Greek++ Compiler

Christos Dimitressis 4351

cs04351@uoi.gr

May 2025

Contents

Chapter 1

Import

The purpose of this report is to present the implementation process of the compiler for the Greek++ language.

Specifically, the aim is to highlight the design choices and the programming approaches adopted, emphasizing in the way the implementation of the individual sections was structured and organized.

The presentation is divided into two main parts: the front part (frontend) and the backend. In each of these, the classes used, the interactions between them are analyzed and the logic behind their design and implementation. In addition, emphasis is placed on how to handle critical issues, such as management of the symbol table, the generation of intermediate code and the translation in RISC-V machine code.

This report is a guide to understanding the techniques and decisions applied during the construction of the compiler and not to repeat familiar theoretical concepts or procedures that are mentioned extensively in the course.

Chapter 2

Execution Flow (GreekPP.java)

The GreekPP class contains the main method and is the initial point of execution of the compiler. The flow of execution follows the logic of the basic phases of a compiler, with a strictly serial structure and distinct stages, such as lexical and syntactic analysis, the production of an intermediate representation, semantic checking and the production of a final code.

Execution begins by checking the command arguments. If none are given input file, execution is aborted with an appropriate error message.

- **1st Step – Verbal Analysis:** Type object is created CharStream that reads the input file, and this is passed to the Lexer. The Lexer converts the stream of characters into verbal units (tokens), which are provided to the Parser sequentially, one at a time, when requested.
- **2nd Step – Editorial Analysis & Editorial Creation Tree:** The Parser uses the Lexer to receive the tokens and construct the abstract syntax tree (AST), through of getABSRoot(). The analysis is done with guided transitions based on the grammar, and the tree produced represents the editorial structure of the program.
- **Step 3 [Part 1] – Intermediate Representation & Symbolism Analysis:** The IntermediateGenerator visits the abstract syntax tree and produces the intermediate form of the program in quads. At the same time, it creates and fills the symbol table with the declarations of variables, parameters and subroutines. Although it does not perform full semantic checking, it incorporates basic checks and prepares data for production final code.

- **Step 3 [Part 2] – Final Code Generation:** The final file in RISC-V Assembly is generated by the AsmManager, which is called by the IntermediateGenerator. The final code is not generated in a centralized manner at the end, but is generated gradually during the recursive visit of the syntax tree. Specifically, each time the processing of a scope is completed, the corresponding assembly instructions related to it are generated and stored.

The overall architecture of the compiler follows an object-oriented approach, aiming at a clear distinction of phases and low coupling between them. Each stage, from lexical analysis to final code generation, is implemented through independent classes such as Lexer, Parser, IntermediateGenerator, RiscVAssemblyGenerator, which communicate only through defined interfaces and objects such as (e.g. ASTNode, Quad, ScopeManager). This structure promotes modular design and facilitates the extension or maintenance of the compiler.

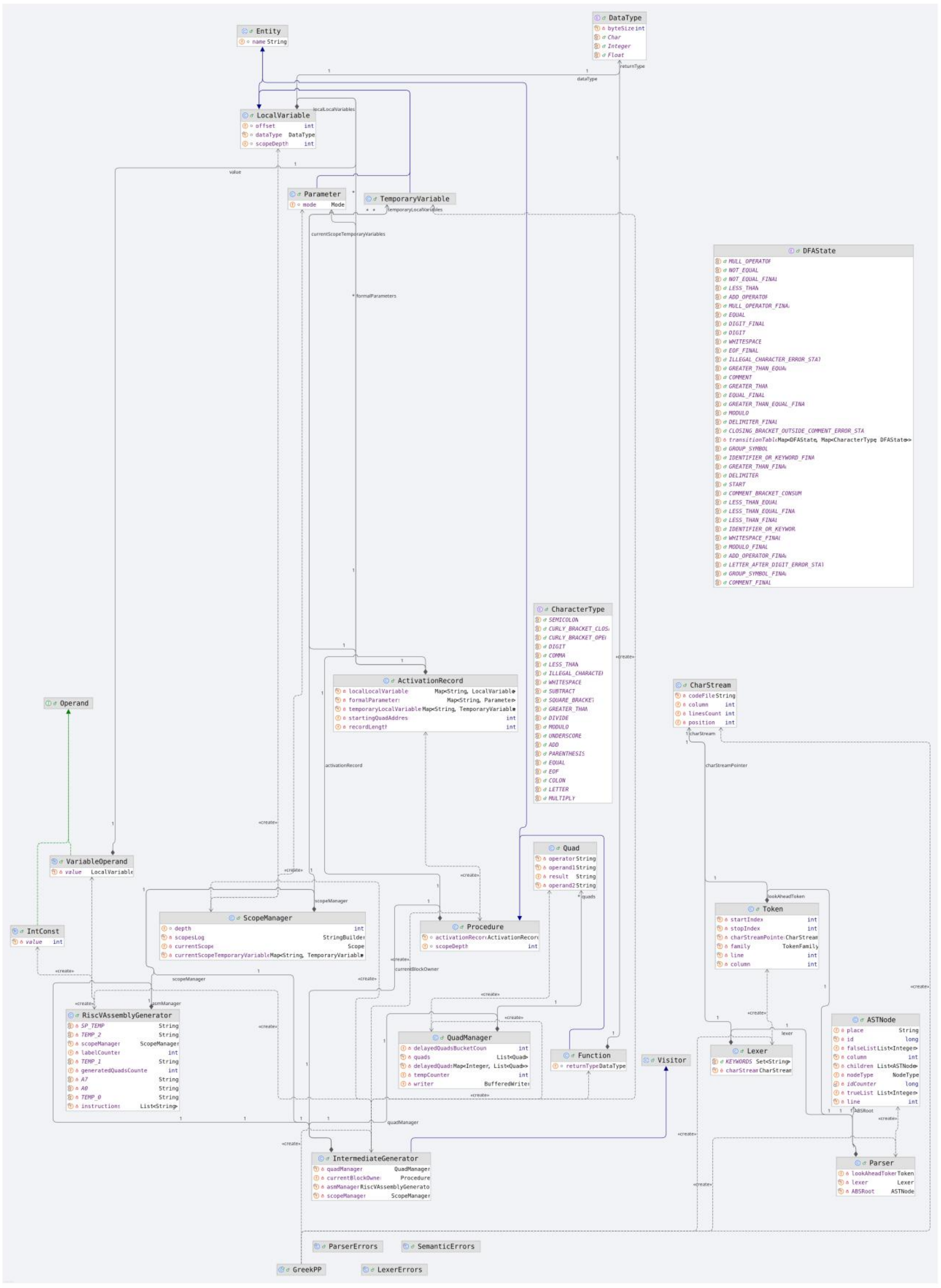


Figure 2.1: Diagrammatic representation of the main classes of the Greek++ compiler

Part I

Front-End

Chapter 3

Lexical Analyzer (Lexer)

The implementation of the lexical analyzer is based on five main classes, which are located in the lexer folder, each with a separate role in the analysis of the input in verbal units (tokens):

- Lexer.java: The central class that implements the parsing mechanism using the finite automaton (DFA), and provides the method for generating tokens.
- CharStream.java: Responsible for reading the character stream from the input file, offering lookahead and character consumption methods.
- CharacterType.java: Defines the categorization of input characters as an enum type, which is used as input to the DFA.
- Token.java: Represents the verbal units produced by the Lexer, storing information such as type, position, and recognized alphanumeric.
- DFAState.java: Describes the states of the finite automaton, along with the transitions between them.

3.1 CharStream

CharStream is the first class activated at runtime
of the compiler, constituting the initial interface between the source
file and the verbal analysis process. It is applied to the first
step of the main program, as shown in the GreekPP class, and
provides Lexer with the ability to securely and accurately access
the characters of the file.

Specifically, CharStream loads the input file once into a
String, allowing efficient random access to characters. The
This String essentially works like a big array of characters,
from which the Lexer reads with positional pointers, without copying or overhead.
It provides the peekNextChar() method, which returns the next character without
consuming it, and the consumeNextChar() method,
which advances the flow and updates the position information (column and
line).

Additionally, the class tracks the current position of the parser in the text via the
getPosition(), getColumn() and getLinesCount() methods, allowing the generation
of accurate diagnostic messages. It provides
also the getText(start, stop) method, which is used to
extracting substrings and reconstructing the recognized
Token.

CharStream is designed to work directly with Lexer's finite state automaton
(DFA), ensuring that the input is parsed with character-by-character accuracy
and lookahead capability.
It thus constitutes the basis upon which the function of verbal analysis is built.

3.2 CharacterType

CharacterType is an enum class that implements the categorization of all
possible input characters, to support clean,
secure and structured analysis of them by the lexical analyzer. It determines
the type of each character and replaces the use of vague
if-else or switch within the lexical analyzer itself.

Its role is not limited to facilitating the reading of characters: the CharacterType
serves as **input to the finite automaton** (DFA) implemented by the Lexer.
Specifically, each character that
read from the CharStream is converted to a CharacterType, which is passed to
the getNextState() method of the current state of the DFA. Thus,
The implementation of the verbal analyzer itself is based on the definition of the al-

DFA character set as a set of character types instead of individual ones
characters.

Using this interface between characters and situations offers the following advantages:

- Reduces the number of transitions to automatic.
- Separates low-level character analysis from high-level
  Low-level logic for recognizing verbal units.
- Facilitates the handling of exceptions, such as illegal characters
  you know.

The supported categories are as follows:

- LETTER: all Latin and Greek letters, with or without accent.
- DIGIT: numeric digits 0–9.
- WHITESPACE: characters such as space, tab, newline, carriage return.
- ADD, SUBTRACT, MULTIPLY, DIVIDE, MODULO: arithmetic operators.
- PARENTHESIS, SQUARE_BRACKET, CURLY_BRACKET_OPEN, CURLY_BRACKET_CLOSE: ÿÿ-
  all types of hinges.
- LESS_THAN, GREATER_THAN, EQUAL, COLON, SEMICOLON, COMMA, UNDERSCORE: ÿÿÿÿ-
  comparison, separation and composition engines.
- EOF: special value for end-of-input recognition (character -1).
- ILLEGAL_CHARACTER: for any character that does not belong to any
  from the above categories.

The presence of ILLEGAL_CHARACTER ensures the robustness of the parser, since it
allows for the timely detection and reporting of errors during
processing characters that are not part of the language.

3.3 Token

Token Family

TokenFamily is an enum type that groups tokens into basic categories, making it easier
to sort them during the process.
analysis. The main categories include:

- KEYWORD: Keywords of the language.
- ADD_OPERATOR: Addition and subtraction operators.
- MUL_OPERATOR: Multiplication, division, etc. operators.
- REL_OPERATOR: Relational operators (e.g. <, >, =).
- REFERENCE_OPERATOR: Reference operators.
- DELIMITER: Separators (e.g. comma, colon).
- GROUP_SYMBOL: Grouping symbols (e.g. parentheses).
- IDENTIFIER: Identifier names.
- NUMBER: Numeric constants.
- EOF: End of file signal.
- OTHER: All other uncategorized tokens.

The use of TokenFamily helps to homogenize the processing of tokens and allows for easy separation of different types of lexical units within the compiler.

Token Class

The Token class represents a token, i.e. an unrecognized symbol of the Greek++ language that has been recorded by the lexical analyzer. Each Token object includes information about the category of the token, the exact code fragment that was recognized, as well as its position in the file: both the line and column, as well as the start and end indicators in the CharStream, which correspond to the characters that make it up.

When creating a token, the final state of the DFA is taken into account, which determines its category (TokenFamily), as well as the start and end indexes of the recognized fragment. The Token class does not store the string itself, but dynamically retrieves the recognized text via the CharStream, holding start and end indexes.

The classification of tokens is based on the enum TokenFamily, which includes categories such as keywords, identifiers, operators, numbers, grouping symbols, etc. In the case of alphanumerics, the assignment of the category is decided dynamically based on the final state of the DFA and the recognized text.

3.4 DFAState

The DFAState class implements the states of the finite automaton (DFA) used in the lexical analyzer for the recognition of lexical units. It is implemented as an enum, allowing direct and standardized management of the automaton's logic through named states, without the use of numeric or string literals.

3.4.1 Transition Table

The set of transitions between states is implemented statically through a transition table (transitionTable), which is of type EnumMap<DFAState, EnumMap<CharacterType, DFAState>>. The choice of use EnumMap offers excellent performance (O(1) access) and reduces the probability of errors compared to more general structures, such as HashMaps or switch-case chains.

That is, for each state from of the automaton, a separate EnumMap<CharacterType, DFAState> object is maintained, which describes all possible transitions of the form CharacterType ÿ nextState. In this way, each state has its own local transition table. Even error states are included in the

table, so that all possible states have certain transitions, ensuring the completeness of the DFA.

The initializeTransitionTable() method automatically populates the transition table, recording the next state for each state and character type pair. The logic covers all character types (CharacterType) and provides:

- Transition from START to DIGIT, IDENTIFIER_OR_KEYWORD, ADD_OPERATOR, etc. depending on the input character.
- Internal transitions that allow repetitions (e.g. consecutive digits or letters).
- Identification of final states through separate _FINAL states of.
- Support for complex operators (<=, !=, >=) and comments with opening/ closing brackets.
- Error detection, with special situations such as ILLEGAL_CHARACTER_ERROR_STATE and CLOSING_BRACKET_OUTSIDE_COMMENT_ERROR_STATE.

3.4.2 Methods

The getNextState(CharacterType), isFinal() and triggersDFARestart() methods are the basic interaction mechanism between Lexer and of the DFA. The getNextState(CharacterType) method updates the lexical analyzer for the next state of the automaton, based on the type of character read and the state in which the automatic at the time of the call. This type is determined through the CharacterType.of(char), and the transition is looked up in the transition table transitionTable.

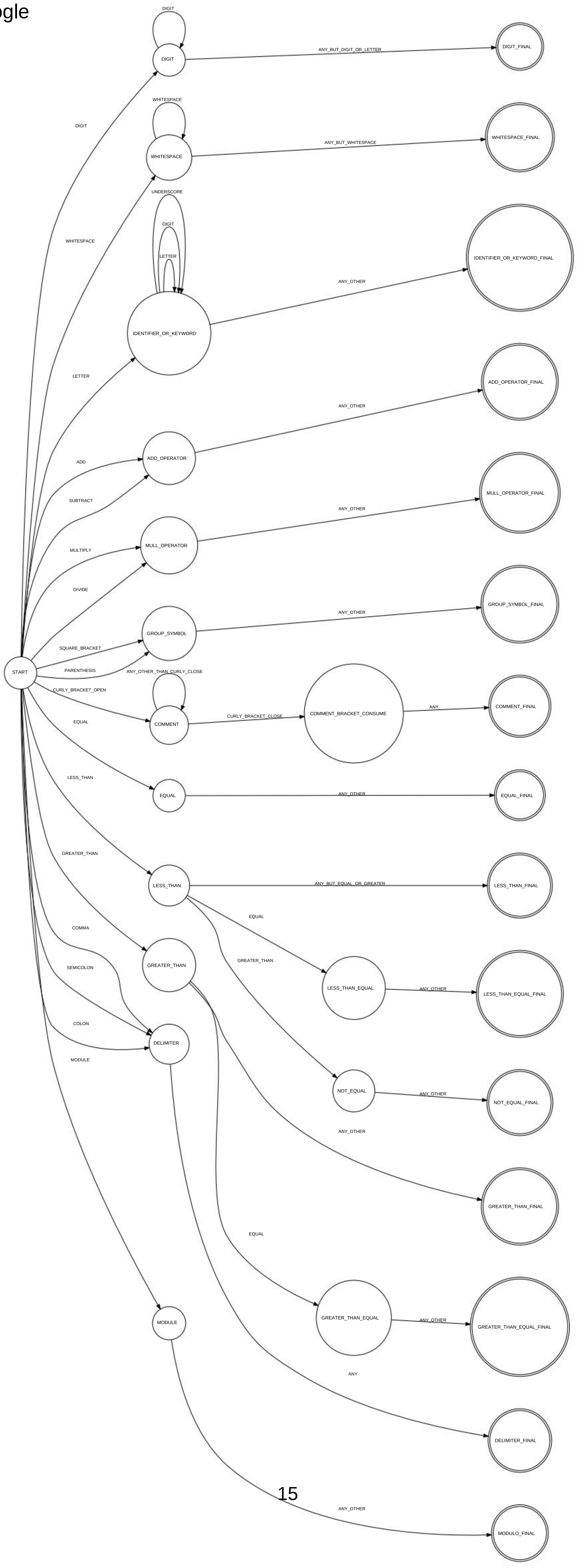
If in a given situation a character comes who could not lead to the creation of an acceptable token for our language is considered unauthorized entry and leads to the explicit transition to the special status ILLEGAL_CHARACTER_ERROR_STATE. In this way, DFA can immediately detect any unrecognized or unsupported symbol, without require exceptions or complex checks within the Lexer.

The isFinal() method indicates to the Lexer whether the current state is final, i.e. if the recognition of a verbal unit has been completed and a Token can be created. The final states are declared explicitly with the suffix _FINAL and correspond to recognizable categories of the language.

The triggersDFARestart() method signals that a DFA state — although final — does not require Token generation. This is true, for example, in cases such as whitespace or comment recognition. When the method returns true, it signals that the verbal unit can be ignored and the automatic must restart the analysis from the its initial START state, continuing with the next character. The This logic allows for the natural and clean filtering of non-semantic elements of the language already at the automaton level, keeping the DFA output free of redundant units.

Designing using enum and transition table via EnumMap makes the parser is easily extensible and simple to maintain. Adding new rules only requires the introduction of additional transitions in the table, without complication in the Lexer code.

Figure 3.4.2: The graph of the finite automaton (DFA) of verbal analyzer.


DIGIT
DIGIT
ANY_BUT_DIGIT_OR_LETTER
DIGIT_FINAL
DIGIT
WHITESPACE
WHITESPACE
ANY_BUT_WHITESPACE
WHITESPACE_FINAL
WHITESPACE
UNDERSCORE
DIGIT
LETTER
IDENTIFIER_OR_KEYWORD
ANY_OTHER
IDENTIFIER_OR_KEYWORD_FINAL
LETTER
ADD_OPERATOR
ANY_OTHER
ADD_OPERATOR_FINAL
ADD
SUBTRACT
MULL_OPERATOR
ANY_OTHER
MULL_OPERATOR_FINAL
MULTIPLY
DIVIDE
GROUP_SYMBOL
ANY_OTHER
GROUP_SYMBOL_FINAL
SQUARE_BRACKET
PARENTHESIS
START
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
SEMICOLON
LESS_THAN_EQUAL
ANY_OTHER
LESS_THAN_EQUAL_FINAL
COLON
DELIMITER
MODULE
NOT_EQUAL
ANY_OTHER
NOT_EQUAL_FINAL
ANY_OTHER
GREATER_THAN_FINAL
EQUAL
GREATER_THAN_EQUAL
ANY_OTHER
GREATER_THAN_EQUAL_FINAL
MODULE
ANY
DELIMITER_FINAL
ANY_OTHER
MODULO_FINAL

3.5 Lexer

When constructing the Lexer, it accepts a CharStream object as an argument, which allows it to access input text flexibly, using lookahead characters, backspace, and precise
position reference (row, column).

getNextToken() method

The most critical method of the class is getNextToken(), which implements
the basic loop for identifying the next Token via DFA:

1. It starts from the initial state START and records the starting position of the token.
2. Each step looks up the next character (without consuming si), and converts it to a character type via CharacterType.of(char).
3. The current state of the DFA is updated based on the character type previously looked up via getNextState(),
returning the new status.
4. If the new state is an error — that is, when the DFA transitions in an error state based on the input character and the current state — the LexerErrors.IllegalStateTransition() method is called,
which stops execution and displays the corresponding message.
5. If the new state requires a restart (triggersDFARestart()), then the token start pointer is reset, the verbal unit is ignored
(e.g. whitespace or comment) and continues from the beginning.
6. If the new state is **final** (isFinal()), the recognition is completed and a Token is constructed with its information.

   Note: final states *do not consume* the *lookahead* character, i.e. the character that caused the transition to the final state is not subtracted from the CharStream.

   This is because to get to a final state, means that a character has been read that *does not belong* to the token recognized at that moment.

   Therefore, **this character should not be consumed,** so that it remains available as *the first character of the next one. available token.*
7. If none of the above applies, the character is consumed through consumeNextChar() and the loop repeats.

The method guarantees that each call to it produces either a valid Token or terminates execution with an error, maintaining the strictness of the literal analysis. Special treatment exists for the EOF character, which which is considered a regular character in the CharacterType enum and leads to the construction of the corresponding special token. In this way, the verbal analysis process remains consistent and universal, as EOF is treated as an explicit end-of-input signal. The EOF token acts as a completion indicator and is usually ignored during parsing, while also helping to detect the "EOF reached" error while parsing" when the input terminates unexpectedly.

Interacting with the Parser

The Lexer is the sole source of tokens, which are consumed by the Parser. The communication between them is simple but fundamental: the Parser repeatedly calls getNextToken() to receive the next available Token and examine it against the grammar. This form of lazy generation ensures that the analysis is done step by step, without requiring a complete pre-analysis of the program.

Proper cooperation between the Lexer and the Parser is critical for the success of the syntactic analysis. This architecture is based on in the concept of **on-demand** tokenization, where each token is produced only when requested, allowing lookahead and backtracking parsers.

Keywords

The Lexer includes a static set KEYWORDS, which is used to separate regular identifiers from keywords.
When a token syntactically matches an identifier, Token checks whether this is contained in KEYWORDS and returns the corresponding category.

This set includes all the reserved words of the language Greek++, such as program, if, for, procedure, and, not, etc.

Chapter 4

Parser

Parsing is a critical stage of the compiler, where the sequence of tokens produced by the Lexer is converted into a hierarchical structure that represents the program. In the Greek++ language, parsing is implemented mainly through two basic classes located in the parser folder:

- Parser.java: Implements the parsing mechanism, recognizing the syntax of the language based on the grammar and creating the Abstract Syntax Tree (AST).
- ASTNode.java: Represents the nodes of the abstract syntax tree, allowing structured storage and processing of the syntactic structure.

The Parser uses the tokens produced by the Lexer to construct the AST, which forms the basis for the next stages of the compiler, such as semantic checking and code generation.

4.1 ASTNode.java

The ASTNode class represents a node in the Abstract Syntax Tree (AST) produced by the Parser. It is
the basic structural unit used to store and represent the hierarchical syntactic structure of the program in
Greek++ language.

Each ASTNode contains information about:

- the type of node (e.g. statement, expression, command),
- its identifier (optional),
- its parents and descendants, allowing for tree formation and AST tour,
- additional metadata required for semantic analysis solution and translation.

The class structure supports extensibility, allowing easy addition of new node types and features as the language and compiler evolve.

The ASTNode class is the foundation for all subsequent stages of meta-translation, such as semantic analysis and the production of the final text. code.

4.2 Parser.java

The Parser class is the core of the parsing in the Greek++ language. Its role is to take as input the sequence of
tokens generated by the Lexer, through the getNextToken() method, and recognize the program syntax according to the rules of
grammar of the language.

The analysis is implemented using the recursive descent technique. descent parsing), where each syntactic rule of the grammar corresponds to a separate method of the class. This mechanistic structure
makes the code clean, self-explanatory and easily extensible,
facilitating the maintenance and addition of new language structures.

During execution, the Parser sequentially consumes the tokens produced by the Lexer, calling the corresponding methods that implement the grammatical productions, gradually creating the abstract syntax.

Abstract Syntax Tree (AST). The AST is a hierarchical, tree-like structure that clearly and structurally represents the syntax and logic of the program in a form suitable for subsequent semantic analysis.
editing and code generation.

The use of recursive descent allows for the natural and efficient handling of complex language constructs, such as expressions, commands
flow control, loops and subroutines (functions and procedures), closely following the structure of the grammar.

Additionally, the Parser incorporates mechanisms for detecting and handling syntax errors. During analysis, if an error is detected, its location (line, column) and nature are recorded precisely.
error, providing constructive feedback that facilitates
the process of debugging the code by the programmer.

The grammar of the Greek++ language is listed below:

University of Ioannina
Department of Informatics
"Translators"
Teacher: G. Manis
February 2025

Greek++ grammar

```
program             :     'program' programblock ID
                    ;

program block       : declarations subprograms
                          'beginning_of_program' sequence 'end_of_program'
                    ;

declarations        :     ( 'declaration' varlist )*
                    |
                    ;

varlist             : ID ( ',' ID )*
                    ;

subprograms         :     ( function | proc )*
                    ;

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

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

formalparlist       : varlist
                    |
                    ;

funcblock           : 'interface' funcinput funcinoutput declarations
                          subprograms
                          'function_begin' sequence 'function_end'
                    ;

procblock           : 'interface' funcinput funcinoutput declarations
                          subprograms
                          'beginning_of_process' sequence 'end_of_process'
                    ;

funcinput           : 'input' varlist
                    |
                    ;

funcoutput          :     'exit' varlist
                    |
                    ;
```

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

statement           : assignment_stat
                          | if_stat
                          | while_stat
                          | to_stat
                          | for_state
                          | input_stat
                          | print_stat
                          | call_stat
                          ;

assignment_stat     : ID ':=' expression
                    ;

if_stat             :    'if' condition 'then' sequence elsepart 'if_end'
                    ;

elsepart            :    'otherwise' sequence
                    |
                    ;

while_stat          :    'as long as' condition 'repeat' sequence 'as_long as'
                    ;

to_stat             :    'repeat' sequence 'until' condition
                    ;

for_state           :    'for' ID ':=' expression 'to' expression step

                            'repeat' sequence 'to_end'
                    ;

step                :    'with_step' expression
                    |
                    ;

print_stat          :    'write' expression
                    ;

input_stat          :    'read' ID
                    ;

call_stat           : 'execute' ID idtail
                    ;
```

```
idtail              : actualpars
                    |
                    ;

actualpars          :       '(' actualparlist ')'
                    ;

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

actualparitem       : expression | '%'ID
                    ;

condition           : boolterm ( 'ÿ' boolterm )*
                    ;

boolterm            : boolfactor ( 'ÿÿÿ' boolfactor )*
                    ;

bool factor         :       'no' '[' 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            :       '+' | '-'
                    ;

many_opers          :       '*' | '/'
                    ;

optional_sign       : add_oper
                    |
                    ;
```

Part II

Back-End

The backend of the Greek++ compiler is organized into three basic subcategories, which correspond to the important stages of the translation process after syntactic analysis:

- **Final Code (final_code):** Includes the classes and operations related to the production of the final code, i.e. the conversion of the intermediate representation into assembly code for the target (RISC-V).
- **Quads:** It concerns the management of the intermediate program form through quads, which constitute a simple and standardized representation of instructions, facilitating optimization and the final code generator.
- **Semantic Control:** Includes the classes that implement semantic correctness checks, such as symbol table management, type checking, and other rules which concern the meaning of the program and not its syntax.

The central orchestrator of the backend is the IntermediateGenerator class, which coordinates the communication between the individual subsystems. Specifically, the IntermediateGenerator visits the abstract syntax tree (AST), generates the quads, manages the symbol table, and calls the appropriate managers to create final code.

This structure ensures a smooth flow of information and a clear division of responsibilities, allowing for easier maintenance and scalability. compiler.

In subsequent chapters, each of these three subcategories will be analyzed separately, presenting the basic classes and functions. that they implement, as well as their importance in the overall functioning of the Greek++ compiler.

Chapter 5

Quads

In the backend of the compiler, **quads** are the intermediate representation of the program, which is more abstract than the final machine code, but detailed enough to support the production of the final code.

This intermediate form is implemented through two basic classes:

- Quad: Represents an intermediate code unit, the quad, which consists of one operator and up to three arguments.
- QuadManager: Manages the collection of quads, providing functions for adding, accessing and managing the list of quads throughout the intermediate code generation.

The QuadManager class functions as the central manager of the intermediate representation, while the Quad is the basic unit that stores the functions corresponding to program commands.

5.1 Quad Class

The Quad class implements the intermediate command, which consists of:

- an **operator** that specifies the action or command,
- up to three **arguments** (operands) which are the parameters of the command.

We call the last argument result and consider it the registered "target" in every intermediate code instruction where this concept makes sense.

The class provides methods to access and modify the above
fields, as well as methods for displaying and debugging the notebook.

Using Quad allows the representation of complex language constructs with simple
commands, facilitating backend design.

5.2 QuadManager Class

The QuadManager class is responsible for managing the list of quads produced
during the intermediate phase of the compiler. It takes care of adding new quads,
updating them, and
management of position numbers (quad numbers), retrieval of quads
based on their location, as well as supporting functions such as
backpatching for fixes to code addresses.

This class is a critical part of intermediate code generation, providing structure
and flexibility for modification and organization.
of the code generated.

5.2.1 Delayed Quads and Recursive Generation

An important feature of the QuadManager class is its support
delayed quads — i.e. quads that are not registered immediately
in the main list, but their registration is delayed. This is necessary in cases of
procedure or function calls, especially when the
arguments of calls are expressions that must first be evaluated
from other syntactic rules of the language. Because of the recursive
nature of syntactic analysis, the corresponding quaternions are not created in
linear order, and thus temporary storage is required
and their subsequent introduction.

In practice, during procedure or function calls, it is created
a HashMap<Integer, List<Quad>> where the key is the scope level
moment the call is made (depth level) and the value is the list of
quads produced at this level.

Because essentially the quads produced in a call and concern the
variables belong to a different scope than the quads they pertain to
the function call or the quad for the return value, this is done
management.

Chapter 6

Symbol Table (ScopeManager)

The ScopeManager class implements the mechanism for managing scopes (fields of visibility) and symbols (variables and subroutines) within the Greek++ compiler. It is a crucial part of semantic analysis, as it allows for declaration, finding and validity of variables and functions at different levels of the program.

Scopes are managed through the internal Scope class, which maintains a symbol table (SymbolTable) that includes:

- local variables,
- and subroutines (functions or procedures).

The class supports a hierarchical tree of scopes, with each scope being knows its parent, allowing name detection and resolution across all visibility environments.

6.1 Functions and Uses

ScopeManager provides functions for:

- opening (openScope()) and closing (closeScope()) scopes, maintaining- given the current scope and its depth,
- adding local variables and subroutines to the current program scope,

- searching for variables and subroutines throughout the scopes chain (with tracing in parent areas),
- recording and storing scope history for debugging and control,
- recording and managing temporary variables in each scope,
- issuing error messages for declarations and calls that violate declaration and usage rules.

6.2 Structure and Implementation

6.2.1 Inner Class Scope

The inner Scope class represents a level of visibility and includes:

- the parent scope (unless it is global),
- and the symbol table with the variables and subroutines that belong to it.

The symbol table (specifically the SymbolTable class) organizes symbols into two Maps: one for variables and one for subroutines.

6.2.2 Symbol Management

The basic methods of ScopeManager for symbol management are:

- addSubroutine(Procedure): Adds a new symbol to the symbol table for a function or procedure. Checks for duplicate declarations based on the routine signature.
- declareVariable, declareParameter, declareFunction, declareProcedure: Implement the declaration of new symbols with validity checking and assignment to the appropriate activation record.
- resolveVariable(String) and resolveSubroutine(String): They look for a variable or subroutine starting from the current scope and ascending to parent scopes to find the name.

6.2.3 Managing Temporary Variables

The ScopeManager maintains a separate map of temporary variables (TemporaryVariable) for the current scope. These are used during generation of intermediate code for temporary storage of values and results of intermediate calculations. Intermediate code for each scope is generated every time from the inside out. Therefore, each scope entity does not need to have a structure for temporary variables.

6.2.4 Scopes History Recording

When closing a scope, the description of the scopes is recorded in scopesLog. variables and subroutines it contained, along with the depth of scope. This log file helps in debugging and analytical
monitoring semantic analysis.

Chapter 7

Symbol

In the symbol package we manage all the entities we store
in the symbol table of each scope, i.e. variables, parameters,
functions, procedures and temporary variables. These classes
they provide the basis for semantic analysis, type checking, and stack organization
during final code generation.

The central entity that orchestrates and connects the symbolic elements to the execution environment is the
ActivationRecord, which corresponds to the active execution frame (activation frame) of each subprogram.

7.1 ActivationRecord

The ActivationRecord class represents the activation frame.
of a subroutine during execution or during code generation.
It includes detailed information about all local variables, parameters, and
temporary variables, as well as their offsets on the stack.

This class is responsible for calculating and managing the
offsets and the total length of the frame, to ensure the
correct access to variables during execution.

7.2 DataType

The DataType class defines the basic data types of the language.
Greek++, like Integer. It provides information such as the size in bytes,
which is necessary for placing variables in the active context.

The existence of this class allows for standardized type management
and facilitates the implementation of type checking and code generation.

7.3 Entity

The abstract Entity class is the basic superclass for all symbols that may appear in a program (e.g. variables, parameters, functions, procedures). It defines common characteristics such as
the name and type.

The use of this class allows for consistent handling and extensibility in the definition of symbols.

7.4 Function

The Function class is a special case of Procedure and represents functions that return a value. It includes additional fields and methods.
concerning the return of value and the management of the corresponding activation framework.

7.5 LocalVariable

The LocalVariable class represents local variables within a scope.
Contains information such as name, type, and offset to activation
record, necessary for memory placement and access.

Additionally, it contains the ScopeDepth of the variable upon its creation.
so that in the final code phase it can be identified in which stack the activation record is stored.

7.6 Parameter

The Parameter class is a subclass of LocalVariable and represents parameters of subroutines (input or output). It includes
additional features that facilitate the handling of the mechanism
passing parameters.

7.7 Procedure

The Procedure class represents procedures (subprograms that do not return a value). It includes elements such as the name, parameters
and the active framework it manages.

7.8 TemporaryVariable

The TemporaryVariable class represents temporary variables used during intermediate code generation. These are variables that are not defined directly by the user but are created
to support compiler calculations and conversions.

It includes mechanisms for managing names, types and offsets so that it is properly integrated into the activation record and supports the
code generation.

Chapter 8

Intermediate Representation (Intermediate Code)

Intermediate Code is a crucial stage in the Greek++ compiler. Its purpose is to transform
the abstract syntax tree (AST) in a more simplified, standardized form, which facilitates semantic checking and
final code generation.

The intermediate representation used is based on **quads ,** which describe basic language commands in
three-four operand form (e.g. operator, left and right operand, and receiver).

8.1 Visitor

The Visitor class defines the visitor pattern interface for browsing.
in the AST. It includes visit methods for each node type (e.g.
statement, expression, command, subprogram).

The IntermediateGenerator implements this interface, defining exactly what should happen on each node visit, i.e. how the
intermediate code for each linguistic structure.

8.2 IntermediateGenerator

The IntermediateGenerator class acts as the central orchestrator of intermediate representation. It uses the visitor pattern to visit the nodes of the abstract syntax tree

(AST) and produce the corresponding quaternions, update the table symbols and cause the production of final RISC-V code.

The class structure includes the following basic fields:

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

- quadManager: Manages the collection of generated quads (intermediate code).

- scopeManager: Responsible for opening and closing scopes, managing variables, parameters and subroutines, as well as the recording of the activation record.

- asmManager: Used to produce final RISC-V assembly code for each scope separately.

- currentBlockOwner: Refers to the subroutine (e.g. main program, procedure, function) that is the **owner of the current scope** and to which the variables, parameters and temporary variables are assigned.

Its basic functions include:

- **Creating quadrats:** Converting statements, expressions, e- of tasks and subprograms in corresponding quadruplets.

- **Scope management:** Integration of ScopeManager for the correct management of variables and subprograms at the scope level.

- **Management of temporary variables:** Allocation and use of pre-temporary variables for intermediate results.

- **Function call handling:** Implementing the logic for function and procedure calls, including delayed quads for handling recursive calls.

- **Code generation for control structures:** Support for commands if, while, for, etc. with appropriate handling of labels and branches.

8.3 Basic Fields of ASTNode for Intermediate Code

When generating the intermediate code, each node of the abstract syntax tree (ASTNode) does not simply function as a structural element of the program, but also conveys critical information that is necessary for the creation of the tetrads and the control of the flow of program.

The three main fields used by the compiler to this purpose is:

- place: Records the location (variable name or temporary value) in which the result of the node evaluation is stored. It is mainly used for numerical and expressive structures.
- trueList: Contains a list of addresses of quads that must be backpatched if the condition associated with the node evaluates to true. Used in logical expressions and flow control conditions.
- falseList: Correspondingly, it records the addresses of the quads that must be corrected in case the condition is false.

These fields constitute the basic communication mechanism between the nodes of the AST and the code generation subsystem. The presence and their correct use is critical for the correct linear representation of the program through quads and the support of complex flows control.

8.4 Indicative Production Flow Description for Selected Rules

This section presents the basic flow of intermediate code generation for some characteristic syntactic rules of Greek++ language. The description focuses on the logic of processing of the abstract syntax tree (AST), in the production of the corresponding quadrats and in the use of the auxiliary fields place, trueList and falseList depending on the nature of each rule.

Edit numeric expression (visitExpression)

The method that processes an arithmetic expression implements the logic of the rule:

Expression ::= OptionalSign Term (AddOp Term)*

The method operates in six consecutive steps:

1. We start by reading the first subnode of the expression, which corresponds to the optional sign (+ or -). We visit it, that is we call the visit method of the corresponding rule (visitOptionalSign) and we wait for it to complete its analysis and fill the place field of the node with the corresponding information (e.g. ”-” or nothing).

2. Then we retrieve the second subnode, which is the first numerical term (Term). We visit it in a similar way: the corresponding method (visitTerm) is called which will recursively visit all its subnodes (e.g. Factor, Number, Variable, etc.), so that the evaluation of the term is completed and the variable containing the result is stored in the node's place This is essentially the first calculated value of the arithmetic expression.

3. If the sign was negative, then the appropriate conversion operation is applied. Specifically, a quaternion of the form is created (-, 0, T1, temp), which subtracts the term from zero, and a temporary variable is created for the result. This new variable replaces the value of the first term, so that it can be used as based on the remaining calculations.

4. Next, we examine whether there are additional operators and arithmetic terms in the expression (e.g. + T2, - T3, etc.). If so, for each pair (operant, condition) that follows:

   - We retrieve the operator node and visit it. This has the effect of filling the node's place with the operator character ("+" or "-").
   - We then retrieve the next arithmetic term and visit it, so that it is fully evaluated and the result is returned. variable containing its result.
   - A new temporary variable is created to store the result of the action.
   - A quadruple is produced that performs the operation of the operator between at the previous price and the new term.

- The result of the operation replaces the previous value and is used as the basis for the next iteration.

5. The above process is repeated for each additional addition or subtraction to the expression. The intermediate result is stored each time in a new temporary variable and fed back to the next step.

6. When all elements of the expression have been processed, the final variable containing the result of the evaluation is placed in the place field of the Expression node, so that it is available to the parent rule.

Edit if statement (visitIfStatement)

This method implements the intermediate code corresponding to an if – else structure. The processing is performed step by step as follows:

1. We start by reading the subnode containing the if condition. We visit this node, that is, we call the visit method corresponding to the logical expression rule. The evaluation does not return a numerical result but two lists:

   - trueList: contains the quads in which the execution address must be filled in if the condition is true.

   - falseList: contains the quads that will be executed when the condition is false.

   The quadruplets in these lists have an open destination (i.e. jumps without a target) and require subsequent completion with the backpatch() procedure.

2. Immediately after, we know what the next available command is — i.e. the first one in the body of the if. Therefore, we perform **backpatching on the trueList,** so that:

   - All the quadruplets of trueList should be updated by jumping to body of if.
   - Execution continues normally within the structure if the condition is true.

3. Then we visit the node corresponding to the body of the if, the sequenceNode , i.e. the instructions that are executed when the condition is true. These instructions are processed recursively and the corresponding intermediate code is generated.

4. After the if body is processed, we create a jump command without a target, so that:
   - Skip the else when the condition is true.
   - The location of this jump statement is temporarily stored in an ifList, to be filled in later.

5. Before the else processing begins, we perform **backpatching on falseList,** that is:
   - All the 4s in falseList are updated to jump to the first statement in the else-block.
   - So, if the condition is false, execution will jump correctly

     in else.

6. We visit the else node, i.e. the statements that are executed when the condition is not true. As before, the content of the node is processed recursively.

7. Finally, backpatch() is executed on the ifList list, so that the jump we created earlier (after the ifList) correctly points to the first statement after the end of the entire structure. This ensures that, if the condition was true, the program will not execute the else as well.

The **backpatching** technique allows for dynamic handling of jumps. The trueList and falseList lists act as temporary structures that hold “pending” jumps, which are completed once we know their exact destination. Thus, the intermediate code maintains the correct logical flow regardless of whether the condition is true or not.

Edit for statement (visitForStatement)

This method implements the for loop, taking into account the dynamic direction of the loop (ascending or descending), depending on the sign of step. The generated intermediate code is dynamically adapted at runtime based on this value. The detailed description of the steps follows:

1. We read the node identifier (ID) that represents the loop's control variable, as well as the expressions:
   - expression1: initial value of the variable.
   - expression2: final value (limit).
   - stepNode: value of the step.
   - sequenceNode: the body of the for.

2. We visit expression1 and the quadratic :=, expr1, –, ID is produced, which sets the initial value to the control variable.

3. We then visit expression2 and stepNode to determine the values of the upper bound and the step.

4. We create a branching structure to determine at runtime whether the step is positive or negative:

   • A quadruple >=, step, 0, – is produced. If true, the logic for ascending iteration will be followed . • A jump is produced

   that leads to the logic for descending, if the first one fails.

5. Two lists are prepared:

   • checkTrueList: quadruplets that will be filled with input to the loop body. • checkFalseList:

   quadruplets that should lead to the loop exit.

6. If step >= 0:

   • A quadruple <=, ID, expr2, – is created, which checks whether the price has exceeded the upper limit.

   • If true, we continue in the body. If not, we jump out of the loop.

7. If step < 0:

   • A quadruple >=, ID, expr2, – is created, for decreasing order. reception.

   • Similar to before, we are led to the body or the exit.

8. All quads pointing to the body are filled in via backpatch(checkTrueList, nextquad()) and we visit the sequenceNode to produce the corresponding intermediate code of the body.

9. After executing the body, the ID value must be increased or decreased:

   • A temporary variable temp is created and the quadratic +, ID, step, temp.

   • An assignment follows: :=, temp, –, ID, which updates the loop counter.

10. Finally, the logic for direction is repeated:

   • If step >= 0, we return to the <= test. • Otherwise, to the

   >= test.

11. Finally, the quadruplets leading to the exit from the loop (checkFalseList) are completed to transfer the flow to the command that follows after the for.

This implementation achieves complete generality in the direction of the iteration. The check whether the step is positive or negative is done dynamically, and the comparisons are adjusted accordingly, ensuring that the loop termination condition is applied correctly. The generated intermediate code is independent of whether the step is a constant or an expression evaluated at runtime.

Edit function (visitFunction)

This method is responsible for the processing and intermediate code of a Greek++ language. Its function implements the declaration, analysis, code generation and return value stages step by step, as described below:

1. **Reading function name:** The function's identifier name is stored and a Function object with type Integer is created, which is declared in the ScopeManager. The function's depth is currentDepth + 1, as a new scope is opened.

2. **Set current owner:** The variable currentBlockOwner is set to point to the function. This allows for proper mapping of parameters and the return variable to the function's activation record.

3. **Opening a new scope:** The ScopeManager opens a new scope so that the function's local variables, parameters, and declarations are stored in its own visibility environment.
4. **Parameter processing:** We visit the formal parameters node (formalParametersListNode) and they are registered in the function's symbol table.
5. **Return variable declaration:** The ScopeManager adds a special variable with the same name as the function, which acts as a return variable. This will be the final recipient of the return value.

6. **Visiting individual sub-nodes of the body:**

   • functionInput: input parameter declarations. • functionOutput: output or reference declarations.

• declarations: local variables. • subprograms: internal subprograms. • sequence: main commands of the function.

All of the above recursively visit and fully process the body of the function.

7. **Starting and marking the block:**
   - A begin_block quad is generated with the function name. • The position of the first instruction of the block is recorded in ActivationRecord of the function.
8. **Sequence visit:** The main body of the function's instructions is visited, and intermediate code is generated.
9. **Return of price:**
   - A retv quad is produced with the value of the e-return variable (which has been declared under the same name as the function).
   - Completed with an end_block quad to indicate the end of the frame.
10. **Recording temporary variables:** All temporary variables created within the function are added to the ActivationRecord for allocation and final code purposes.
11. **Generate final code:** generateAsmForCurrentScope() is called to translate the intermediate code of the specific scope into final RISC-V assembly.
12. **Close scope:** The ScopeManager terminates the function's scope and returns to the external environment.

Note: The remaining syntactic rules of the Greek++ language are processed in a similar way, following the same systematic logic: each node recursively visits its children, collects the necessary semantic information (such as place, trueList, falseList), and generates intermediate code through quadrats. The approach is normative and fully aligned with the method proposed in the course's educational material, ensuring that the intermediate code generation process is rule-independent, clean, and extensible.

Chapter 9

Final Code Generation (Final Code)

The last stage of compilation involves the conversion of the intermediate representation of the program into final machine code, specifically designed for the RISC-V architecture. The production of the final code is based on the quads created in the previous stages and strictly follows the stack model described in the course.

The generated code is fully executable in the Ripes simulator and compatible with the calling conventions and frame layout of the architecture. The conversion process is implemented through the final_code package, which includes the necessary classes for representing operands, arithmetic constants, variable accesses, and the RISC-V instruction generator.

The main classes involved in this stage are:

- Operand: Abstract superclass that describes anything that can be used as an operand in a final instruction (variable, constant, etc.).
- IntConst: Implements integer constants (immediate values) used as command arguments.
- VariableOperand: Describes access to a variable (local, temporary or parameter) and how to calculate the address.
- RiscVAssemblyGenerator: It is responsible for converting each quad into one or more assembly instructions, according to

the subroutine activation record and stack management.

9.1 Abstract Representation of Operators: Operand, IntConst, VariableOperand

The logic behind the existence of Operands is not about language complexity or the need to support multiple data types, but serves a very specific purpose: to unify the behavior of operators when generating final code, regardless of whether the operator is a variable or a constant.

In the assembly phase, each instruction argument must be converted to an appropriate symbolic or numeric form (e.g. offset from sp, literal value, temporary register, etc.). However, the origin of each operand can be very different:

- **If it is a variable,** it has already been stored in the activation record with a predefined offset. This offset is available through the Entity type entity returned by the ScopeManager during analysis. Therefore, it is sufficient to resolve correctly and the operand knows how to generate the appropriate code (lw, sw) for access via VariableOperand.

- **If it is a constant,** then there is no corresponding stored location on the stack. The value is “injected” at runtime and does not need to be looked up in the scope. Such values are implemented as IntConst and are assigned directly to the code with instructions such as li, addi, etc.

Without the abstract Operand class, RiscVAssemblyGenerator would have to constantly do instanceof or if-else to distinguish between operand types. With the current system, each operand type implements its own toString() or code generation methods, and the final code generator can work seamlessly and without dependencies.

The hierarchy is as follows:

- **Operand:** Abstract superclass that defines the interface of all operands.

- **IntConst:** Implements integer constants and returns the value directly. their numeric value as a string.
- **VariableOperand:** Represents variables (local, parameters, temporary) and uses offset to access sp.

This structure ensures that any operand can be used in a uniform manner across code generation methods, dramatically reducing complexity and increasing backend extensibility.

Chapter 10

Final Code Generation: RiscVAssemblyGenerator

The RiscVAssemblyGenerator class is responsible for translating intermediate code quads into executable RISC-V assembly code, compatible with the Ripes simulator. Its main feature is that it provides self-contained assembly code generation for each instruction, while following the rules of the RISC-V ABI and stack management depending on the scope depth.

10.1 Purpose and Architecture

RiscVAssemblyGenerator does not function as a simple string converter. Its role is to maintain the correspondence between the symbolic intermediate level (quads) and the low level (assembly), taking into account:

- The depth of scopes (ScopeManager),
- The type of each operator (using Operand, VariableOperand, IntConst),
- The need to manage registers, access from different levels, and temporary values.

The assembly code is generated incrementally, each time the generator is called at the close of each scope, executing each time:

1. Processing each quad via generateAsmForCurrentScope(),

2. Choosing the appropriate method depending on the operator (:=, +, call, retv etc.),

3. Create asm instructions in String format, stored in the instructions list,

4. Final writing to a file via writeToFile().

10.2 Command Generators

Each command category is supported by a special function:

10.2.1 Assignments and Arithmetic Operations

Arithmetic operations (+, -, *, /) are translated through:

- generateAsmForArithmeticOperation(),
- emitArithmeticOperation().

The generator loads the operands, uses the registers t1 and t2, performs the arithmetic operation, and stores the result.

10.2.2 Assignments :=

generateAsmForAssigment() implements value copying. If the operator is a constant, a direct transfer via li is performed. If it is a variable, its position on the stack is searched based on depth.

10.2.3 Control Flow Structures (Jumps, Conditions)

Jumps are implemented through:

- emitJump() for unconditional jumps,
- emitConditionalJump() for comparisons.

Commands like beq, blt, bge are generated depending on the operator.

10.2.4 Printing and Login

The out and in commands are converted with:

- generateAsmForPrint(),
- generateAsmForInput().

Standard ecalls with a7 are used for I/O.

10.3 Subroutines and Stack Management

The most demanding function of the generator concerns the support of subroutines and the activation record.

10.3.1 Starting and Ending a Subroutine

The begin_block and end_block implement:

- Stack space allocation,
- Saving / restoring ra (return address),
- Management of local variables.

10.3.2 Function Call

generateAsmForSubroutineBlock() handles:

1. Stack allocation for the called routine,
2. Dynamic link storage,
3. Copying parameters depending on the passing method (by value / by reference),
4. Jump to the beginning of the subroutine,
5. Freeing the stack at the end.

10.3.3 Parameters

Parameters are supported by:

- emitParameterByValue(),
- emitParameterByReference().

The distinction based on Parameter.Mode ensures that even recursive or complex calls work correctly with the correct offset.

10.4 Using Operands

The class uses the Operand structure to maintain uniformity in assembly production. Depending on the type:

- IntConst: Produces li,
- VariableOperand: Uses lw/sw with offsets,

while looking up variables in parent scope stacks is handled with gnlvcode() and TEMP_0 as an intermediate register.