδιαστικές επιλογές και οι προγραμματιστικές προσεγγίσεις που υιοθετήθηκαν, δίνοντας έμφαση στον τρόπο που δομήθηκε και οργανώθηκε η υλοποίηση των επιμέρους τμημάτων.

Η παρουσίαση χωρίζεται σε δύο βασικά τμήματα: το εμπρόσθιο τμήμα (frontend) και το οπίσθιο τμήμα (backend). Σε κάθε ένα από αυτά αναλύονται οι κλάσεις που χρησιμοποιήθηκαν, οι μεταξύ τους αλληλεπιδράσεις και η λογική πίσω από τη σχεδίαση και την υλοποίησή τους. Επιπλέον, δίνεται έμφαση στον τρόπο χειρισμού κρίσιμων ζητημάτων, όπως η διαχείριση του πίνακα συμβόλων, η παραγωγή ενδιάμεσου κώδικα και η μετάφραση σε κώδικα μηχανής RISC-V.

Η αναφορά αυτή είναι οδηγός για την κατανόηση των τεχνικών και των αποφάσεων που εφαρμόστηκαν κατά την κατασκευή του μεταγλωττιστή και όχι να επαναλάβει γνωστές θεωρητικές έννοιες ή διαδικασίες που αναφέρονται εκτενώς στο μάθημα.

Κεφάλαιο 2

Ροή Εκτέλεσης (GreekPP.java)

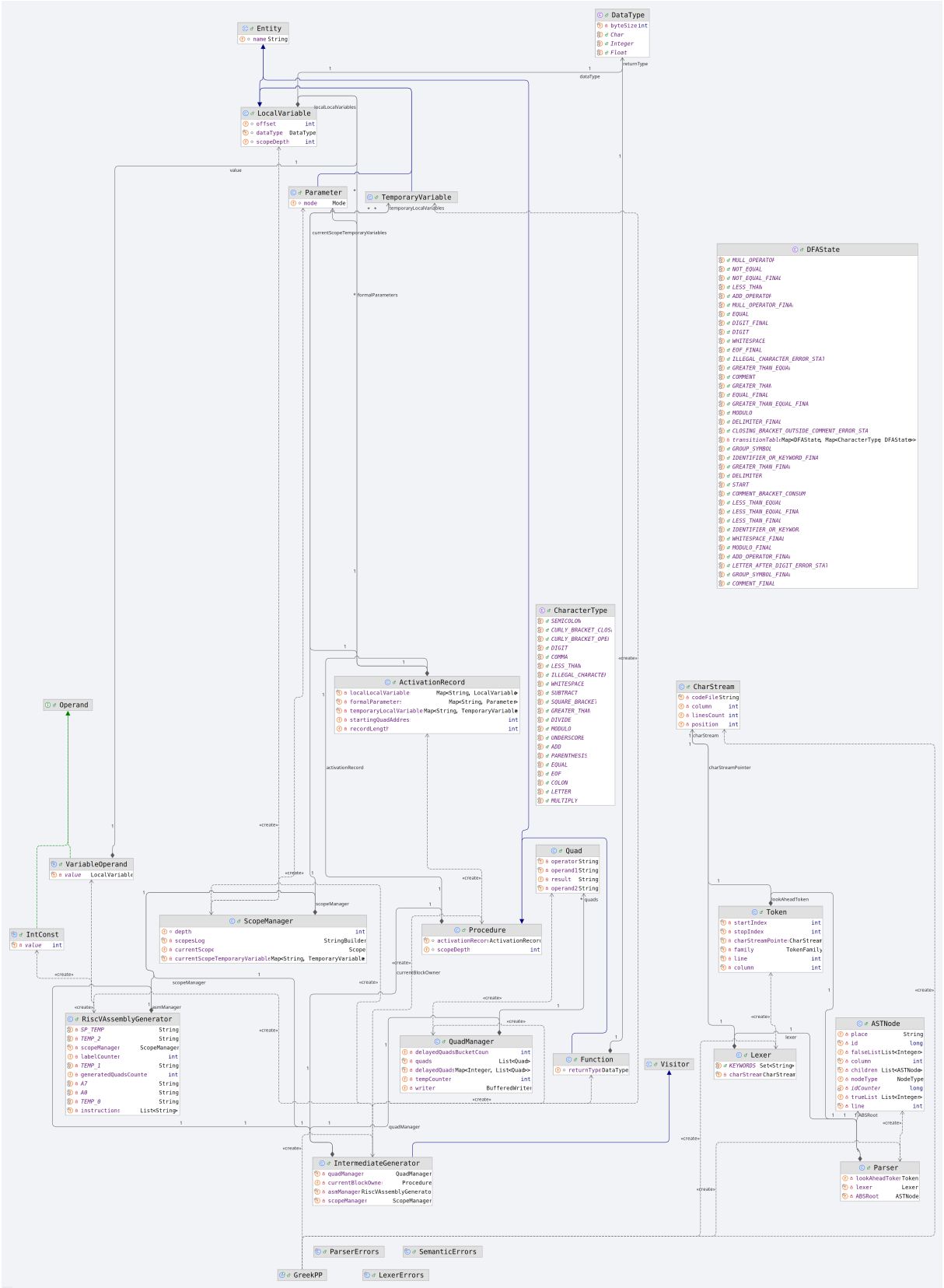
Η κλάση `GreekPP` περιέχει τη μέθοδο `main` και αποτελεί το αρχικό σημείο εκτέλεσης του μεταγλωττιστή. Η ροή της εκτέλεσης ακολουθεί τη λογική των βασικών φάσεων ενός compiler, με αυστηρά σειριακή δομή και διακριτά στάδια, όπως η λεκτική και συντακτική ανάλυση, η παραγωγή ενδιάμεσης αναπαράστασης, ο σημασιολογικός έλεγχος και η παραγωγή τελικού κώδικα.

Η εκτέλεση ξεκινά με τον έλεγχο των ορισμάτων της εντολής. Αν δεν δοθεί αρχείο εισόδου, η εκτέλεση διακόπτεται με κατάλληλο μήνυμα σφάλματος.

- **1ο Βήμα – Λεκτική Ανάλυση**: Δημιουργείται αντικείμενο τύπου `CharStream` που διαβάζει το αρχείο εισόδου, και αυτό περνά στον `Lexer`. Ο `Lexer` μετατρέπει τη ροή χαρακτήρων σε λεκτικές μονάδες (tokens), οι οποίες παρέχονται στον `Parser` διαδοχικά, μία κάθε φορά, όταν ζητηθούν.
- **2ο Βήμα – Συντακτική Ανάλυση & Δημιουργία Συντακτικού Δέντρου**: Ο `Parser` χρησιμοποιεί τον `Lexer` για να παραλάβει τα tokens και να κατασκευάσει το αφηρημένο συντακτικό δέντρο (AST), μέσω του `getABSRoot()`. Η ανάλυση γίνεται με καθοδηγούμενες μεταβάσεις βάσει της γραμματικής, και το δέντρο που παράγεται αναπαριστά τη συντακτική δομή του προγράμματος.
- **3ο Βήμα [Μέρος 1] – Ενδιάμεση Αναπαράσταση & Συμβολική Ανάλυση**: Ο `IntermediateGenerator` επισκέπτεται το αφηρημένο συντακτικό δέντρο και παράγει την ενδιάμεση μορφή του προγράμματος σε τετράδες (quads). Παράλληλα, δημιουργεί και γεμίζει τον πίνακα συμβόλων με τις δηλώσεις μεταβλητών, παραμέτρων και υποπρογραμμάτων. Αν και δεν εκτελεί πλήρη σημασιολογικό έλεγχο, ενσωματώνει βασικούς ελέγχους και προετοιμάζει τα δεδομένα για την παραγωγή τελικού κώδικα.

- **3ο Βήμα [Μέρος 2] – Παραγωγή Τελικού Κώδικα**: Η παραγωγή του τελικού αρχείου σε RISC-V Assembly γίνεται από τον `AsmManager`, ο οποίος καλείται από τον `IntermediateGenerator`. Ο τελικός κώδικας δεν παράγεται συγκεντρωτικά στο τέλος, αλλά παράγεται σταδιακά κατά την αναδρομική επίσκεψη του συντακτικού δέντρου. Συγκεκριμένα, κάθε φορά που ολοκληρώνεται η επεξεργασία ενός scope, παράγονται και αποθηκεύονται οι αντίστοιχες assemply εντολές που το αφορούν.

Η συνολική αρχιτεκτονική του μεταγλωττιστή ακολουθεί αντικειμενοστραφή (object-oriented) προσέγγιση, με στόχο τη σαφή διάκριση των φάσεων και τη χαμηλή σύζευξη (low coupling) μεταξύ τους. Κάθε στάδιο, από τη λεκτική ανάλυση έως την παραγωγή τελικού κώδικα, υλοποιείται μέσω ανεξάρτητων κλάσεων όπως `Lexer`, `Parser`, `IntermediateGenerator`, `RiscVAssemblyGenerator`, οι οποίες επικοινωνούν μόνο μέσω καθορισμένων διεπαφών και αντικειμένων όπως (π.χ. `ASTNode`, `Quad`, `ScopeManager`). Η δομή αυτή προάγει τη modular σχεδίαση και διευκολύνει την επέκταση ή συντήρηση του compiler.



Σχήμα 2.1: Διαγραμματική αναπαράσταση των βασικών κλάσεων του Greek++ μεταγλωττιστή

Μέρος Ι

Front-End

Κεφάλαιο 3

Λεκτικός Αναλυτής(Lexer)

Η υλοποίηση του λεκτικού αναλυτή βασίζεται σε πέντε κύριες κλάσεις, που βρίσκονται στον φάκελο `lexer`, καθεμία με ξεχωριστό ρόλο στην ανάλυση της εισόδου σε λεκτικές μονάδες (tokens):

- `Lexer.java`: Η κεντρική κλάση που υλοποιεί τον μηχανισμό ανάλυσης χρησιμοποιώντας το πεπερασμένο αυτόματο (DFA), και παρέχει τη μέθοδο παραγωγής των tokens.
- `CharStream.java`: Υπεύθυνη για την ανάγνωση της ροής χαρακτήρων από το αρχείο εισόδου, προσφέροντας μεθόδους lookahead και κατανάλωσης χαρακτήρων.
- `CharacterType.java`: Ορίζει την κατηγοριοποίηση των χαρακτήρων εισόδου ως τύπος `enum`, η οποία χρησιμοποιείται ως είσοδος στο DFA.
- `Token.java`: Αναπαριστά τις λεκτικές μονάδες που παράγονται από τον `Lexer`, αποθηκεύοντας πληροφορίες όπως τύπο, θέση και αναγνωρισμένο αλφαριθμιτικό.
- `DFAState.java`: Περιγράφει τις καταστάσεις του πεπερασμένου αυτόματου, μαζί με τις μεταβάσεις μεταξύ αυτών.

3.1 CharStream

Η `CharStream` είναι η πρώτη κλάση που ενεργοποιείται κατά την εκτέλεση του μεταγλωττιστή, αποτελώντας την αρχική διεπαφή μεταξύ του πηγαίου αρχείου και της διαδικασίας λεκτικής ανάλυσης. Εφαρμόζεται στο πρώτο βήμα του κύριου προγράμματος, όπως φαίνεται στην κλάση `GreekPP`, και παρέχει στο `Lexer` τη δυνατότητα να προσπελάσει με ασφάλεια και ακρίβεια τους χαρακτήρες του αρχείου.

Αναλυτικότερα, η `CharStream` φορτώνει εφάπαξ το αρχείο εισόδου σε ένα `String`, επιτρέποντας την αποδοτική τυχαία πρόσβαση σε χαρακτήρες. Το `String` αυτό λειτουργεί ουσιαστικά σαν ένας μεγάλος πίνακας χαρακτήρων, από τον οποίο ο `Lexer` διαβάζει με δείκτες θέσης, χωρίς αντιγραφές ή επιπλέον κόστη. Παρέχει τη μέθοδο `peekNextChar()`, η οποία επιστρέφει τον επόμενο χαρακτήρα χωρίς να τον καταναλώνει, και τη μέθοδο `consumeNextChar()`, η οποία προχωρά τη ροή και ενημερώνει τις πληροφορίες θέσης (στήλη και γραμμή).

Επιπλέον, η κλάση παρακολουθεί την τρέχουσα θέση του αναλυτή στο κείμενο μέσω των μεθόδων `getPosition()`, `getColumn()` και `getLinesCount()`, επιτρέποντας την παραγωγή ακριβών διαγνωστικών μηνυμάτων. Παρέχει επίσης τη μέθοδο `getText(start, stop)`, η οποία χρησιμοποιείται για την εξαγωγή υποσυμβολοσειρών και την ανακατασκευή του αναγνωρισμένου `Token`.

Η `CharStream` έχει σχεδιαστεί έτσι ώστε να συνεργάζεται άμεσα με το πεπερασμένο αυτόματο (DFA) του `Lexer`, εξασφαλίζοντας ότι η είσοδος αναλύεται με ακρίβεια χαρακτήρα προς χαρακτήρα και με δυνατότητα lookahead. Αποτελεί έτσι τη βάση πάνω στην οποία οικοδομείται η λειτουργία της λεκτικής ανάλυσης.

3.2 CharacterType

Η `CharacterType` είναι μία `enum` κλάση που υλοποιεί την κατηγοριοποίηση όλων των πιθανών χαρακτήρων εισόδου, ώστε να υποστηρίζεται η καθαρή, ασφαλής και δομημένη ανάλυσή τους από τον λεκτικό αναλυτή. Προσδιορίζει τον τύπο του κάθε χαρακτήρα και αντικαθιστά τη χρήση ασαφών `if-else` ή `switch` μέσα στον ίδιο τον λεκτικό αναλυτή.

Ο ρόλος της δεν περιορίζεται στη διευκόλυνση της ανάγνωσης χαρακτήρων: η `CharacterType` χρησιμεύει ως **είσοδος του πεπερασμένου αυτόματου** (DFA) που υλοποιεί ο `Lexer`. Συγκεκριμένα, κάθε χαρακτήρας που διαβάζεται από το `CharStream` μετατρέπεται σε `CharacterType`, το οποίο περνά στη μέθοδο `getNextState()` της τρέχουσας κατάστασης του DFA. Έτσι, η ίδια η υλοποίηση του λεκτικού αναλυτή βασίζεται στον ορισμό του αλ-

φαβήτου του DFA ως σύνολο τύπων χαρακτήρων αντί για μεμονωμένους χαρακτήρες.

Η χρήση αυτής της διεπαφής μεταξύ χαρακτήρων και καταστάσεων προσφέρει τα εξής πλεονεκτήματα:

- Μειώνει το πλήθος μεταβάσεων στο αυτόματο.
- Διαχωρίζει τη χαμηλού επιπέδου ανάλυση χαρακτήρων από την υψηλού επιπέδου λογική αναγνώρισης λεκτικών μονάδων.
- Διευκολύνει τη διαχείριση εξαιρέσεων, όπως οι παράνομοι χαρακτήρες.

Οι κατηγορίες που υποστηρίζονται είναι οι εξής:

- `LETTER`: όλα τα λατινικά και ελληνικά γράμματα, με ή χωρίς τόνο.
- `DIGIT`: αριθμητικά ψηφία 0–9.
- `WHITESPACE`: χαρακτήρες όπως space, tab, newline, carriage return.
- `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `MODULO`: αριθμητικοί τελεστές.
- `PARENTHESIS`, `SQUARE_BRACKET`, `CURLY_BRACKET_OPEN`, `CURLY_BRACKET_CLOSE`: παρενθέσεις όλων των τύπων.
- `LESS_THAN`, `GREATER_THAN`, `EQUAL`, `COLON`, `SEMICOLON`, `COMMA`, `UNDERSCORE`: χαρακτήρες σύγκρισης, διαχωρισμού και σύνταξης.
- `EOF`: ειδική τιμή για την αναγνώριση τέλους εισόδου (character -1).
- `ILLEGAL_CHARACTER`: για οποιονδήποτε χαρακτήρα δεν ανήκει σε κάποια από τις παραπάνω κατηγορίες.

Η παρουσία της `ILLEGAL_CHARACTER` διασφαλίζει την ανθεκτικότητα του αναλυτή, αφού επιτρέπει την έγκαιρη ανίχνευση και αναφορά λαθών κατά την επεξεργασία χαρακτήρων που δεν είναι μέρος της γλώσσας.

3.3 Token

Token Family

Το `TokenFamily` είναι ένας τύπος `enum` που ομαδοποιεί τα tokens σε βασικές κατηγορίες, διευκολύνοντας την ταξινόμησή τους κατά τη διαδικασία ανάλυσης. Οι βασικές κατηγορίες περιλαμβάνουν:

- `KEYWORD`: Λέξεις-κλειδιά της γλώσσας.
- `ADD_OPERATOR`: Τελεστές πρόσθεσης και αφαίρεσης.
- `MUL_OPERATOR`: Τελεστές πολλαπλασιασμού, διαίρεσης κλπ.
- `REL_OPERATOR`: Σχεσιακοί τελεστές (π.χ. <, >, =).
- `REFERENCE_OPERATOR`: Τελεστές αναφοράς.
- `DELIMITER`: Διαχωριστικά (π.χ. κόμμα, άνω τελεία).
- `GROUP_SYMBOL`: Σύμβολα ομαδοποίησης (π.χ. παρενθέσεις).
- `IDENTIFIER`: Αναγνωριστικά ονόματα.
- `NUMBER`: Αριθμητικές σταθερές.
- `EOF`: Σήμα τέλους αρχείου.
- `OTHER`: Όλα τα άλλα μη κατηγοριοποιημένα tokens.

Η χρήση του `TokenFamily` βοηθά στην ομογενοποίηση της επεξεργασίας των tokens και επιτρέπει τον εύκολο διαχωρισμό των διαφόρων τύπων λεκτικών μονάδων μέσα στον compiler.

Token Class

Η κλάση `Token` αντιπροσωπεύει μία λεκτική μονάδα (token), δηλαδή ένα αναγνωρισμένο σύμβολο της γλώσσας Greek++ το οποίο έχει καταγραφεί από τον λεκτικό αναλυτή. Κάθε αντικείμενο `Token` περιλαμβάνει πληροφορίες για την κατηγορία της λεκτικής μονάδας, το ακριβές απόσπασμα του κώδικα που αναγνωρίστηκε, καθώς και τη θέση του στο αρχείο: τόσο τη γραμμή και τη στήλη, όσο και τους δείκτες αρχής και τέλους στο `CharStream`, που αντιστοιχούν στους χαρακτήρες που το αποτελούν.

Κατά τη δημιουργία ενός token, λαμβάνεται υπόψη η τελική κατάσταση του DFA, η οποία καθορίζει την κατηγορία του (`TokenFamily`), καθώς και οι δείκτες αρχής και τέλους του αποσπάσματος που αναγνωρίστηκε. Η κλάση `Token` δεν αποθηκεύει το ίδιο το string, αλλά ανακτά δυναμικά το αναγνωρισμένο κείμενο μέσω του `CharStream`, κρατώντας δείκτες αρχής και τέλους.

Η ταξινόμηση των tokens γίνεται με βάση το enum `TokenFamily`, το οποίο περιλαμβάνει κατηγορίες όπως λέξεις-κλειδιά, αναγνωριστικά, τελεστές, αριθμοί, ομαδοποιητικά σύμβολα, κ.ά. Στην περίπτωση των αλφαριθμητικών η απόδοση της κατηγορίας αποφασίζεται δυναμικά με βάση την τελική κατάστασης του DFA και του κειμένου που αναγνωρίστηκε.

3.4 DFAState

Η κλάση `DFAState` υλοποιεί τις καταστάσεις του πεπερασμένου αυτόματου (DFA) που χρησιμοποιείται στον λεκτικό αναλυτή για την αναγνώριση λεκτικών μονάδων. Υλοποιείται ως `enum`, επιτρέποντας άμεση και τυποποιημένη διαχείριση της λογικής του αυτομάτου μέσω ονομασμένων καταστάσεων, χωρίς τη χρήση αριθμητικών ή συμβολοσειρών.

3.4.1 Πίνακας Μεταβάσεων

Το σύνολο μεταβάσεων μεταξύ των καταστάσεων υλοποιείται στατικά μέσω ενός πίνακα μετάβασης (`transitionTable`), ο οποίος είναι τύπου `EnumMap<DFAState, EnumMap<CharacterType, DFAState>>`. Η επιλογή χρήσης `EnumMap` προσφέρει εξαιρετική απόδοση (O(1) προσπέλαση) και μειώνει την πιθανότητα λαθών σε σχέση με πιο γενικές δομές, όπως τα `HashMap` ή αλυσίδες `switch-case`.

Δηλαδή, για κάθε κατάσταση `from` του αυτόματου, διατηρείται ένα ξεχωριστό αντικείμενο `EnumMap<CharacterType, DFAState>`, το οποίο περιγράφει όλες τις δυνατές μεταβάσεις της μορφής `CharacterType → nextState`. Με αυτόν τον τρόπο, κάθε κατάσταση διαθέτει τον δικό της τοπικό πίνακα μεταβάσεων. Ακόμη και οι καταστάσεις σφάλματος περιλαμβάνονται στον πίνακα, ώστε όλες οι δυνατές καταστάσεις να έχουν ορισμένες μεταβάσεις, διασφαλίζοντας την πληρότητα του DFA.

Η μέθοδος `initializeTransitionTable()` γεμίζει αυτόματα τον πίνακα μεταβάσεων, καταγράφοντας για κάθε ζεύγος κατάστασης και τύπου χαρακτήρα την επόμενη κατάσταση. Η λογική καλύπτει όλους τους τύπους χαρακτήρων (`CharacterType`) και προβλέπει:

- Μετάβαση από `START` σε `DIGIT`, `IDENTIFIER_OR_KEYWORD`, `ADD_OPERATOR`, κ.ά. ανάλογα με τον χαρακτήρα εισόδου.
- Εσωτερικές μεταβάσεις που επιτρέπουν επαναλήψεις (π.χ. συνεχόμενα ψηφία ή γράμματα).
- Αναγνώριση τελικών καταστάσεων μέσω ξεχωριστών `_FINAL` καταστάσεων.
- Υποστήριξη για σύνθετους τελεστές (`<=`, `!=`, `>=`) και σχόλια με άνοιγμα/κλείσιμο αγκύλης.
- Ανίχνευση σφαλμάτων, με ειδικές καταστάσεις όπως `ILLEGAL_CHARACTER_ERROR_STATE` και `CLOSING_BRACKET_OUTSIDE_COMMENT_ERROR_STATE`.

3.4.2 Μέθοδοι

Οι μέθοδοι `getNextState(CharacterType)`, `isFinal()` και `triggersDFARestart()` αποτελούν τον βασικό μηχανισμό αλληλεπίδρασης μεταξύ του `Lexer` και του DFA. Η μέθοδος `getNextState(CharacterType)` ενημερώνει τον λεκτικό αναλυτή για την επόμενη κατάσταση του αυτόματου, βάσει του τύπου χαρακτήρα που διαβάζεται αλλά και την κατάσταση στην οποία βρίσκεται το αυτόματο την στιγμή της κλήσης. Ο τύπος αυτός καθορίζεται μέσω της `CharacterType.of(char)`, και η μετάβαση αναζητείται στον πίνακα μεταβάσεων `transitionTable`.

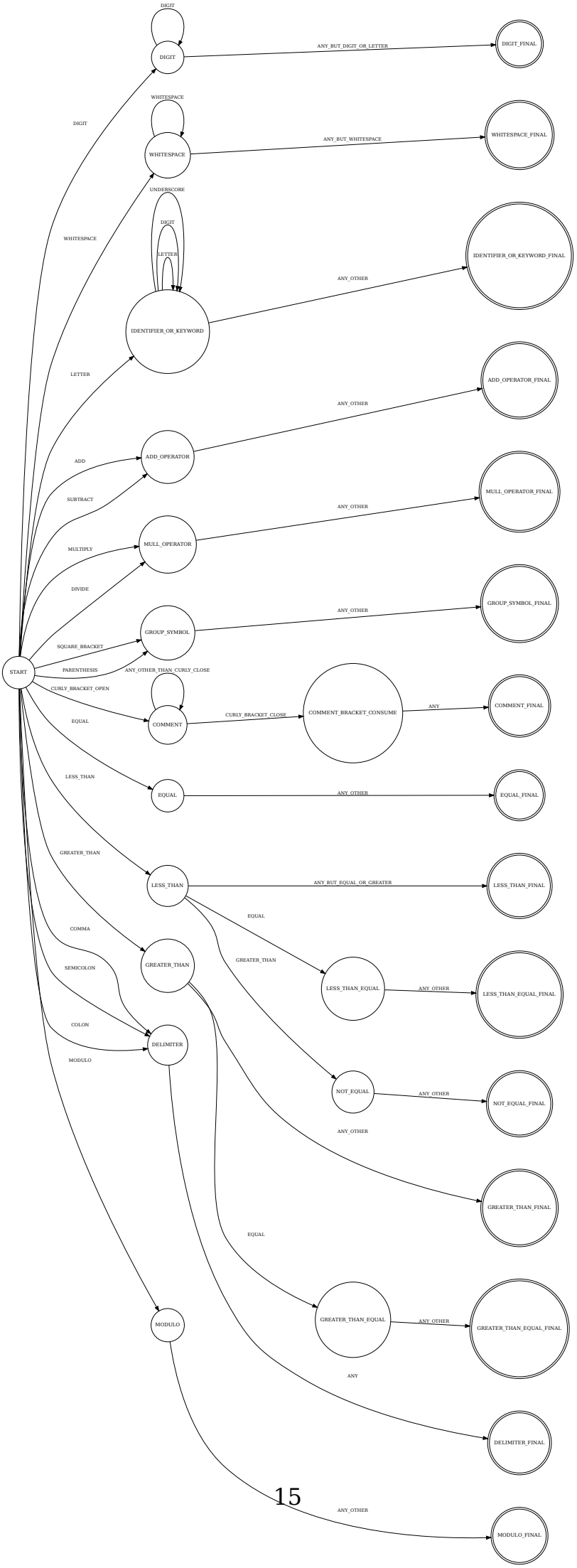
Αν σε δεδομένη κατάσταση έρθει χαρακτήρας ο όποιος δεν θα μπορούσε να οδηγήσει στην δημιουργία αποδεκτού token για την γλώσσα μας θεωρείται μη επιτρεπτή είσοδος και οδηγεί στη ρητή μετάβαση στην ειδική κατάσταση `ILLEGAL_CHARACTER_ERROR_STATE`. Με αυτόν τον τρόπο, το DFA μπορεί να εντοπίσει άμεσα κάθε μη αναγνωρίσιμο ή μη υποστηριζόμενο σύμβολο, χωρίς να απαιτούνται εξαιρέσεις ή πολύπλοκοι έλεγχοι εντός του `Lexer`.

Η μέθοδος `isFinal()` υποδεικνύει στον `Lexer` αν η τρέχουσα κατάσταση είναι τελική, δηλαδή αν έχει ολοκληρωθεί η αναγνώριση μίας λεκτικής μονάδας και μπορεί να δημιουργηθεί ένα `Token`. Οι τελικές καταστάσεις δηλώνονται ρητά με την κατάληξη `_FINAL` και αντιστοιχούν σε αναγνωρίσιμες κατηγορίες της γλώσσας.

Η μέθοδος `triggersDFARestart()` επισημαίνει ότι μία κατάσταση του DFA — αν και τελική — δεν απαιτεί την παραγωγή `Token`. Αυτό ισχύει, για παράδειγμα, σε περιπτώσεις όπως η αναγνώριση whitespace ή σχολίων. Όταν η μέθοδος επιστρέφει `true`, σηματοδοτεί ότι η λεκτική μονάδα μπορεί να αγνοηθεί και το αυτόματο πρέπει να ξεκινήσει ξανά την ανάλυση από την αρχική του κατάσταση `START`, συνεχίζοντας με τον επόμενο χαρακτήρα. Η λογική αυτή επιτρέπει τη φυσική και καθαρή φιλτραρίσματος των μη σημασιολογικών στοιχείων της γλώσσας ήδη σε επίπεδο αυτόματου, διατηρώντας την έξοδο του DFA απαλλαγμένη από περιττές μονάδες.

Η σχεδίαση με χρήση `enum` και πίνακα μεταβάσεων μέσω `EnumMap` καθιστά τον λεκτικό αναλυτή εύκολα επεκτάσιμο και απλό στη συντήρηση. Η προσθήκη νέων κανόνων απαιτεί μόνο την εισαγωγή επιπλέον μεταβάσεων στον πίνακα, χωρίς επιπλοκή στον κώδικα του `Lexer`.

Σχήμα 3.4.2: Το γράφημα του πεπερασμένου αυτόματου (DFA) του λεκτικού αναλυτή.


START
DIGIT
DIGIT
DIGIT_FINAL
ANY_BUT_DIGIT_OR_LETTER
WHITESPACE
WHITESPACE
WHITESPACE_FINAL
ANY_BUT_WHITESPACE
UNDERSCORE
DIGIT
LETTER
IDENTIFIER_OR_KEYWORD
IDENTIFIER_OR_KEYWORD_FINAL
ANY_OTHER
LETTER
ADD_OPERATOR
ADD_OPERATOR_FINAL
ANY_OTHER
ADD
SUBTRACT
MULL_OPERATOR
MULL_OPERATOR_FINAL
ANY_OTHER
MULTIPLY
DIVIDE
GROUP_SYMBOL
GROUP_SYMBOL_FINAL
ANY_OTHER
SQUARE_BRACKET
PARENTHESIS
ANY_OTHER_THAN_CURLY_CLOSE
CURLY_BRACKET_OPEN
COMMENT
CURLY_BRACKET_CLOSE
COMMENT_BRACKET_CONSUME
ANY
COMMENT_FINAL
EQUAL
EQUAL
ANY_OTHER
EQUAL_FINAL
LESS_THAN
GREATER_THAN
LESS_THAN
ANY_BUT_EQUAL_OR_GREATER
LESS_THAN_FINAL
EQUAL
COMMA
GREATER_THAN
GREATER_THAN
LESS_THAN_EQUAL
ANY_OTHER
LESS_THAN_EQUAL_FINAL
SEMICOLON
COLON
DELIMITER
NOT_EQUAL
ANY_OTHER
NOT_EQUAL_FINAL
MODULO
ANY_OTHER
GREATER_THAN_FINAL
EQUAL
GREATER_THAN_EQUAL
ANY_OTHER
GREATER_THAN_EQUAL_FINAL
ANY
DELIMITER_FINAL
MODULO
ANY_OTHER
MODULO_FINAL

3.5 Lexer

Κατά την κατασκευή του `Lexer`, δέχεται ως όρισμα ένα αντικείμενο `CharStream`, το οποίο της επιτρέπει να προσπελάζει το κείμενο εισόδου με ευελιξία, χρησιμοποιώντας lookahead χαρακτήρες, δυνατότητα επιστροφής, και ακριβή αναφορά θέσης (γραμμή, στήλη).

Μέθοδος `getNextToken()`

Η πιο κρίσιμη μέθοδος της κλάσης είναι η `getNextToken()`, η οποία υλοποιεί τον βασικό βρόχο αναγνώρισης του επόμενου `Token` μέσω του DFA:

1. Ξεκινά από την αρχική κατάσταση `START` και καταγράφει τη θέση έναρξης του token.
2. Κάθε βήμα κάνει lookahead τον επόμενο χαρακτήρα (χωρίς κατανάλωση), και τον μετατρέπει σε τύπο χαρακτήρα μέσω της `CharacterType.of(char)`.
3. Η τρέχουσα κατάσταση του DFA ενημερώνεται με βάση τον τύπο χαρακτήρα που έγινε lookahed προηγουμένως μέσω της `getNextState()`, επιστρέφοντας τη νέα κατάσταση.
4. Αν η νέα κατάσταση είναι σφάλμα — δηλαδή, όταν το DFA μεταβαίνει σε κατάσταση λάθους βάσει του χαρακτήρα εισόδου και της τρέχουσας κατάστασης — καλείται η μέθοδος `LexerErrors.IllegalStateTransition()`, που διακόπτει την εκτέλεση και εμφανίζει το αντίστοιχο μήνυμα.
5. Αν η νέα κατάσταση απαιτεί επανεκκίνηση (`triggersDFARestart()`), τότε ο δείκτης αρχής token επαναφέρεται, αγνοείται η λεκτική μονάδα (π.χ. whitespace ή σχόλιο) και συνεχίζεται από την αρχή.
6. Αν η νέα κατάσταση είναι **τελική** (`isFinal()`), ολοκληρώνεται η αναγνώριση και κατασκευάζεται ένα `Token` με τις πληροφορίες του.

   Σημείωση: οι τελικές καταστάσεις *δεν καταναλώνουν* τον *lookahead* χαρακτήρα, δηλαδή ο χαρακτήρας που προκάλεσε τη μετάβαση σε τελική κατάσταση δεν αφαιρείται από το `CharStream`.

   Αυτό συμβαίνει γιατί για να μεταβούμε σε τελική κατάσταση, σημαίνει ότι έχει διαβαστεί ένας χαρακτήρας που *δεν ανήκει* στο token που αναγνωρίζεται εκείνη τη στιγμή.

   Επομένως, **δεν πρέπει να καταναλωθεί αυτός ο χαρακτήρας**, ώστε να παραμείνει διαθέσιμος ως *ο πρώτος χαρακτήρας του επόμενου διαθέσιμου token*.
7. Αν δεν ισχύει τίποτα από τα παραπάνω, ο χαρακτήρας καταναλώνεται μέσω της `consumeNextChar()` και επαναλαμβάνεται ο βρόχος.

Η μέθοδος εγγυάται ότι κάθε κλήση της παράγει είτε ένα έγκυρο `Token` είτε τερματίζει την εκτέλεση με σφάλμα, διατηρώντας την αυστηρότητα της λεκτικής ανάλυσης. Ειδική μεταχείριση υπάρχει για τον χαρακτήρα EOF, ο οποίος θεωρείται κανονικός χαρακτήρας στο `CharacterType` enum και οδηγεί στην κατασκευή του αντίστοιχου ειδικού token. Με αυτόν τον τρόπο, η διαδικασία λεκτικής ανάλυσης παραμένει συνεπής και καθολική, καθώς το EOF αντιμετωπίζεται ως ξεκάθαρο σήμα τέλους εισόδου. Το EOF token λειτουργεί ως δείκτης ολοκλήρωσης και συνήθως αγνοείται κατά την ανάλυση, ενώ παράλληλα βοηθά στον εντοπισμό του σφάλματος "EOF reached while parsing" όταν η είσοδος τερματίζει απροσδόκητα.

Αλληλεπίδραση με τον Parser

Ο `Lexer` είναι η μοναδική πηγή παραγωγής tokens, τα οποία καταναλώνονται από τον `Parser`. Η επικοινωνία μεταξύ τους είναι απλή αλλά θεμελιώδης: ο `Parser` καλεί επανειλημμένα τη `getNextToken()` για να παραλάβει το επόμενο διαθέσιμο `Token` και να το εξετάσει βάσει της γραμματικής. Αυτή η μορφή lazy παραγωγής διασφαλίζει ότι η ανάλυση γίνεται βήμα-βήμα, χωρίς να απαιτείται εξολοκλήρου πρό-ανάλυση του προγράμματος.

Η σωστή συνεργασία μεταξύ του `Lexer` και του `Parser` είναι κρίσιμη για την επιτυχία της συντακτικής ανάλυσης. Η αρχιτεκτονική αυτή βασίζεται στην έννοια του **on-demand** tokenization, όπου κάθε token παράγεται μόνο όταν ζητηθεί, επιτρέποντας αναλυτές με lookahead και backtracking.

Λέξεις-Κλειδιά

Ο `Lexer` περιλαμβάνει ένα στατικό σύνολο `KEYWORDS`, το οποίο χρησιμοποιείται για να διαχωρίζονται τα κανονικά αναγνωριστικά από τις λέξεις-κλειδιά. Όταν ένα token ταιριάζει συντακτικά με αναγνωριστικό, η `Token` ελέγχει αν αυτό περιέχεται στο `KEYWORDS` και αποδίδει την αντίστοιχη κατηγορία.

Το σύνολο αυτό περιλαμβάνει όλες τις δεσμευμένες λέξεις της γλώσσας Greek++, όπως `πρόγραμμα`, `εάν`, `για`, `διαδικασία`, `και`, `όχι`, κ.λπ.

Κεφάλαιο 4

Συντακτική Ανάλυση (Parser)

Η συντακτική ανάλυση αποτελεί κρίσιμο στάδιο του μεταγλωττιστή, όπου η ακολουθία λεκτικών μονάδων (tokens) που παράγει ο `Lexer` μετατρέπεται σε μια ιεραρχική δομή που αναπαριστά το πρόγραμμα. Στη γλώσσα Greek++, η συντακτική ανάλυση υλοποιείται κυρίως μέσα από δύο βασικές κλάσεις που βρίσκονται στον φάκελο `parser`:

- `Parser.java`: Υλοποιεί τον μηχανισμό της ανάλυσης, αναγνωρίζοντας τη σύνταξη της γλώσσας βάσει της γραμματικής και δημιουργώντας το αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree - AST).
- `ASTNode.java`: Αναπαριστά τους κόμβους του αφηρημένου συντακτικού δέντρου, επιτρέποντας τη δομημένη αποθήκευση και επεξεργασία της συντακτικής δομής.

Ο `Parser` χρησιμοποιεί τα tokens που παράγει ο `Lexer` για να κατασκευάσει το AST, το οποίο αποτελεί τη βάση για τα επόμενα στάδια του μεταγλωττιστή, όπως ο σημασιολογικός έλεγχος και η παραγωγή κώδικα.

4.1 ASTNode.java

Η κλάση `ASTNode` αναπαριστά έναν κόμβο στο αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree - AST) που παράγεται από τον `Parser`. Αποτελεί τη βασική δομική μονάδα που χρησιμοποιείται για την αποθήκευση και αναπαράσταση της ιεραρχικής συντακτικής δομής του προγράμματος στην γλώσσα Greek++.

Κάθε `ASTNode` περιέχει πληροφορίες σχετικά με:

- τον τύπο του κόμβου (π.χ. δήλωση, έκφραση, εντολή),
- το αναγνωριστικό του (προαιρετικά),
- τους γονείς και τους απογόνους του, επιτρέποντας την δένδρωση και περιήγηση στο AST,
- επιπλέον μεταδεδομένα που απαιτούνται για τη σημασιολογική ανάλυση και τη μετάφραση.

Η δομή της κλάσης υποστηρίζει επεκτασιμότητα, επιτρέποντας εύκολη προσθήκη νέων τύπων κόμβων και χαρακτηριστικών κατά την εξέλιξη της γλώσσας και του μεταγλωττιστή.

Η κλάση `ASTNode` αποτελεί το θεμέλιο για όλα τα επόμενα στάδια της μεταγλώττισης, όπως η σημασιολογική ανάλυση αλλά και η παραγωγή τελικού κώδικα.

4.2 Parser.java

Η κλάση `Parser` αποτελεί τον πυρήνα της συντακτικής ανάλυσης στη γλώσσα Greek++. Ο ρόλος της είναι να λαμβάνει ως είσοδο την ακολουθία των `tokens` που παράγει ο `Lexer`, μέσω της μεθόδου `getNextToken()`, και να αναγνωρίζει τη σύνταξη του προγράμματος σύμφωνα με τους κανόνες της γραμματικής της γλώσσας.

Η ανάλυση υλοποιείται με την τεχνική της αναδρομικής κατάβασης (recursive descent parsing), όπου κάθε συντακτικός κανόνας της γραμματικής αντιστοιχεί σε μια ξεχωριστή μέθοδο της κλάσης. Αυτή η μηχανιστική δομή καθιστά τον κώδικα καθαρό, αυτο-επεξηγηματικό και εύκολα επεκτάσιμο, διευκολύνοντας τη συντήρηση και την προσθήκη νέων γλωσσικών δομών.

Κατά την εκτέλεση, ο `Parser` καταναλώνει διαδοχικά τα `tokens` που παράγει ο `Lexer`, καλώντας τις αντίστοιχες μεθόδους που υλοποιούν τις γραμματικές παραγωγές, δημιουργώντας σταδιακά το αφηρημένο συντακτικό

δέντρο (Abstract Syntax Tree - AST). Το AST αποτελεί μια ιεραρχική, δένδρινη δομή που αναπαριστά με σαφήνεια και δομή τη σύνταξη και τη λογική του προγράμματος σε μορφή κατάλληλη για επόμενη σημασιολογική επεξεργασία και παραγωγή κώδικα.

Η χρήση της αναδρομικής κατάβασης επιτρέπει τον φυσικό και αποδοτικό χειρισμό σύνθετων γλωσσικών κατασκευών, όπως εκφράσεις, εντολές ελέγχου ροής, βρόχους και υποπρογράμματα (συναρτήσεις και διαδικασίες), ακολουθώντας πιστά τη δομή της γραμματικής.

Επιπλέον, ο `Parser` ενσωματώνει μηχανισμούς για την ανίχνευση και διαχείριση συντακτικών σφαλμάτων. Κατά την ανάλυση, αν εντοπιστεί σφάλμα, καταγράφεται με ακρίβεια η θέση του (γραμμή, στήλη) και η φύση του σφάλματος, παρέχοντας εποικοδομητική ανατροφοδότηση που διευκολύνει τη διαδικασία αποσφαλμάτωσης του κώδικα από τον προγραμματιστή.

Παρατίθεται η γραμματική της γλώσσας Greek++ στη συνέχεια:

Πανεπιστήμιο Ιωαννίνων
Τμήμα Πληροφορικής
"Μεταφραστές"
Διδάσκων: Γ. Μανής
Φεβρουάριος 2025

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
                    |
                    ;
```

Μέρος ΙΙ

Back-End

Το οπίσθιο τμήμα του μεταγλωττιστή Greek++ οργανώνεται σε τρεις βασικές υποκατηγορίες, οι οποίες αντιστοιχούν στα σημαντικά στάδια της διαδικασίας μετάφρασης μετά τη συντακτική ανάλυση:

- **Τελικός Κώδικας (final_code):** Περιλαμβάνει τις κλάσεις και τις λειτουργίες που σχετίζονται με την παραγωγή του τελικού κώδικα, δηλαδή τη μετατροπή της ενδιάμεσης αναπαράστασης σε assembly κώδικα για τον στόχο (RISC-V).
- **Τετράδες (quads):** Αφορά τη διαχείριση της ενδιάμεσης μορφής προγράμματος μέσω τετραδίων (quads), που αποτελούν μια απλή και τυποποιημένη αναπαράσταση εντολών, διευκολύνοντας τη βελτιστοποίηση και τη γεννήτρια τελικού κώδικα.
- **Σημασιολογικός Έλεγχος (semantic):** Περιλαμβάνει τις κλάσεις που υλοποιούν τους ελέγχους σημασιολογικής ορθότητας, όπως η διαχείριση του πίνακα συμβόλων, ο έλεγχος τύπων, και άλλους κανόνες που αφορούν το νόημα του προγράμματος και όχι τη σύνταξή του.

Κεντρικός ενορχηστρωτής του backend είναι η κλάση `IntermediateGenerator`, η οποία συντονίζει την επικοινωνία μεταξύ των επιμέρους υποσυστημάτων. Συγκεκριμένα, η `IntermediateGenerator` επισκέπτεται το αφηρημένο συντακτικό δέντρο (AST), παράγει τις τετράδες, διαχειρίζεται τον πίνακα συμβόλων, και καλεί τους κατάλληλους managers για τη δημιουργία τελικού κώδικα.

Η δομή αυτή εξασφαλίζει ομαλή ροή πληροφορίας και σαφή κατανομή αρμοδιοτήτων, επιτρέποντας ευκολότερη συντήρηση και επεκτασιμότητα του μεταγλωττιστή.

Σε επόμενα κεφάλαια, κάθε μία από τις τρεις αυτές υποκατηγορίες θα αναλυθεί ξεχωριστά, παρουσιάζοντας τις βασικές κλάσεις και τις λειτουργίες που υλοποιούν, καθώς και τη σημασία τους στην συνολική λειτουργία του μεταγλωττιστή Greek++.

Κεφάλαιο 5

Τετράδες (Quads)

Στο οπίσθιο τμήμα (backend) του μεταγλωττιστή, οι **τετράδες (quads)** αποτελούν την ενδιάμεση αναπαράσταση του προγράμματος, η οποία είναι πιο αφηρημένη από τον τελικό κώδικα μηχανής, αλλά αρκετά λεπτομερής ώστε να υποστηρίζει την παραγωγή του τελικού κώδικα.

Η ενδιάμεση αυτή μορφή υλοποιείται μέσω δύο βασικών κλάσεων:

- `Quad`: Αναπαριστά μια μονάδα ενδιάμεσου κώδικα, την τετράδα, που αποτελείται από έναν τελεστή και έως τρία ορίσματα.
- `QuadManager`: Διαχειρίζεται τη συλλογή των τετράδων, παρέχοντας λειτουργίες προσθήκης, πρόσβασης και διαχείρισης της λίστας των quads καθ' όλη τη διάρκεια της ενδιάμεσης παραγωγής κώδικα.

Η κλάση `QuadManager` λειτουργεί ως κεντρικός διαχειριστής της ενδιάμεσης αναπαράστασης, ενώ η `Quad` αποτελεί την βασική μονάδα που αποθηκεύει τις λειτουργίες που αντιστοιχούν σε εντολές του προγράμματος.

5.1 Κλάση Quad

Η κλάση `Quad` υλοποιεί την ενδιάμεση εντολή, η οποία αποτελείται από:

- έναν **τελεστή** (operator) που καθορίζει την πράξη ή εντολή,
- έως τρία **ορίσματα** (operands) που είναι οι παράμετροι της εντολής.

Το τελευταίο όρισμα το ονομάζουμε result και το θεωρούμε τον καταχωρήτη "στοχο" σε κάθε εντολή ενδιαμέσου κώδικα όπου η έννοια αυτή έχει νόημα.

Η κλάση παρέχει μεθόδους για πρόσβαση και τροποποίηση των παραπάνω πεδίων, καθώς και μεθόδους για εμφάνιση και αποσφαλμάτωση της τετράδας.

Η χρήση της `Quad` επιτρέπει την αναπαράσταση σύνθετων γλωσσικών κατασκευών με απλές εντολές, διευκολύνοντας τον σχεδιασμό του backend.

5.2 Κλάση QuadManager

Η κλάση `QuadManager` είναι υπεύθυνη για τη διαχείριση της λίστας των τετράδων (`quads`) που παράγονται κατά την ενδιάμεση φάση του μεταγλωττιστή. Αναλαμβάνει την προσθήκη νέων τετράδων, την ενημέρωση και διαχείριση των αριθμών θέσης (quad numbers), την ανάκτηση τετράδων βάσει της θέσης τους, καθώς και την υποστήριξη λειτουργιών όπως το backpatching για διορθώσεις σε διευθύνσεις κώδικα.

Η κλάση αυτή αποτελεί κρίσιμο κομμάτι της παραγωγής ενδιάμεσου κώδικα, προσφέροντας δομή και ευελιξία στην τροποποίηση και την οργάνωση του κώδικα που παράγεται.

5.2.1 Delayed Quads και Αναδρομική Παραγωγή

Ένα σημαντικό χαρακτηριστικό της κλάσης `QuadManager` είναι η υποστήριξη των *delayed quads* — δηλαδή των τετράδων που δεν καταχωρούνται άμεσα στη βασική λίστα, αλλά καθυστερεί η εγγραφή τους. Αυτό είναι απαραίτητο σε περιπτώσεις κλήσεων διαδικασιών ή συναρτήσεων, ιδίως όταν τα ορίσματα των κλήσεων είναι εκφράσεις που πρέπει πρώτα να αποτιμηθούν από άλλους συντακτικούς κανόνες της γλώσσας. Λόγω της αναδρομικής φύσης της συντακτικής ανάλυσης, οι αντίστοιχες τετράδες δεν δημιουργούνται με γραμμική σειρά, και έτσι απαιτείται προσωρινή αποθήκευση και μεταγενέστερη εισαγωγή τους.

Πρακτικά, κατά τις κλήσεις διαδικασιών ή συναρτήσεων, δημιουργείται ένα `HashMap<Integer, List<Quad>>` όπου το κλειδί είναι το επίπεδο scope την στιγμή που γίνεται η κλήση (depth level) και η τιμή είναι η λίστα με τα quads που παράγονται σε αυτό το επίπεδο.

Επείδη ουσιαστικά τα quads που παράγονται σε μια κλήση και αφορούν τις μεταβλητές ανήκουν σε διαφορετικό scope απο τα quads τα όποια αφορούν την κλήση της συνάρτησης ή το quad για το return value , γίνεται αυτή η διαχείρηση.

Κεφάλαιο 6

Πίνακας Συμβόλων (ScopeManager)

Η κλάση `ScopeManager` υλοποιεί το μηχανισμό διαχείρισης των scopes (πεδίων ορατότητας) και των συμβόλων (μεταβλητών και υποπρογραμμάτων) μέσα στον μεταγλωττιστή Greek++. Αποτελεί κρίσιμο μέρος της σημασιολογικής ανάλυσης, καθώς επιτρέπει τον έλεγχο δήλωσης, εύρεσης και εγκυρότητας των μεταβλητών και των συναρτήσεων σε διαφορετικά επίπεδα του προγράμματος.

Η διαχείριση των scopes γίνεται μέσω της εσωτερικής κλάσης `Scope`, η οποία διατηρεί έναν πίνακα συμβόλων (`SymbolTable`) που περιλαμβάνει:

- τις τοπικές μεταβλητές (`variables`),
- και τις υπορουτίνες (συναρτήσεις ή διαδικασίες) (`subroutines`).

Η κλάση υποστηρίζει ιεραρχική δένδρωση των scopes, με κάθε scope να γνωρίζει τον γονέα του, επιτρέποντας την ανίχνευση και επίλυση ονομάτων σε όλα τα περιβάλλοντα ορατότητας.

6.1 Λειτουργίες και Χρήσεις

Ο `ScopeManager` παρέχει λειτουργίες για:

- το άνοιγμα (`openScope()`) και κλείσιμο (`closeScope()`) scopes, διατηρώντας το τρέχον scope και το βάθος του,
- την προσθήκη τοπικών μεταβλητών και υποπρογραμμάτων στο τρέχον scope,

- την αναζήτηση μεταβλητών και υποπρογραμμάτων σε όλη την αλυσίδα scopes (με ανίχνευση σε γονικές περιοχές),
- την καταγραφή και αποθήκευση ιστορικού των scopes για αποσφαλμάτωση και έλεγχο,
- την καταγραφή και διαχείριση προσωρινών μεταβλητών σε κάθε scope,
- την έκδοση μηνυμάτων σφαλμάτων για δηλώσεις και κλήσεις που παραβιάζουν τους κανόνες δήλωσης και χρήσης.

6.2 Δομή και Υλοποίηση

6.2.1 Εσωτερική Κλάση Scope

Η εσωτερική κλάση `Scope` αντιπροσωπεύει ένα επίπεδο ορατότητας και περιλαμβάνει:

- το γονικό scope (εκτός αν είναι το global),
- και τον πίνακα συμβόλων με τις μεταβλητές και τις υπορουτίνες που ανήκουν σε αυτό.

Ο πίνακας συμβόλων (ειδικότερα η κλάση `SymbolTable`) οργανώνει τα σύμβολα σε δύο χάρτες `Map`: ένα για τις μεταβλητές και ένα για τις υπορουτίνες.

6.2.2 Διαχείριση Συμβόλων

Οι βασικές μέθοδοι του `ScopeManager` για διαχείριση συμβόλων είναι:

- `addSubroutine(Procedure)`: Προσθέτει ένα νέο σύμβολο στον πίνακα συμβόλων για μια συνάρτηση ή μια διαδικασία. Ελέγχει για διπλές δηλώσεις με βάση την υπογραφή της ρουτίνας.
- `declareVariable`, `declareParameter`, `declareFunction`, `declareProcedure`: Υλοποιούν τη δήλωση νέων συμβόλων με έλεγχο εγκυρότητας και εκχώρηση στο κατάλληλο activation record.
- `resolveVariable(String)` και `resolveSubroutine(String)`: Αναζητούν μια μεταβλητή ή υπορουτίνα ξεκινώντας από το τρέχον scope και ανεβαίνοντας προς τα γονικά scopes για την εύρεση του ονόματος.

6.2.3 Διαχείριση Προσωρινών Μεταβλητών

Ο `ScopeManager` διατηρεί ένα ξεχωριστό χάρτη προσωρινών μεταβλητών (`TemporaryVariable`) για το τρέχον scope. Αυτές χρησιμοποιούνται κατά την παραγωγή ενδιάμεσου κώδικα για προσωρινή αποθήκευση τιμών και αποτελεσμάτων ενδιάμεσων υπολογισμών.Ενδιάμεσος κωδικάς για το εκκάστοτε scope δημιουργέιται κάθε φόρα από μέσα προς τα έξω. Αρά δεν χρειάζεται να έχει το κάθε scope entity δομή για τις προσωρινές μεταβλητές.

6.2.4 Καταγραφή Ιστορικού Scopes

Κατά το κλείσιμο ενός scope, καταγράφεται σε `scopesLog` η περιγραφή των μεταβλητών και υπορουτινών που περιείχε, μαζί με το βάθος του scope. Αυτό το αρχείο καταγραφής βοηθά στην αποσφαλμάτωση και στην αναλυτική παρακολούθηση της σημασιολογικής ανάλυσης.

Κεφάλαιο 7

Symbol

Στο πακέτο symbol διαχειριζόμαστε όλες τις οντότητες που αποθηκεύουμε στον πίνακα συμβόλων του κάθε scope, δηλαδή μεταβλητές, παραμέτρους, συναρτήσεις, διαδικασίες και προσωρινές μεταβλητές. Αυτές οι κλάσεις παρέχουν τη βάση για τη σημασιολογική ανάλυση, τον έλεγχο τύπων, καθώς και την οργάνωση της στοίβας κατά την παραγωγή τελικού κώδικα.

Η κεντρική οντότητα που ενορχηστρώνει και συνδέει τα συμβολικά στοιχεία με το περιβάλλον εκτέλεσης είναι το `ActivationRecord`, το οποίο αντιστοιχεί στο ενεργό πλαίσιο εκτέλεσης (activation frame) κάθε υποπρογράμματος.

7.1 ActivationRecord

Η κλάση `ActivationRecord` αναπαριστά το ενεργό πλαίσιο (activation frame) ενός υποπρογράμματος κατά την εκτέλεση ή κατά την παραγωγή κώδικα. Περιλαμβάνει αναλυτική πληροφορία για όλες τις τοπικές μεταβλητές, παραμέτρους, και προσωρινές μεταβλητές, καθώς και τα offset τους στη στοίβα.

Αυτή η κλάση είναι υπεύθυνη για τον υπολογισμό και τη διαχείριση των offsets και του συνολικού μήκους του πλαισίου, ώστε να εξασφαλίζεται η σωστή πρόσβαση σε μεταβλητές κατά τη διάρκεια της εκτέλεσης.

7.2 DataType

Η κλάση `DataType` ορίζει τους βασικούς τύπους δεδομένων της γλώσσας Greek++, όπως το `Integer`. Παρέχει πληροφορίες όπως το μέγεθος σε bytes, που είναι απαραίτητο για την τοποθέτηση μεταβλητών στο ενεργό πλαίσιο.

Η ύπαρξη αυτής της κλάσης επιτρέπει την τυποποιημένη διαχείριση τύπων και διευκολύνει την υλοποίηση του ελέγχου τύπων και την παραγωγή κώδικα.

7.3 Entity

Η αφηρημένη κλάση `Entity` αποτελεί τη βασική υπερκλάση για όλα τα σύμβολα που μπορεί να εμφανιστούν σε ένα πρόγραμμα (π.χ. μεταβλητές, παράμετροι, συναρτήσεις, διαδικασίες). Ορίζει κοινά χαρακτηριστικά όπως το όνομα και τον τύπο.

Η χρήση της κλάσης αυτής επιτρέπει ομοιογενή χειρισμό και επεκτασιμότητα στον ορισμό συμβόλων.

7.4 Function

Η κλάση `Function` εξειδικεύει την `Procedure` και αντιπροσωπεύει τις συναρτήσεις που επιστρέφουν τιμή. Περιλαμβάνει επιπλέον πεδία και μεθόδους που αφορούν την επιστροφή τιμής και τη διαχείριση του αντίστοιχου πλαισίου ενεργοποίησης.

7.5 LocalVariable

Η κλάση `LocalVariable` αναπαριστά τοπικές μεταβλητές μέσα σε ένα scope. Περιέχει πληροφορίες όπως το όνομα, ο τύπος, και το offset στο activation record, απαραίτητο για την τοποθέτηση και πρόσβαση στη μνήμη.

Επιπλέον περιέχει το ScopeDepth της μεταβλήτη κάτα την δημιουργία της προκειμένου στην φάση του τελικού κώδικα να μπορεί να μπορεί να εντοπιστεί σε ποιας στοίβας activation record βρίσκεται αποθηκευμένη.

7.6 Parameter

Η κλάση `Parameter` είναι υποκατηγορία της `LocalVariable` και αντιπροσωπεύει παραμέτρους υποπρογραμμάτων (εισόδου ή εξόδου). Περιλαμβάνει πρόσθετα χαρακτηριστικά που διευκολύνουν τον χειρισμό του μηχανισμού περάσματος παραμέτρων.

7.7 Procedure

Η κλάση `Procedure` αναπαριστά διαδικασίες (υποπρογράμματα που δεν επιστρέφουν τιμή). Περιλαμβάνει στοιχεία όπως το όνομα, τις παραμέτρους και το ενεργό πλαίσιο που διαχειρίζεται.

7.8 TemporaryVariable

Η κλάση `TemporaryVariable` αναπαριστά προσωρινές μεταβλητές που χρησιμοποιούνται κατά την παραγωγή ενδιάμεσου κώδικα. Πρόκειται για μεταβλητές που δεν ορίζονται απευθείας από τον χρήστη αλλά δημιουργούνται για την υποστήριξη των υπολογισμών και μετατροπών του compiler.

Περιλαμβάνει μηχανισμούς για διαχείριση ονομάτων, τύπων και offsets ώστε να ενσωματώνεται σωστά στο activation record και να υποστηρίζει την παραγωγή κώδικα.

Κεφάλαιο 8

Ενδιάμεση Αναπαράσταση (Intermediate Code)

Η ενδιάμεση αναπαράσταση (Intermediate Code) αποτελεί ένα κρίσιμο στάδιο στον μεταγλωττιστή Greek++. Ο σκοπός της είναι να μετασχηματίσει το αφηρημένο συντακτικό δέντρο (AST) σε μία πιο απλοποιημένη, τυποποιημένη μορφή, η οποία διευκολύνει τον σημασιολογικό έλεγχο και την παραγωγή τελικού κώδικα.

Η ενδιάμεση αναπαράσταση που χρησιμοποιείται είναι βασισμένη σε **τετράδες** (quads), οι οποίες περιγράφουν βασικές εντολές της γλώσσας σε μορφή τριών-τεσσάρων τελεστών (π.χ. τελεστής, αριστερός και δεξιός τελεστής, και αποδέκτης).

8.1 Visitor

Η κλάση `Visitor` ορίζει τη διεπαφή του visitor pattern για την περιήγηση στο AST. Περιλαμβάνει μεθόδους επίσκεψης για κάθε τύπο κόμβου (π.χ. δήλωση, έκφραση, εντολή, υποπρόγραμμα).

Η `IntermediateGenerator` υλοποιεί αυτή τη διεπαφή, ορίζοντας το τι ακριβώς πρέπει να συμβεί σε κάθε επίσκεψη κόμβου, δηλαδή πώς παράγεται ο ενδιάμεσος κώδικας για κάθε γλωσσική δομή.

8.2 IntermediateGenerator

Η κλάση `IntermediateGenerator` λειτουργεί ως κεντρικός ενορχηστρωτής της ενδιάμεσης αναπαράστασης. Χρησιμοποιεί το πρότυπο `visitor pattern` ώστε να επισκέπτεται τους κόμβους του αφηρημένου συντακτικού δέντρου

(AST) και να παράγει τις αντίστοιχες τετράδες, να ενημερώνει τον πίνακα συμβόλων και να προκαλεί την παραγωγή τελικού κώδικα RISC-V.

Η δομή της κλάσης περιλαμβάνει τα εξής βασικά πεδία:

```
private final QuadManager quadManager;
private final ScopeManager scopeManager;
private final RiscVAssemblyGenerator asmManager;
private Procedure currentBlockOwner;
```

- `quadManager`: Διαχειρίζεται τη συλλογή των παραγόμενων τετραδών (intermediate code).
- `scopeManager`: Υπεύθυνος για το άνοιγμα και κλείσιμο scopes, τη διαχείριση μεταβλητών, παραμέτρων και υποπρογραμμάτων, καθώς και την καταγραφή του activation record.
- `asmManager`: Χρησιμοποιείται για την παραγωγή τελικού RISC-V assembly code για κάθε scope ξεχωριστά.
- `currentBlockOwner`: Αναφέρεται στο υποπρόγραμμα (π.χ. κύριο πρόγραμμα, διαδικασία, συνάρτηση) που αποτελεί τον **κάτοχο του τρέχοντος scope** και στο οποίο αποδίδονται οι μεταβλητές, οι παράμετροι και οι προσωρινές μεταβλήτες.

Βασικές λειτουργίες της περιλαμβάνουν:

- **Δημιουργία τετραδών**: Μετατροπή των δηλώσεων, εκφράσεων, εντολών και υποπρογραμμάτων σε αντίστοιχες τετράδες.
- **Διαχείριση scope**: Ενσωμάτωση του `ScopeManager` για τη σωστή διαχείριση μεταβλητών και υποπρογραμμάτων σε επίπεδο scope.
- **Διαχείριση προσωρινών μεταβλητών**: Κατανομή και χρήση προσωρινών μεταβλητών για ενδιάμεσα αποτελέσματα.
- **Διαχείριση κλήσεων συναρτήσεων**: Υλοποίηση της λογικής για κλήσεις συναρτήσεων και διαδικασιών, συμπεριλαμβανομένων των delayed quads για τη διαχείριση των αναδρομικών κλήσεων.
- **Παραγωγή κώδικα για δομές ελέγχου**: Υποστήριξη για εντολές `if`, `while`, `for` κ.ά. με κατάλληλο χειρισμό των ετικετών και των διακλαδώσεων.

8.3 Βασικά Πεδία του ASTNode για Ενδιάμεσο Κώδικα

Κατά την παραγωγή του ενδιάμεσου κώδικα, κάθε κόμβος του αφηρημένου συντακτικού δέντρου (`ASTNode`) δεν λειτουργεί απλώς ως δομικό στοιχείο του προγράμματος, αλλά μεταφέρει και κρίσιμες πληροφορίες που είναι απαραίτητες για τη δημιουργία των τετράδων και τον έλεγχο της ροής του προγράμματος.

Τα τρία βασικά πεδία που χρησιμοποιούνται από τον μεταγλωττιστή για αυτόν τον σκοπό είναι:

- `place`: Καταγράφει τη θέση (όνομα μεταβλητής ή προσωρινής τιμής) στην οποία αποθηκεύεται το αποτέλεσμα της αξιολόγησης του κόμβου. Χρησιμοποιείται κυρίως για αριθμητικές και εκφραστικές δομές.
- `trueList`: Περιέχει λίστα από διευθύνσεις τετραδών που πρέπει να συμπληρωθούν (backpatch) εφόσον η συνθήκη που σχετίζεται με τον κόμβο αξιολογηθεί ως αληθής. Χρησιμοποιείται σε λογικές εκφράσεις και συνθήκες ελέγχου ροής.
- `falseList`: Αντίστοιχα, καταγράφει τις διευθύνσεις τετραδών που πρέπει να διορθωθούν σε περίπτωση που η συνθήκη είναι ψευδής.

Τα πεδία αυτά αποτελούν το βασικό μηχανισμό επικοινωνίας μεταξύ των κόμβων του AST και του υποσυστήματος παραγωγής κώδικα. Η παρουσία και σωστή χρήση τους είναι κρίσιμη για την ορθή γραμμική αναπαράσταση του προγράμματος μέσω τετραδών και την υποστήριξη σύνθετων ροών ελέγχου.

8.4 Ενδεικτική Περιγραφή Ροής Παραγωγής για Επιλεγμένους Κανόνες

Στην παρούσα ενότητα παρουσιάζεται η βασική ροή παραγωγής ενδιάμεσου κώδικα για ορισμένους χαρακτηριστικούς συντακτικούς κανόνες της γλώσσας Greek++. Η περιγραφή εστιάζει στη λογική της επεξεργασίας του αφηρημένου συντακτικού δέντρου (AST), στην παραγωγή των αντίστοιχων τετραδών και στη χρήση των βοηθητικών πεδίων `place`, `trueList` και `falseList` ανάλογα με τη φύση του κάθε κανόνα.

Επεξεργασία αριθμητικής έκφρασης (visitExpression)

Η μέθοδος που επεξεργάζεται μία αριθμητική έκφραση υλοποιεί τη λογική του κανόνα:

```
Expression ::= OptionalSign Term ( AddOp Term )*
```

Η λειτουργία της μεθόδου πραγματοποιείται σε έξι διαδοχικά βήματα:

1. Ξεκινάμε διαβάζοντας τον πρώτο υποκόμβο της έκφρασης, που αντιστοιχεί στο προαιρετικό πρόσημο (+ ή -). Τον επισκεπτόμαστε, δηλαδή καλούμε τη μέθοδο επίσκεψης του αντίστοιχου κανόνα (`visitOptionalSign`) και περιμένουμε να ολοκληρώσει την ανάλυσή του και να γεμίσει το πεδίο `place` του κόμβου με την αντίστοιχη πληροφορία (π.χ. "-" ή τίποτα).

2. Στη συνέχεια ανακτούμε τον δεύτερο υποκόμβο, που είναι ο πρώτος αριθμητικός όρος (Term). Τον επισκεπτόμαστε με παρόμοιο τρόπο: καλείται η αντίστοιχη μέθοδος (`visitTerm`) η οποία αναδρομικά θα επισκεφθεί όλους τους υποκόμβους του (π.χ. `Factor`, `Number`, `Variable`, κ.λπ.), ώστε να ολοκληρωθεί η αξιολόγηση του όρου και να καταχωρηθεί στο `place` του κόμβου η μεταβλητή που περιέχει το αποτέλεσμά του. Αυτή είναι ουσιαστικά η πρώτη υπολογισμένη τιμή της αριθμητικής έκφρασης.

3. Αν το πρόσημο ήταν αρνητικό, τότε εφαρμόζεται η κατάλληλη ενέργεια μετατροπής. Συγκεκριμένα, δημιουργείται τετράδα της μορφής `(-, 0, T1, temp)`, η οποία αφαιρεί τον όρο από το μηδέν, και δημιουργείται προσωρινή μεταβλητή για το αποτέλεσμα. Η νέα αυτή μεταβλητή αντικαθιστά την τιμή του πρώτου όρου, ώστε να χρησιμοποιηθεί ως βάση στους υπόλοιπους υπολογισμούς.

4. Στη συνέχεια εξετάζουμε αν υπάρχουν επιπλέον τελεστές και αριθμητικοί όροι στην έκφραση (π.χ. `+ T2`, `- T3`, κ.λπ.). Αν ναι, για κάθε ζευγάρι (τελεστής, όρος) που ακολουθεί:

   - Ανακτούμε τον κόμβο του τελεστή και τον επισκεπτόμαστε. Αυτό έχει ως αποτέλεσμα να γεμίσει το `place` του κόμβου με τον χαρακτήρα του τελεστή ("+" ή "-").
   - Στη συνέχεια ανακτούμε τον επόμενο αριθμητικό όρο και τον επισκεπτόμαστε, ώστε να αξιολογηθεί πλήρως και να επιστραφεί η μεταβλητή που περιέχει το αποτέλεσμά του.
   - Δημιουργείται νέα προσωρινή μεταβλητή για την αποθήκευση του αποτελέσματος της πράξης.
   - Παράγεται τετράδα που εκτελεί την πράξη του τελεστή ανάμεσα στην προηγούμενη τιμή και το νέο όρο.

- Το αποτέλεσμα της πράξης αντικαθιστά την προηγούμενη τιμή και χρησιμοποιείται ως βάση για την επόμενη επανάληψη.

5. Η παραπάνω διαδικασία επαναλαμβάνεται για κάθε επιπλέον πρόσθεση ή αφαίρεση στην έκφραση. Το ενδιάμεσο αποτέλεσμα αποθηκεύεται κάθε φορά σε νέα προσωρινή μεταβλητή και επανατροφοδοτείται στο επόμενο βήμα.

6. Όταν ολοκληρωθεί η επεξεργασία όλων των στοιχείων της έκφρασης, η τελική μεταβλητή που περιέχει το αποτέλεσμα της αποτίμησης τοποθετείται στο πεδίο `place` του κόμβου `Expression`, ώστε να είναι διαθέσιμη στον γονικό κανόνα.

Επεξεργασία εντολής `if` (visitIfStatement)

Η μέθοδος αυτή υλοποιεί τον ενδιάμεσο κώδικα που αντιστοιχεί σε μία δομή `if - else`. Η επεξεργασία πραγματοποιείται βήμα-βήμα ως εξής:

1. Ξεκινάμε διαβάζοντας τον υποκόμβο που περιέχει τη συνθήκη της `if`. Επισκεπτόμαστε αυτόν τον κόμβο, δηλαδή καλούμε τη μέθοδο επίσκεψης που αντιστοιχεί στον κανόνα της λογικής έκφρασης. Η αποτίμηση δεν επιστρέφει αριθμητικό αποτέλεσμα αλλά δύο λίστες:

   - `trueList`: περιέχει τις τετράδες στις οποίες πρέπει να συμπληρωθεί διεύθυνση εκτέλεσης για την περίπτωση που η συνθήκη είναι αληθής.
   - `falseList`: περιέχει τις τετράδες που θα εκτελεστούν όταν η συνθήκη είναι ψευδής.

   Οι τετράδες στις λίστες αυτές έχουν ανοιχτό προορισμό (δηλαδή άλματα χωρίς στόχο) και απαιτούν μεταγενέστερη συμπλήρωση με τη διαδικασία `backpatch()`.

2. Αμέσως μετά, γνωρίζουμε ποια είναι η επόμενη διαθέσιμη εντολή — δηλαδή η πρώτη του σώματος της `if`. Επομένως, εκτελούμε **backpatching στη λίστα `trueList`**, ώστε:

   - Όλες οι τετράδες της `trueList` να ενημερωθούν με άλμα προς το σώμα της `if`.
   - Η εκτέλεση να συνεχίσει κανονικά μέσα στη δομή, αν η συνθήκη είναι αληθής.

3. Στη συνέχεια επισκεπτόμαστε τον κόμβο που αντιστοιχεί στο σώμα της `if` το sequenceNode , δηλαδή τις εντολές που εκτελούνται όταν η συνθήκη ισχύει. Οι εντολές αυτές επεξεργάζονται αναδρομικά και παράγεται ο αντίστοιχος ενδιάμεσος κώδικας.

4. Αφού ολοκληρωθεί η επεξεργασία του σώματος της `if`, δημιουργούμε μία άλμα-εντολή χωρίς στόχο, ώστε:
    - Να παρακαμφθεί το `else` όταν η συνθήκη είναι αληθής.
    - Η θέση αυτής της άλμα-εντολής αποθηκεύεται προσωρινά σε λίστα `ifList`, για να συμπληρωθεί αργότερα.
5. Πριν ξεκινήσει η επεξεργασία του `else`, εκτελούμε **backpatching στη `falseList`**, δηλαδή:
    - Όλες οι τετράδες στη `falseList` ενημερώνονται ώστε να κάνουν άλμα στην πρώτη εντολή του `else`-μπλοκ.
    - Έτσι, αν η συνθήκη είναι ψευδής, η εκτέλεση θα μεταβεί σωστά στο `else`.
6. Επισκεπτόμαστε τον κόμβο του `else`, δηλαδή τις εντολές που εκτελούνται όταν η συνθήκη δεν ισχύει. Όπως και πριν, το περιεχόμενο του κόμβου επεξεργάζεται αναδρομικά.
7. Τέλος, εκτελείται `backpatch()` στη λίστα `ifList`, ώστε το άλμα που είχαμε δημιουργήσει νωρίτερα (μετά την `ifList`) να δείχνει σωστά στην πρώτη εντολή μετά το τέλος της όλης δομής. Με αυτόν τον τρόπο διασφαλίζεται ότι, αν η συνθήκη ήταν αληθής, το πρόγραμμα δεν θα εκτελέσει και το `else`.

Η τεχνική του **backpatching** επιτρέπει τη διαχείριση των άλμα-εντολών με δυναμικό τρόπο. Οι λίστες `trueList` και `falseList` λειτουργούν ως προσωρινές δομές που συγκρατούν "εκκρεμή" άλματα, τα οποία συμπληρώνονται μόλις γνωρίζουμε τον ακριβή προορισμό τους. Έτσι, ο ενδιάμεσος κώδικας διατηρεί τη σωστή λογική ροή ανεξαρτήτως του αν η συνθήκη ισχύει ή όχι.

Επεξεργασία εντολής `for` (visitForStatement)

Η μέθοδος αυτή υλοποιεί την εντολή επανάληψης `for`, λαμβάνοντας υπόψη τη δυναμική κατεύθυνση του βρόχου (αύξουσα ή φθίνουσα), ανάλογα με το πρόσημο του `step`. Ο παραγόμενος ενδιάμεσος κώδικας προσαρμόζεται δυναμικά σε χρόνο εκτέλεσης (runtime) βάσει της τιμής αυτής. Ακολουθεί η αναλυτική περιγραφή των βημάτων:

1. Διαβάζουμε τον αναγνωριστικό κόμβο (`ID`) που αντιπροσωπεύει τη μεταβλητή ελέγχου του βρόχου, καθώς και τις εκφράσεις:
    - `expression1`: αρχική τιμή της μεταβλητής.
    - `expression2`: τελική τιμή (όριο).
    - `stepNode`: τιμή του βήματος.
    - `sequenceNode`: το σώμα της `for`.

2. Επισκεπτόμαστε την expression1 και παράγεται τετράδα :=, expr1, -, ID, που θέτει την αρχική τιμή στη μεταβλητή ελέγχου.

3. Επισκεπτόμαστε στη συνέχεια την expression2 και το stepNode για να προσδιοριστούν οι τιμές του άνω ορίου και του βήματος.

4. Δημιουργούμε δομή διακλάδωσης για να διαπιστωθεί σε χρόνο εκτέλεσης αν το step είναι θετικό ή αρνητικό:

   - Παράγεται τετράδα >=, step, 0, -. Αν ισχύει, θα ακολουθηθεί η λογική για **αύξουσα επανάληψη**.
   - Παράγεται jump που οδηγεί στη λογική της φθίνουσας, αν αποτύχει η πρώτη.

5. Προετοιμάζονται δύο λίστες:

   - checkTrueList: τετράδες που θα συμπληρωθούν με είσοδο στο σώμα του βρόχου.
   - checkFalseList: τετράδες που πρέπει να οδηγούν στην έξοδο από τον βρόχο.

6. Αν το step >= 0:

   - Δημιουργείται τετράδα <=, ID, expr2, -, που ελέγχει αν η τιμή έχει υπερβεί το άνω όριο.
   - Αν ισχύει, συνεχίζουμε στο σώμα. Αν όχι, γίνεται άλμα έξω από τον βρόχο.

7. Αν το step < 0:

   - Δημιουργείται τετράδα >=, ID, expr2, -, για φθίνουσα επανάληψη.
   - Παρόμοια με πριν, οδηγούμαστε στο σώμα ή στην έξοδο.

8. Όλες οι τετράδες που κατευθύνουν στο σώμα συμπληρώνονται μέσω backpatch(checkTrueList, nextquad()) και επισκεπτόμαστε το sequenceNode για να παραχθεί ο αντίστοιχος ενδιάμεσος κώδικας του σώματος.

9. Μετά την εκτέλεση του σώματος, πρέπει να αυξηθεί ή να μειωθεί η τιμή του ID:

   - Δημιουργείται προσωρινή μεταβλητή temp και τετράδα +, ID, step, temp.
   - Ακολουθεί ανάθεση :=, temp, -, ID, που ενημερώνει τον μετρητή του βρόχου.

10. Στο τέλος, επαναλαμβάνεται η λογική για την κατεύθυνση:

    - Αν το step >= 0, επιστρέφουμε στον έλεγχο <=.
    - Αλλιώς, στον έλεγχο >=.

11. Τέλος, οι τετράδες που οδηγούν στην έξοδο από τον βρόχο (checkFalseList) συμπληρώνονται ώστε να μεταφέρουν τη ροή στην εντολή που ακολουθεί μετά τη for.

Η συγκεκριμένη υλοποίηση επιτυγχάνει πλήρη γενικότητα ως προς την κατεύθυνση της επανάληψης. Ο έλεγχος αν το step είναι θετικό ή αρνητικό γίνεται δυναμικά, και οι συγκρίσεις προσαρμόζονται ανάλογα, εξασφαλίζοντας ότι η συνθήκη τερματισμού του βρόχου εφαρμόζεται ορθά. Ο παραγόμενος ενδιάμεσος κώδικας είναι ανεξάρτητος από το αν το step είναι σταθερά ή έκφραση που υπολογίζεται σε runtime.

Επεξεργασία συνάρτησης function (visitFunction)

Η μέθοδος αυτή είναι υπεύθυνη για την επεξεργασία και τον ενδιάμεσο κώδικα μιας στη γλώσσα Greek++. Η λειτουργία της υλοποιεί βήμα-βήμα τα στάδια δήλωσης, ανάλυσης, παραγωγής κώδικα και επιστροφής τιμής, όπως περιγράφεται παρακάτω:

1. **Ανάγνωση ονόματος συνάρτησης:** Αποθηκεύεται το αναγνωριστικό όνομα της συνάρτησης και δημιουργείται αντικείμενο Function με τύπο Integer, το οποίο και δηλώνεται στο ScopeManager. Το βάθος της συνάρτησης είναι currentDepth + 1, καθώς ανοίγεται νέο scope.
2. **Ορισμός τρέχοντος ιδιοκτήτη:** Η μεταβλητή currentBlockOwner τίθεται να δείχνει στη συνάρτηση. Αυτό επιτρέπει την κατάλληλη αντιστοίχιση παραμέτρων και της μεταβλητής επιστροφής με το activation record της συνάρτησης.
3. **Άνοιγμα νέου scope:** Ο ScopeManager ανοίγει νέο scope, ώστε οι τοπικές μεταβλητές, παράμετροι και δηλώσεις της συνάρτησης να αποθηκευτούν στο δικό της περιβάλλον ορατότητας.
4. **Επεξεργασία παραμέτρων:** Επισκεπτόμαστε τον κόμβο των τυπικών παραμέτρων (formalParametersListNode) και καταχωρούνται στον πίνακα συμβόλων της συνάρτησης.
5. **Δήλωση μεταβλητής επιστροφής:** Ο ScopeManager προσθέτει ειδική μεταβλητή με το ίδιο όνομα της συνάρτησης, που λειτουργεί ως return variable. Αυτή θα είναι και ο τελικός αποδέκτης της τιμής επιστροφής.
6. **Επίσκεψη επιμέρους υποκόμβων του σώματος:**
   - functionInput: δηλώσεις παραμέτρων εισόδου.
   - functionOutput: δηλώσεις εξόδου ή αναφοράς.

- `declarations`: τοπικές μεταβλητές.
- `subprograms`: εσωτερικά υποπρογράμματα.
- `sequence`: κύριες εντολές της συνάρτησης.

Όλα τα παραπάνω επισκέπτονται αναδρομικά και επεξεργάζονται πλήρως το σώμα της συνάρτησης.

7. **Έναρξη και επισήμανση του block:**
   - Παράγεται τετράδα `begin_block` με το όνομα της συνάρτησης.
   - Καταγράφεται η θέση της πρώτης εντολής του block στο `ActivationRecord` της συνάρτησης.
8. **Επίσκεψη του sequence:** Εκτελείται επίσκεψη στον κύριο κορμό των εντολών της συνάρτησης, και παράγεται ενδιάμεσος κώδικας.
9. **Επιστροφή τιμής:**
   - Παράγεται τετράδα `retv` με την τιμή της μεταβλητής επιστροφής (η οποία έχει δηλωθεί στο ίδιο όνομα με τη συνάρτηση).
   - Ολοκληρώνεται με τετράδα `end_block` για να δηλωθεί το τέλος του frame.
10. **Καταγραφή προσωρινών μεταβλητών:** Όλες οι `temporary variables` που δημιουργήθηκαν εντός της συνάρτησης προστίθενται στο `ActivationRecord` για λόγους allocation και τελικού κώδικα.
11. **Παραγωγή τελικού κώδικα:** Καλείται η `generateAsmForCurrentScope()` ώστε να μεταφραστεί ο ενδιάμεσος κώδικας του συγκεκριμένου scope σε τελικό RISC-V assembly.
12. **Κλείσιμο scope:** Ο `ScopeManager` τερματίζει το scope της συνάρτησης και επιστρέφει στο εξωτερικό περιβάλλον.

Σημείωση: Οι υπόλοιποι συντακτικοί κανόνες της γλώσσας Greek++ επεξεργάζονται με παρόμοιο τρόπο, ακολουθώντας την ίδια συστηματική λογική: κάθε κόμβος επισκέπτεται αναδρομικά τα παιδιά του, συλλέγει τις απαραίτητες σημασιολογικές πληροφορίες (όπως `place`, `trueList`, `falseList`), και παράγει ενδιάμεσο κώδικα μέσω τετράδων. Η προσέγγιση είναι κανονιστική και ευθυγραμμίζεται πλήρως με τη μέθοδο που προτείνεται στο εκπαιδευτικό υλικό του μαθήματος, διασφαλίζοντας ότι η διαδικασία παραγωγής ενδιάμεσου κώδικα είναι ανεξάρτητη ανά κανόνα, καθαρή και επεκτάσιμη.

Κεφάλαιο 9

Παραγωγή Τελικού Κώδικα (Final Code)

Το τελευταίο στάδιο της μεταγλώττισης αφορά τη μετατροπή της ενδιάμεσης αναπαράστασης του προγράμματος σε τελικό κώδικα μηχανής, ειδικά σχεδιασμένο για την αρχιτεκτονική RISC-V. Η παραγωγή του τελικού κώδικα γίνεται με βάση τις τετράδες (quads) που έχουν δημιουργηθεί στα προηγούμενα στάδια και ακολουθεί αυστηρά το μοντέλο στοίβας που περιγράφεται στο μάθημα.

Ο κώδικας που παράγεται είναι πλήρως εκτελέσιμος στον προσομοιωτή Ripes και συμβατός με τις συμβάσεις (calling conventions και frame layout) της αρχιτεκτονικής. Η διαδικασία μετατροπής υλοποιείται μέσω του πακέτου `final_code`, το οποίο περιλαμβάνει τις απαραίτητες κλάσεις για την αναπαράσταση τελεστών (`Operand`), αριθμητικών σταθερών, προσβάσεων σε μεταβλητές, καθώς και τον γεννήτορα εντολών RISC-V.

Οι βασικές κλάσεις που συμμετέχουν σε αυτό το στάδιο είναι οι εξής:

- `Operand`: Αφηρημένη υπερκλάση που περιγράφει κάθε τι που μπορεί να χρησιμοποιηθεί ως τελεστής σε τελική εντολή (μεταβλητή, σταθερά, κ.λπ.).
- `IntConst`: Υλοποιεί ακέραιες σταθερές (`immediate values`) που χρησιμοποιούνται ως ορίσματα εντολών.
- `VariableOperand`: Περιγράφει την πρόσβαση σε μία μεταβλητή (τοπική, προσωρινή ή παράμετρο) και τον τρόπο υπολογισμού της διεύθυνσης.
- `RiscVAssemblyGenerator`: Είναι υπεύθυνη για τη μετατροπή κάθε τετράδας σε μία ή περισσότερες εντολές assembly, σύμφωνα με

το activation record του υποπρογράμματος και τη διαχείριση της στοίβας.

9.1 Αφηρημένη Αναπαράσταση Τελεστών: Operand, IntConst, VariableOperand

Η λογική πίσω από την ύπαρξη των Operand δεν αφορά την πολυπλοκότητα της γλώσσας ή την ανάγκη υποστήριξης πολλών τύπων δεδομένων, αλλά υπηρετεί έναν πολύ συγκεκριμένο σκοπό: να ενοποιήσει τη συμπεριφορά τελεστών κατά την παραγωγή τελικού κώδικα, ανεξαρτήτως του αν ο τελεστής είναι μεταβλητή ή σταθερά.

Στη φάση δημιουργίας του τελικού κώδικα (assembly), κάθε ορίσμα εντολής πρέπει να μετατραπεί σε κατάλληλη συμβολική ή αριθμητική μορφή (π.χ. offset από το sp, literal τιμή, προσωρινός καταχωρητής κ.λπ.). Όμως, η προέλευση κάθε operand μπορεί να είναι πολύ διαφορετική:

- **Αν πρόκειται για μεταβλητή**, αυτή έχει ήδη αποθηκευτεί στο *activation record* με προκαθορισμένο offset. Το offset αυτό είναι διαθέσιμο μέσα από την οντότητα τύπου Entity που επιστρέφεται από τον ScopeManager κατά την ανάλυση. Επομένως, αρκεί να γίνει σωστό resolve και ο operand να γνωρίζει πώς να παραγάγει τον κατάλληλο κώδικα (lw, sw) για προσπέλαση μέσω VariableOperand.
- **Αν πρόκειται για σταθερά**, τότε δεν υπάρχει καμία αντίστοιχη αποθηκευμένη θέση στο stack. Η τιμή "εμφυτεύεται" (injected) κατά την εκτέλεση και δεν χρειάζεται αναζήτηση στο scope. Τέτοιες τιμές υλοποιούνται ως IntConst και αποδίδονται άμεσα στον κώδικα με οδηγίες όπως li, addi κ.λπ.

Αν δεν υπήρχε η αφηρημένη κλάση Operand, ο RiscVAssemblyGenerator θα έπρεπε να κάνει συνεχώς instanceof ή if-else για να ξεχωρίζει τους τύπους των τελεστών. Με το παρόν σύστημα, κάθε τύπος operand υλοποιεί το δικό του toString() ή μεθόδους παραγωγής κώδικα, και ο γεννήτορας τελικού κώδικα μπορεί να λειτουργεί ενιαία και χωρίς εξαρτήσεις.

Η ιεραρχία έχει ως εξής:

- Operand: Αφηρημένη υπερκλάση που ορίζει τη διεπαφή όλων των operands.

- `IntConst`: Υλοποιεί ακέραιες σταθερές και επιστρέφει άμεσα την αριθμητική τιμή τους ως συμβολοσειρά.
- `VariableOperand`: Αναπαριστά μεταβλητές (τοπικές, παραμέτρους, προσωρινές) και χρησιμοποιεί offset για πρόσβαση στο `sp`.

Η δομή αυτή εξασφαλίζει ότι οποιοσδήποτε operand μπορεί να χρησιμοποιηθεί με ενιαίο τρόπο στις μεθόδους παραγωγής κώδικα, μειώνοντας δραματικά την πολυπλοκότητα και αυξάνοντας την επεκτασιμότητα του backend.

Κεφάλαιο 10

Παραγωγή Τελικού Κώδικα: RiscVAssemblyGenerator

Η κλάση RiscVAssemblyGenerator είναι υπεύθυνη για τη μετάφραση των τετραδών ενδιάμεσου κώδικα (intermediate code quads) σε εκτελέσιμο RISC-V assembly κώδικα, συμβατό με τον προσομοιωτή Ripes. Το κύριο χαρακτηριστικό της είναι ότι παρέχει αυτοτελή παραγωγή assembly κώδικα για κάθε εντολή, ακολουθώντας ταυτόχρονα τους κανόνες του RISC-V ABI και τη διαχείριση της στοίβας ανάλογα με το βάθος του scope.

10.1 Σκοπός και Αρχιτεκτονική

Η RiscVAssemblyGenerator δεν λειτουργεί ως απλός μετατροπέας συμβολοσειρών. Ο ρόλος της είναι να διατηρεί την αντιστοιχία ανάμεσα στο συμβολικό ενδιάμεσο επίπεδο (τετράδες) και το χαμηλό επίπεδο (assembly), λαμβάνοντας υπόψη:

- Το βάθος των scopes (ScopeManager),
- Τον τύπο του κάθε τελεστή (με χρήση των Operand, VariableOperand, IntConst),
- Την ανάγκη διαχείρισης καταχωρητών, προσπέλασης από διαφορετικά επίπεδα, και προσωρινών τιμών.

Ο assembly κώδικας παράγεται σταδιακά, κάθε φόρα που καλείται ο generator στο κλείσιμο κάθε scope, εκτελώντας κάθε φορά :

1. Επεξεργασία κάθε quad μέσω της generateAsmForCurrentScope(),

2. Επιλογή της κατάλληλης μεθόδου ανάλογα με τον τελεστή (`:=`, `+`, `call`, `retv` κ.λπ.),
3. Δημιουργία asm εντολών σε μορφή `String`, αποθηκευμένων στη λίστα `instructions`,
4. Τελική εγγραφή σε αρχείο μέσω `writeToFile()`.

10.2 Γεννήτριες Εντολών

Κάθε κατηγορία εντολών υποστηρίζεται από ειδική συνάρτηση:

10.2.1 Αναθέσεις και Αριθμητικές Πράξεις

Οι αριθμητικές πράξεις (`+`, `-`, `*`, `/`) μεταφράζονται μέσω των:

- `generateAsmForArithmeticOperation()`,
- `emitArithmeticOperation()`.

Ο γεννήτορας φορτώνει τους operands, χρησιμοποιεί τους καταχωρητές `t1` και `t2`, εκτελεί την αριθμητική πράξη, και αποθηκεύει το αποτέλεσμα.

10.2.2 Αναθέσεις `:=`

Η `generateAsmForAssigment()` υλοποιεί αντιγραφή τιμών. Αν ο τελεστής είναι σταθερά, εκτελείται απευθείας μεταφορά μέσω `li`. Αν είναι μεταβλητή, αναζητείται η θέση της στη στοίβα με βάση το βάθος.

10.2.3 Δομές Control Flow (Άλματα, Συνθήκες)

Οι άλματα υλοποιούνται μέσω:

- `emitJump()` για άλματα χωρίς συνθήκη,
- `emitConditionalJump()` για συγκρίσεις.

Εντολές όπως `beq`, `blt`, `bge` παράγονται ανάλογα με τον τελεστή.

10.2.4 Εκτύπωση και Είσοδος

Οι εντολές `out` και `in` μετατρέπονται με:

- `generateAsmForPrint()`,
- `generateAsmForInput()`.

Γίνεται χρήση των standard `ecall` με `a7` για I/O.

10.3 Υπορουτίνες και Διαχείριση Στοίβας

Η πιο απαιτητική λειτουργία του generator αφορά την υποστήριξη υπορουτινών και του activation record.

10.3.1 Έναρξη και Τερματισμός Υπορουτίνας

Οι `begin_block` και `end_block` υλοποιούν:

- Εκχώρηση stack space,
- Αποθήκευση / αποκατάσταση του `ra` (return address),
- Διαχείριση των τοπικών μεταβλητών.

10.3.2 Κλήση Συνάρτησης

Η `generateAsmForSubroutineBlock()` χειρίζεται:

1. Κατανομή stack για την καλούμενη ρουτίνα,
2. Αποθήκευση δυναμικού δεσμού,
3. Αντιγραφή παραμέτρων ανάλογα με τον τρόπο περάσματος (`by value` / `by reference`),
4. Άλμα στην αρχή της υπορουτίνας,
5. Αποδέσμευση του stack στο τέλος.

10.3.3 Παράμετροι

Οι παράμετροι υποστηρίζονται από:

- `emitParameterByValue()`,
- `emitParameterByReference()`.

Η διάκριση βάσει του `Parameter.Mode` εξασφαλίζει ότι ακόμα και αναδρομικές ή σύνθετες κλήσεις δουλεύουν σωστά με το σωστό `offset`.

10.4 Χρήση των Operand

Η κλάση χρησιμοποιεί τη δομή `Operand` για να διατηρεί ομοιομορφία στην παραγωγή assembly. Ανάλογα με τον τύπο:

- `IntConst`: Παράγει `li`,
- `VariableOperand`: Χρησιμοποιεί `lw / sw` με offsets,

ενώ η αναζήτηση μεταβλητών σε στοιβές γονικών scope χειρίζονται με `gnlvcode()` και το `TEMP_0` ως ενδιάμεσος καταχωρητής.