Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

European Master in Software Engineering

Master Thesis

Development of a Generic Compiler: Intermediate Code Design, Intermediate Code and Assembler Analyzers, and Assembler Translator

Author: Javier Ramírez Manzanares

June, 2025

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis

European Master in Software Engineering

Title: Development of a Generic Compiler:
Intermediate Code Design, Intermediate Code and Assembler Analyzers, and Assembler Translator

June / 2025

Author: Javier Ramírez Manzanares

Supervisor:

Name and Surnames:
José Luis Fuertes Castro

Academic title:
PhD in Computer Science

University of the presented title:
Universidad Politécnica de Madrid

Department /School:
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos

University:
Universidad Politécnica de Madrid

Abstract

This Master Thesis is part of a broader project whose objective is the development of a comprehensive system that enables students to automatically self-assess the practical assignment of the Language Translators course in the Bachelor's Degree in Computer Engineering at the Universidad Politécnica de Madrid.

This system is integrated into DRACO, a gamified web platform aimed at enhancing learning in the courses Language Processors and Language Translators.

The practical assignment in this course consists of developing a compiler for a defined language. Currently, there is no environment available where students can test the systems they develop, so the responsibility for validation lies entirely with them, making the task more difficult.

To address this situation, the proposed solution is the development of an automated system capable of evaluating the outputs generated by the compilers developed by the students. Furthermore, the system is reusable, as it is independent of the specific language specifications defined each academic year.

From the intermediate code or assembly code produced by the student's compiler, the system performs an analysis to detect errors, translates the code into object code, and executes it to compare the output with the expected result. This enables an automated and objective self-assessment process that shows students the errors found.

The complete generic compiler project consists of several phases: definition of the generic source language, source code compiler with translation to an intermediate representation in the form of quadruples, quadruple-to-assembly compiler, assembly code analyzer, code executor module, and integration of the system into the web platform.

This Master's Thesis focuses on the design and development of three key components: the design of the intermediate language (quadruples), the quadruple-to-assembly compiler, and the assembly code analyzer. All these components have been implemented in PHP, the base language of the DRACO platform, enabling efficient integration with the existing environment. Additionally, the development has followed a rigorous design process, applying various Software Engineering techniques to ensure the system’s maintainability, clarity, and consistency.

The results obtained through this work represent a significant step forward in the future functionality of the DRACO system. Once fully integrated into DRACO, the generic compiler will become a key support tool for students during the development of the practical assignment. Altogether, it is expected to increase student motivation and engagement, as well as the overall quality of learning in the course.

Resumen

El Trabajo de Fin de Máster que se presenta forma parte de un proyecto más amplio, el objetivo de este es el desarrollo de un sistema integral que permite a los alumnos autoevaluar de forma automática el trabajo práctico de la asignatura Traductores de Lenguajes, del Grado en Ingeniería Informática de la Universidad Politécnica de Madrid.

Este sistema se integra en DRACO, una plataforma web gamificada orientada a mejorar el aprendizaje en las asignaturas de Procesadores de Lenguajes y Traductores de Lenguajes.

El trabajo práctico de la asignatura consiste en desarrollar un compilador para un lenguaje definido. Actualmente no existe ningún entorno donde los alumnos puedan realizar pruebas sobre el sistema que desarrollan, por lo que la labor de validación recae exclusivamente sobre ellos, dificultando el trabajo.

Ante esta situación, se plantea como solución el desarrollo de un sistema automatizado capaz de evaluar los resultados generados por los compiladores desarrollados por los estudiantes. Además, este sistema es reutilizable puesto que es independiente de las distintas especificaciones del lenguaje que se definan cada año.

A partir del código intermedio o del ensamblador, que produce el compilador del alumno, el sistema realiza un análisis en busca de errores, lo traduce a código objeto y ejecuta para comparar la salida con el resultado esperado. Gracias a esto se logra una evaluación automatizada y objetiva que muestra a los estudiantes los errores encontrados.

El proyecto completo del compilador genérico incluye varias fases: definición del lenguaje fuente genérico, compilador fuente con traducción a una representación intermedia en forma de cuartetos, compilador de cuartetos y su traducción a lenguaje ensamblador, analizador de código ensamblador, módulo de ejecución del código y la integración de este sistema en la plataforma web.

Este Trabajo de Fin de Máster se centra en el diseño y desarrollo de tres componentes clave: el diseño del lenguaje intermedio (cuartetos), el compilador de cuartetos a ensamblador y el analizador de código ensamblador. Todos estos componentes se han programado en PHP, el lenguaje base del sistema DRACO, lo que permite una integración eficiente con el entorno ya existente. Además, el desarrollo se ha llevado a cabo siguiendo un diseño riguroso y aplicando distintas técnicas propias de la Ingeniería del Software con el objetivo de garantizar la mantenibilidad, claridad y coherencia del sistema.

Los resultados obtenidos con este trabajo suponen un avance significativo en la futura funcionalidad del sistema DRACO. Con su integración final en DRACO el compilador genérico supondrá una herramienta de apoyo clave para los alumnos durante el desarrollo del trabajo práctico. Todo ello permitirá aumentar la motivación y la implicación de los alumnos, así como la calidad del aprendizaje en la asignatura.

Table of Contents

1 Introduction

1.1 Context and Motivation

The courses *Procesadores de Lenguajes* and *Traductores de Lenguajes*, taught within the Bachelor's Degree in Computer Engineering, provide comprehensive training in the design and implementation of compilers. The former focuses on the analysis stages of compilation — lexical, syntactic, and semantic — while the latter is dedicated to the synthesis stages, including intermediate code generation, its optimization, and subsequent object code generation.

Both courses include mandatory practical assignments in which students must design and implement a compiler for the language defined for that academic year. Specifically, the practical component of the *Traductores de Lenguajes* course builds upon the work completed in the *Procesadores de Lenguajes* course, addressing the final stages of the compilation process: intermediate and assembly code generation.

Currently, the *Traductores de Lenguajes* course lacks any integrated tool within the DRACO platform that allows students to test and validate the results generated by the compiler they are developing. Therefore, the entire responsibility for verifying the correctness of their implementation lies with the students themselves. This process must be carried out manually, making it difficult to identify errors efficiently and increasing the complexity of the assignment. The lack of immediate feedback also limits students' ability to iterate and improve their solution throughout the development process.

However, in the *Procesadores de Lenguajes* course, this situation has been mitigated using an automated self-assessment tool integrated into the DRACO web system. This platform allows students to test their compilers using automatically generated test files and to analyze the results obtained.

It is worth highlighting that the work presented here is framed within the DRACO system. This web platform has been well-established for several years in the educational context of the aforementioned courses. It serves as a support tool for learning, using a gamification approach that encourages students to complete exercises and earn rewards based on their performance.

In addition to fostering active student participation, DRACO automates part of the grading process for the *Procesadores de Lenguajes* course and provides immediate feedback to the students, which contributes to more effective learning, as previously mentioned.

1.2 Scope of the Work

Within the DRACO system, a project was proposed during the 2024/2025 academic year to develop a generic compiler aimed at helping students automatically assess their own practical assignments for the *Traductores de Lenguajes* course.

Given the project's complexity and scale, it has been divided into several stages:

1. Design of a generic source language expressive enough to adapt to the different languages defined each academic year.
2. Design and implementation of a generic source language compiler capable of analyzing code and generating intermediate code.

3. Design of the intermediate language, using a representation based on quadruples.
4. Design and implementation of a quadruples compiler that generates assembly code.
5. Design of the final object code, based on an adapted version of the ENS 2001 assembly language [1].
6. Design and implementation of an assembly code analyzer.
7. Design and implementation of the module that executes the assembly language.

This Master Thesis is part of this initiative, addressing the development of several key phases of the project: from the design of the intermediate language to the development of the assembly analyzer.

1.3 Objective of the Master Thesis

The main objective is to develop and complete phases from 3 to 6 of the project:

- Design an expressive and flexible quadruples language capable of adapting to the source language defined each academic year.
- Implement a quadruples compiler that generates valid assembly code.
- Adapt and use a modified version of the ENS 2001 assembly language.
- Implement an assembly analyzer capable of performing lexical, syntactic, and semantic analysis.

Another fundamental objective is to ensure that all these components are easily integrable with the rest of the modules of the generic compiler within the DRACO web system.

1.4 Methodology

The development of the Master Thesis has been carried out iteratively and collaboratively with the administrator of the DRACO platform, who also serves as the academic supervisor of the project. Throughout the process, biweekly meetings were held to review progress, discuss improvement proposals, resolve doubts, and validate design decisions.

Each component was designed, implemented, and validated incrementally. Furthermore, to ensure system robustness, an extensive testing phase was conducted using multiple test cases simulating both valid and erroneous input files.

1.5 Results

The developed system is fully operational and, as previously mentioned, has been thoroughly tested with satisfactory results. Both the quadruples compiler and the assembly analyzer have demonstrated correct behavior.

Although the system has not yet been deployed in DRACO's production environment, its integration is planned for later phases of the project, once the remaining modules of the generic compiler are completed.

1.6 Document Structure

This document is organized into the following chapters, which systematically present the work carried out:

- **State of the Art**: Provides a review of key concepts related to compiler design, covering its various phases and the technologies and languages relevant to this project.
- **Problem Statement**: Presents the system requirements analysis, including both functional and non-functional requirements. It clearly defines the scope of the project and the technical objectives to be achieved.
- **Development**: Describes the design, implementation, and testing of the proposed solutions. It is divided into several sections that detail the design of the developed components.
- **Results and Conclusions**: Summarizes the outcomes of the development, including use cases, achievements, and reflections on the work done.
- **Future Lines of Work**: Outlines potential improvements and future extensions for the system.
- **Bibliography**: Lists the references used throughout the document.
- **Annexes**: Contains supplementary material to aid in understanding the work performed.

2 State of the art

This chapter provides an overview of the key technologies and foundational knowledge relevant to the development of this work.

2.1 Compiler

A compiler is a software program that translates source code written in a given programming language (typically high-level) into machine language, which is then understood by the computer. The use of compilers has facilitated the evolution of high-level programming languages. These offer a higher level of readability and maintainability for developers compared to the direct usage of machine language. This technological advancement has contributed to the development of more efficient and faster languages. [2]

The advent of the compiler marked a crucial turning point in the history of computing, as it significantly streamlined the programming process and enabled individuals lacking expertise in machine code to utilize computers. This pivotal advancement was spearheaded by Grace Hopper, who in 1952 developed the first compiler, A-0, with the capability of converting high-level language instructions into machine code. The success of A-0 led Hopper to develop FLOW-MATIC, a compiler with English commands that constituted the foundation for COBOL, one of the first languages accessible to a wider community of users.

In 1957, John Backus and the IBM team introduced the FORTRAN compiler, which became standard in scientific programming and reflected the considerable complexity involved in developing such tools. Over time, self-hosted compilers, such as LISP in the 1960s, emerged, capable of compiling themselves, thereby affording greater flexibility to software development. These advances consolidated compilers as fundamental pieces of modern software development, facilitating the creation of more complex and accessible programs. [3]

Even though compilers and interpreters may serve similar purposes, they are, in fact, distinct entities. A compiler is a program that analyzes and translates the complete source code of a program, generating machine code that can be executed. This process encompasses a series of phases that facilitate the generation of efficient programs.

In contrast, an interpreter translates and executes the code of a program line by line. This facilitates the debugging and testing of programs for developers, although it results in slower execution times. The selection of the appropriate tool depends on the specific requirements of the project and the language being used. [2]

The compiler executes a series of steps to transform the source code into an executable program. The initial stage of the process is a lexical analysis, whereby the code is divided into tokens or basic elements of the language. Subsequently, through syntactic and semantic analysis, the structure and meaning of the code are verified to ensure correctness and consistency. Next, it generates an intermediate code, an abstract representation of the program. Ultimately, the compiler generates the object code and links it with external libraries, if necessary, to produce an executable file that is ready to be loaded and executed on the computer. It is also worth noting that many compilers include an optimization phase, which may operate on the intermediate code, the object code, or both, depending on the implementation.

In addition to the main phases, most compilers include auxiliary components such as the symbol table, which stores information about identifiers, and the error handling, which detects and manages errors throughout the compilation process.

Figure 1 provides an overview of the structure of a compiler, delineating the sequence of phases.

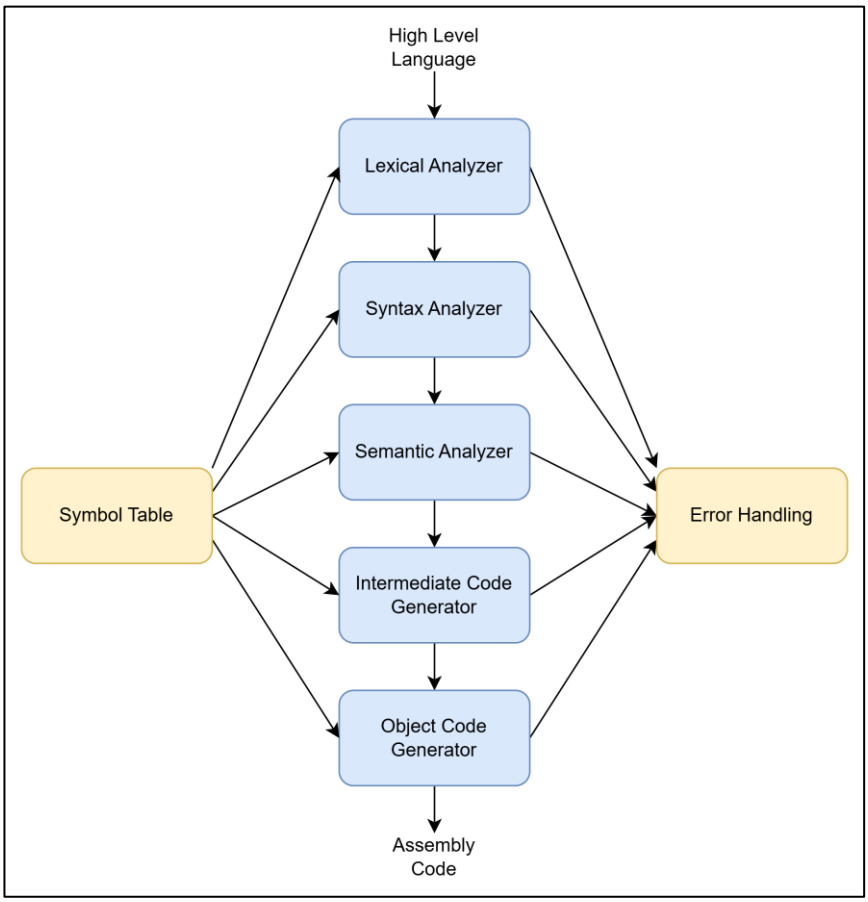

Figure 1 Phases of a compiler

The compiler can be divided into two phases: analysis and synthesis. The analysis phase includes the lexical, syntactic, and semantic analyzer and is considered the language processor. The synthesis phase is carried out by the intermediate code generator and the object code generator, forming the language translator.

The various phases of a compiler are delineated below, apart from the code optimizer, which has not been a component of the development process for this project.

2.1.1 Lexical Analysis

Lexical analysis constitutes the initial phase of the compiler process. The objective of this phase is to read and analyze the user's source code in order to decompose it into smaller, meaningful elements. The resulting components of the source code, known as "tokens" will subsequently be utilized by the compiler.

Furthermore, as the scanner is the component of the compiler responsible for processing the source file, it is tasked with associating the error messages generated by the compiler with the corresponding line of source code. [4]

A token represents the fundamental unit of meaning in the source code. The lexeme is defined as the specific sequence of characters in the source code that forms a token. The rules that define the sequences of characters that correspond to a specific token are referred to as the token pattern. Table 1 below provides examples of generic tokens.

Token	Pattern	Lexeme
Integer	One or more digits	245
String	List of characters between " and "	"String example"
Identifier	A letter followed by letters and digits	counter
open parentheses	An open parenthesis	(
Comma	A comma	,

Table 1 Examples of tokens with lexeme and pattern

In addition to this, each token consists of two main parts:

- **Token type**: indicates the category of the token, which is defined according to the language in question.
- **Attribute**: provides additional information about the token, and its content varies according to the type of token.

To illustrate, consider the following example:

```
<constant integer, 4>
```

The example illustrates the tokenization of a number from the source code. The token type is an integer constant, and the attribute serves to store information pertaining to the value of the number.

The design of a scanner for a language compiler requires the completion of several tasks:

1. The initial stage of the process is to **identify and define the tokens** of the source language. In order to accomplish this, it is essential to have a comprehensive understanding of the language to be compiled, as well as to undertake a detailed analysis of its constituent meaningful elements.
2. It is essential to **define the regular grammar** that generates the valid lexemes of the defined tokens. A regular grammar is constituted of rules that indicate the permitted combinations of symbols from an alphabet that can be used to form valid lexemes.

   Figure 2 is an example of a regular grammar for language identifiers that begin with a letter followed by other letters or digits:

```
S → letter A
A → letter A | digit A | λ
```

Figure 2 Regular grammar example

3. The subsequent phase of the process is the **design of the deterministic finite automaton (DFA)**. It is a model that is employed for the representation and processing of sequences of symbols within a formal language.

   A DFA is constituted by a finite set of states, which can be defined as the conditions under which the automaton finds itself. During the processing

period, the automaton analyzes the input symbols, which may include characters. For each state and symbol read, a transition function is defined that determines the rule for switching from one state to another.

Upon reaching a final state, the automaton accepts the processed string. Otherwise, the string is rejected, as it does not comply with the defined pattern, resulting in the generation of an error.

The main characteristic of an AFD is that the transition functions are deterministic, indicating that for each input state and symbol, there is a single defined target state.

Figure 3 illustrates the AFD of the regular grammar of the previous example.

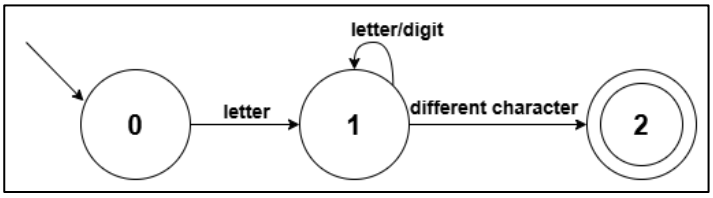

Figure 3 Deterministic finite automaton example

4. The following step is to incorporate the necessary **semantic actions** into the deterministic finite automaton. These are the operations that the automaton must perform when it is in a state and processes an input symbol, in addition to the corresponding transition.

   The objective of these actions is to associate meaning with the lexemes read. To exemplify, one may store the value of a number, concatenate characters to form a word, generate linguistic tokens upon reaching a final state, or store an identifier in the symbol table.

   Figure 4 illustrates the semantic actions associated with the aforementioned example.

> **Transition 0–1:** Save letter, read next symbol.
> **Transition 1–1:** Concatenate character to the string saved, read next symbol.
> **Transition 1–2:** Generate identifier token.

Figure 4 Semantic actions example

5. The final step is to **detail the error cases**. It is essential to ensure that all expected errors are correctly reported in the cases identified as incorrect. Such unspecified transitions between states.

As previously stated, the scanner represents the initial component of a compiler. Figure 5 illustrates its interaction with other modules. As indicated in the figure, this is the responsible for processing the source file. Furthermore, it interacts with the parser, sending it the respective tokens when required. In addition, it relates to the symbol table to store information and with the error handling to generate lexical errors when required. [5]

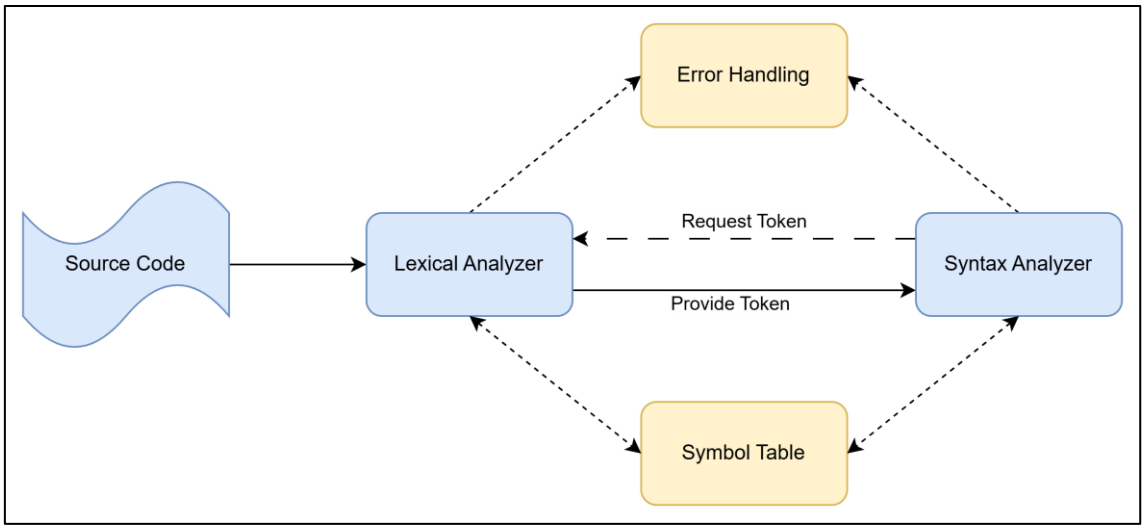

Figure 5 Scheme of some of the parts of the compiler.

2.1.2 Syntactic Analysis

Syntactic analysis represents a pivotal stage in the compilation process. Its function is to verify that the tokens adhere to the grammatical structure of the specified language. This process takes place next to the lexical analysis.

As illustrated in Figure 5, the parser can be considered the second phase of the compiler. It requests tokens from the scanner for the purpose of verifying compliance with the established grammatical rules. Furthermore, it interacts with the error handling component to launch syntactic errors. Additionally, it makes use of the symbol table to query information on the program identifiers.

In this syntactic verification phase, type II grammars, also designated as context-free grammars, are employed. In the process of following these grammars, the analyzer constructs a syntactic tree whose root is the axiom and whose leaves are the tokens. This is represented by the parse, which are the numbers of the rules of the grammar that are employed in the construction of the tree. The parse is then sent to the semantic analyzer, which proceeds with the compilation process. [6]

Regarding the manner of parsing, there are two distinct types. Those that employ backtracking to identify the solution through a process of trial and error, exploring a range of potential approaches. The second method, which is used without backtracking, is known as predictive. This method always progresses along the correct path of the parse tree. This final category is the most efficient and is the one that is used most frequently.

There are two methodologies for the construction of a parser:

- **Bottom-up approach**: The tree is constructed from the lowest-level nodes to the root. In order to achieve this, right-hand derivations are employed, with the terminals being successively reduced.
- **Top-down approach:** It is constructed from the root to the lowest-level nodes. This method employs left derivations, which are used to successively expand the non-terminals.

The following section provides a comprehensive explanation of each of the aforementioned methods, with a particular focus on the top-down approach which was employed in this project.

2.1.2.1 Bottom-up Parser

In bottom-up parsers, the derivation tree is constructed from the leaves to the root, employing a strategy of reduction and shifting. An LR(1) parser employs a stack and a parse table that is divided into two sections: Action and Goto.

The Action table indicates the appropriate shift, reduce, accept, or error-mark operations, based on the current token and the state at the top of the stack. In contrast, the Goto table is responsible for managing transitions between states for non-terminal elements. This type of parsing is particularly effective for programming languages, as LR grammars allow for the representation of complex structures.

In order to construct an LR parser, the Action and Goto tables are generated using, for instance, the SLR (Simple LR) method. The process begins with an augmented grammar and the construction of the canonical collection of LR(0) items. This collection serves to identify the feasible prefixes, thereby enabling the construction of an automaton that is capable of recognizing these prefixes. This process allows for the systematic definition of each action within the Action and Goto tables, facilitating the resolution of conflicts that may arise when the grammar is LR. This approach guarantees accurate and efficient analysis, which is fundamental for the correct interpretation and compilation of code in programming languages.

2.1.2.2 Top-down Parser

Top-down parsers build the syntactic tree from the root to the leaves through a process known as left derivation.

The grammar of top-down parsers must adhere to a few constraints. It is not permissible for there to be left recursion. Furthermore, no two rules from the same non-terminal may begin (or derive) with the same symbol.

To comply with the constraints, it is essential to eliminate left recursion and to left factorize the rules of the grammar.

Consequently, for predictive top-down analyzers, it is imperative to utilize LL(1) grammars. These grammars must satisfy the following conditions:

- If rules A -> α, A -> β exist, then First(α) ∩ First(β) = ∅.
- If λ ∈ First(α), then it must be checked that Follow(α) ∩ First(β) = ∅.

The concepts First and Follow are defined below:

First: The set of terminal symbols that can appear as the initial terminal symbol in strings derived from a given symbol X. This set is represented by the notation First (X), where X is a terminal symbol belonging to the set T ∪ N, with T denoting the set of terminal symbols and N the set of non-terminal symbols.

Follow: The set Follow(A) is defined for nonterminal A (A ∈ N) as the set of terminals t (t ∈ T) that can appear immediately to the right of A in some sentential form. Formally, this is the set of terminals t such that there is a derivation of the form S ⇒* α A t β, where α, β ∈ (N ∪ T)*. In the event that a given symbol A can be positioned as the rightmost symbol within a sentential form, then the element $ is included within the set Follow (A).

Figure 6 illustrates an example of a grammar that is not of the LL(1) type due to the presence of left recursion.

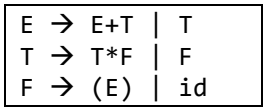

Figure 6 Example grammar that is not LL(1)

Figure 7 presents the equivalent grammar that is valid for a predictive descendant syntactic parser.

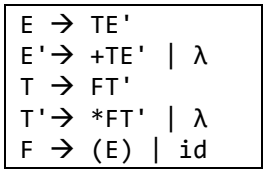

Figure 7 Equivalent grammar LL(1)

In the category of predictive top-down analyzers (without backtracking), two distinct methods can be identified:

- **Recursive**: These analyzers implement recursive parsing procedures, utilizing both the final received token and the present state.
- **With table**: Such analyzers employ tables to query and determine the subsequent action to be performed by the parser.

Although a top-down parser with tables can be considered a faster option for implementation, in this project, we have elected to utilize recursive top-down analyzers. This selection is based on the premise that a recursive parser is more intuitive and straightforward. Furthermore, this approach enables greater control over the semantic actions of the compiler's semantic analyzer.

The following section provides a detailed overview of the recursive top-down parser.

2.1.2.2.1 Recursive parser

When designing a recursive top-down parser, the initial step is to obtain an LL(1) grammar and verify that it meets all previously mentioned constraints.

The design of this type of parser is based on the implementation of a procedure for each non-terminal symbol of the grammar. This procedure implements a logic that determines the applicable rule when an input token is analyzed.

The rules of the procedures are detailed below [7]:

- If the `token ∈ First(α)` the rule `A → α` is used.
- If the `token ∉ First(α)` nor `First(β)` but there is rule `A → λ`, that rule is used, otherwise a syntactic error is thrown.
- If there is a terminal symbol in α that matches the token, it is consumed, and the next token is requested to the scanner.
- If there is a non-terminal symbol in α, the procedure for that symbol is called.

2.1.3 Semantic Analysis

The semantic analyzer represents the third phase of the compiler. Its function is to verify the semantics of the source program. To perform this function, it makes use of a syntactic tree and the symbol table, checking that the rules of the source language are complied with. Some of the verification processes conducted by the semantic analyzer include ensuring that data types are used

correctly, checking the correct use of labels, and ensuring that control structures are used in accordance with the source language specification. [8]

Semantic analysis represents the final stage of the compiler analysis phase, immediately preceding the start of the synthesis phase. It is placed between the parser and the intermediate code generator. Figure 8 illustrates the relationship between the semantic parser and other components of the compiler.

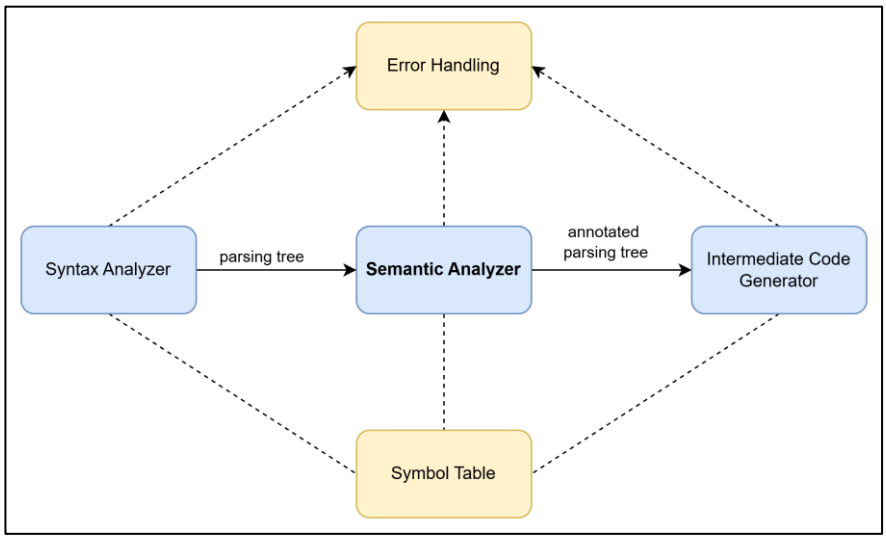

Figure 8 Semantic analyzer interaction scheme

As illustrated, the semantic analyzer is situated after the syntactic analyzer. Once the semantic verification has been carried out and the necessary information has been noted, the intermediate code is generated. In addition, it is connected to the symbol table, which can be used to query identifier information when necessary. Furthermore, it is also linked to error handling, which can be used to launch semantic errors that have been identified.

Attribute grammars are employed in semantic analysis to associate semantic information with the syntactic structures of a language. These grammars extend the context-independent grammars of the parser by incorporating attributes and semantic actions. As a result, attribute grammars facilitate the capture of the meaning of language constructs by enabling processes such as type checking and intermediate code generation.

- **Attributes**: represent properties of grammatical symbols and are divided into two main types: synthesized and inherited.

  Synthesized attributes: This type of attribute has its value determined from the values of the attributes of the child attributes of a node in the parse tree. This represents an upward flow of information. For example, in the rule: `E → ( E₁ )` the attribute `E.type` can be derived from the attribute `E₁.type`, thus synthesizing information from a child node to its parent.

  Inherited attributes: An inherited attribute, on the other hand, takes its value from the attributes of a node's parent or sibling. This type of attribute represents a horizontal or downward flow of information. For example, in the rule `D → T L ;` the `L.type` attribute is inherited from the `T.type`, which transfers information from the context to the current node.

- **Semantic actions:** represent the operations that compute the values of attributes within a given grammar. These actions are associated with specific syntactic rules and utilize the attributes of the grammar symbols

corresponding to the rules. For example, in the rule `L → L1` id the semantic action has the potential to set `L1.type = L.type` ensuring that the relevant information is transmitted between the nodes of the parse tree.

In the field of attribute grammars, syntax-driven translations (SDT) represent a formalism that allows for the specification of how each construct of a language is translated in accordance with its syntactic structure. There are two principal notations through which this formalism may be implemented: the Syntax-Driven Definition (SDD) and the Translation Schema (TD).

- **Syntax-Driven Definition (SDD):** Represents a generalization of context-independent grammars, in which each grammatical symbol is associated with a set of attributes, and each syntactic rule is accompanied by a set of semantic actions. These actions compute the values of the attributes, thereby establishing the semantic meaning of the constructions described by the grammar.
- **Translation Schema (TS):** In contrast, the TS represents a variant of the DDS that incorporates a crucial additional element: the explicit order of evaluation of semantic rules. In distinction to the DDS, the TS associates semantic actions with specific positions within the productions of the grammar. This allows for the indication of the execution of these actions in relation to the derivation of the construction.

  This capability renders TS especially advantageous in practical applications, as it permits more precise control over evaluation and facilitates compatibility with top-down and bottom-up parsing methodologies. This is the reason why it has been decided to employ translation schemas in the design of the semantic analyzers of the project.

  The following rules should be adhered to in the translation scheme:

  - If only synthesized attributes are present, the semantic actions should be situated at the end of the rule.
    $E \rightarrow (E_1) \{E.type = E_1.type\}$
  - If inherited attributes are present, the following rules must be observed:
    - In the case of an inherited attribute pertaining to a grammatical symbol situated to the right, it is necessary to calculate this attribute in a preceding semantic action.
      $S \rightarrow \{A_1.h = 1\}\ A_1\ \{A_2.h = 2\}\ A_2$
    - Semantic actions should always refer to synthesized attributes of symbols situated to the left of it.
      `D → T {L.type = T.type} L ;`
    - If a synthesized attribute is required for a non-terminal symbol situated to the left of the rule, it must be calculated after the completion of all the attributes to which it refers.
      `L → A B {L.s = A.a + B.b}`

Semantic verifications represent a fundamental aspect of semantic analysis, with the objective of guaranteeing that a source program adheres to the established semantic rules of the language in question.

Such verifications encompass elements such as control flow, name declaration uniqueness, type matching, and the validity of references to declared names. Such verifications not only guarantee semantic correctness but also prevent

errors during program execution, thereby enhancing the robustness and reliability of the program.

A common example of type checking is the addition operation. Let us consider a rule for an expression: `E → E1 + E2`. In this case, the semantic action serves to verify that both operands, E_1 and E_2, can be added.

In the event that both operands are of the integer type, the result will also be of the integer type. If both operands are of the real type, the result will also be of the real type. However, if the types are not compatible —for example, if a combination is attempted between an integer and a logical type— a semantic error is generated. [9]

2.1.4 Symbol Table

The symbol table is defined as a data structure. The purpose is to store all the necessary information related to the source program identifiers. It is conceptualized as a table comprising a single entry or record for each identifier.

As illustrated in Figure 1, the symbol table is accessible by all compiler modules, serving both as a store for information and as a source for the retrieval of that information. [10]

The data structure contains a variety of information related to identifiers. The entries are not homogeneous; that is, depending on the identifier, one specific information item (attribute) or another may be stored. However, the lexeme and the type of identifier are always stored for each entry in the table. Furthermore, the symbol table can accommodate additional information such as the offset, the dimension of an array, the number and type of parameters, the label, and the return type in the case of a function.

A compiler symbol table must be capable of performing the following operations: table creation, entry insertion, entry query, table deletion, attribute insertion and attribute request.

It is important to note that there is not a single symbol table. The structure in question is specific to each scope. Consequently, a symbol table is created for each function that is declared in a program. This table is unique to that function and is deleted when the function leaves the scope of the program. Furthermore, a global table is always present.

The implementation of a compiler's symbol table can be approached in several ways, for instance [11]:

- **Linear**: In this type of symbol table, the insertion is performed in a consecutive manner without any order. The implementation of this type of symbol table is relatively straightforward; however, the search for an entry is subject to a linear complexity.
- **Linear ordered:** The symbol table is constructed with records inserted following a specific order. In most cases, an alphabetical order is employed. This introduces additional complexity during the insertion process but reduces it during the search, allowing the implementation of more efficient binary search strategies.
- **Hash**: This type of symbol table is sorted and organized according to a hash function. This enhances the efficiency of the symbol table, but it also elevates the intricacy of its design and implementation.

The following example presents an illustrative program, Code 1, along with the corresponding symbol tables, Table 2, Table 3. It is presumed that the language under consideration requires the prior declaration of variables. It is also assumed that integer types are represented by two bytes.

```
int a, b;
boolean func1(int x){
    int b;
    b=10;
    return x>b;
}
void main(){
    a=1;
    func1(a);
}
```

Code 1 Example source program

Lexeme	Type	Offset	Number of parameters	Type of parameters	Return type	Label
a	Integer	0				
b	Integer	2				
func1	Function		1	Integer	Boolean	Et1

Table 2 Global symbol table

Lexeme	Type	Offset	Number of parameters	Type of parameters	Return type	Label
x	Integer	0				
b	Integer	2				

Table 3 Function1 symbol table

2.1.5 Intermediate Code Generation

The Intermediate Code Generator (ICG) is the initial component in the synthesis phase of the compiler. The principal objective of the ICG is to facilitate the transformation of a correctly analyzed source program into a machine-independent intermediate representation. This representation facilitates not only the portability of the compiler but also the implementation of optimizations at an early stage of the synthesis phase and the object code generation. [12]

The ICG serves as an intermediary between the analysis of the source program and the generation of the object code. In particular, it receives the annotated syntactic tree, the result of the semantic analysis, and translates it into an intermediate machine-independent language. This representation can then be processed by the Object Code Generator, which is responsible for producing the assembly or machine code. The position of this component within the synthesis phase of the compiler is illustrated in Figure 9.

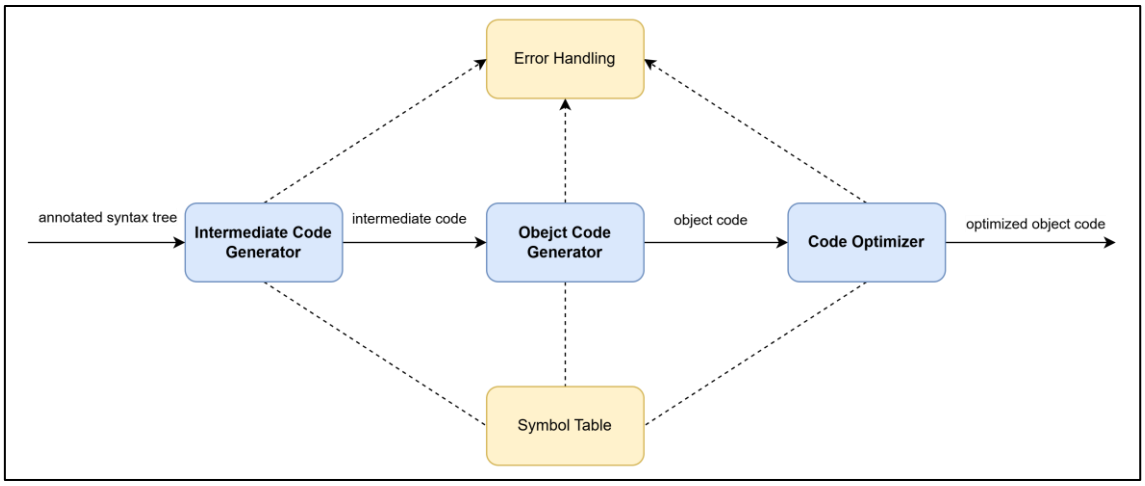

Figure 9 Components of the synthesis phase of a compiler

The GCI operates on an annotated syntactic tree. By employing Syntax-Driven Definition or Translation Schema methodologies, the system translates each source language construct into instructions for the intermediate representation. The attributes associated with the grammatical symbols of the tree facilitate the generation of the intermediate code, which stores information such as data type or memory locations.

The three-address code is the most utilized technique for this intermediate representation. This code employs instructions comprising up to two operands and a result, rendering it appropriate for representing both expressions and statements present in the original program. To illustrate, consider the following example: `x = y + z`. This three-address code enables the representation of arithmetic operations such as the sum of two numbers.

In practice, three-address code instructions are implemented as quadruples, a structured representation that facilitates subsequent translation into machine code. Each quadruple is comprised of an operator, two operands, and a result, which is expressed in the form (operator, operand1, operand2, result). [13]

Three-address code	Quadruple	Example
`x = y op z`	`(op, y, z, x)`	`(+, a, b, z)`
`x = op y`	`(op, y, , x)`	`(-U, y, , x)`
`x = y`	`(=, y, , x)`	`(=, 7, , b)`
`goto etiq`	`(goto, , , etiq)`	`(goto, , , end)`
`if x relop y goto etiq`	`(goto_opr, x, y, etiq)`	`(goto>, a, 5, end)`
`param x`	`(param, x, , )`	`(param, 24, , )`
`param& x`	`(param&, x, , )`	`(param&, 9, , )`
`call etiq`	`(call, etiq, , )`	`(call, label_4, , )`
`x = call etiq`	`(call, etiq, , x)`	`(call, label_4, , b)`
`return`	`(return, , , )`	`(return, , , )`
`return x`	`(return, , , x)`	`(return, , , 37)`
`etiq:`	`(: , etiq, , )`	`(: , end, , )`

Table 4 Implementation of a three-address code

Table 4 illustrates the structure of a generic intermediate code, including its 3-address code and quadruples.

2.1.6 Runtime Environment

The runtime environment of a compiler is the set of structures and resources provided by the system at compile time for the execution of the object code generated by the compiler. It is not a module of the compiler itself, as it belongs to the runtime.

It encompasses the memory, data structures, and mechanisms necessary to ensure that the code instructions are executed efficiently and correctly. Consequently, the design of the runtime environment is of great importance for optimizing resources and developing an efficient compiler.

When an operating system (OS) executes a program, it allocates a specific memory area for its use. It is essential that this memory area contains several key areas that are integral to the operation of the program. Initially, the object code of the program must be copied into a memory region. Moreover, an area is required for the storage of static data, which encompasses information such as constants, global variables, and character strings. A dynamic memory area is also a requisite component of the system, typically organized in a stack or heap structure.

The compiler is aware of the size of the code area and the static data area, but is unaware of the size of the stack and heap. This distinction allows the data offset in the Symbol Table (TS) to be made with respect to the starting address of the data area, regardless of how the memory is organized.

The activation record is a contiguous memory structure that contains all the information necessary for the execution of a subprogram. Upon the activation of a subprogram, a new activation record is created, which is then discarded once the subprogram has completed its execution. This prevents the unnecessary occupation of memory.

The activation record encapsulates seven distinct fields:

- **Parameters**: The caller stores the values of the parameters that will be handled by the called subprogram. This field facilitates the passing of arguments between functions during execution.
- **Local variables**: Space where the values of the local variables of the subprogram are stored. This field is essential for managing the internal state of each function or procedure during its execution.
- **Temporary data**: Stores temporary data such as intermediate results of calculations or temporal variables generated in the intermediate code. This area is crucial for operations that do not require persistence beyond the current function's execution.
- **Machine state**: Stores machine registers before calling the subprogram, and the return address, needed for later restoration. This field ensures that the state of the machine can be restored after the subprogram call.
- **Return value**: The location where the subprogram stores the value to be returned to the calling subprogram. This field holds the result of the subprogram's execution that will be passed back to the caller.
- **Control pointer**: Points to the activation record of the calling subprogram. This pointer helps maintain the correct sequence of function calls and returns.

- **Access pointer**: Points to the activation record of the subprogram where the current subprogram (parent subprogram) is defined, allowing access to non-local variables. This pointer enables the subprogram to access variables that are not local to it but are defined in higher-level subprograms.

The fixed fields of an activation record are the machine state and the control pointer (although the latter may be omitted in some cases). The compiler can determine the size of the activation record for each function, as it knows the size of each field at compile time.

The process of creating an activation record consists of two sequences: the call sequence and the return sequence.

Call sequence:

1. Allocate a memory area for the activation record, using a pointer to an empty space.
2. Copy the values of the actual parameters into the parameters field.
3. Save the machine state, including the necessary registers and the return address.
4. Set the control pointer, which is provided by the caller.
5. Configure, if needed, the access pointer.
6. Jump to the subprogram.

Return sequence:

1. Restore the machine registers saved in the machine state field.
2. Restore the original activation record using the control pointer.
3. If the function returns a value, transfer the return value to the calling subprogram.
4. Jump to the instruction stored in the machine state.

There are several strategies for the allocation of activation records, each with its own characteristics and limitations [14]:

- **Static allocation**: In this strategy, a fixed memory region is assigned to each subprogram. The activation record of a subprogram will always be located at the same address, which allows the addresses of all subprograms and their fields to be known. However, this strategy does not support recursive languages, variable-sized data types, and is not compatible with dynamic memory.
- **Stack allocation**: Activation records are stored in a stack, where they are pushed as procedures are called and popped when they finish. This strategy allows recursion, as a copy of the activation record is maintained for each recursive call, enabling variables to have different values in each call. Additionally, since the pointer to the previous activation record is always available, the control pointer can be omitted. The main limitation is that values are not preserved between calls.
- **Heap allocation**: In this strategy, activation records are created in a memory area within the heap, allowing the location of each record to be chosen. This allows a called function to survive its caller and facilitates recursion, as each call creates a new activation record. However, this strategy is more complex due to memory fragmentation in the heap, requiring a garbage collector for management.

2.1.7 Object Code Generation

The Object Code Generator is one of the final stages in the compilation process. Its function is to transform the intermediate program instructions into a language that can be executed by the target machine. The object code comprises both executable instructions and the data necessary for program execution. [15]

This component is situated after the intermediate code generator or, if an intermediate code optimizer exists, subsequent to that component. It may be regarded as the ultimate stage of the compiler process, preceding the object code optimizer. Figure 9 illustrates the position of the object code generator within the compiler components.

In order to translate the intermediate code, the object code generator translates that code into instructions that are specific to the target architecture. Furthermore, it performs an address assignment for the various symbols and variables within the program, considering the different scopes and their requisite access permissions.

The object code generator implementation is typically based on a predefined translation strategy utilizing a template. The template in question associates each instruction of the intermediate code with specific instructions of the object code. This implementation strategy simplifies the generator design and ensures the systematicity of the process.

In the design of the translation template, it is of great importance to consider the various types of variables and their placement in memory, in accordance with the design of the runtime environment. The generation of the object code is dependent on the location of the variable in question, whether that be global, local, or non-local. The generator will therefore generate the necessary instructions to handle each variable type. [16]

2.1.8 Error Handling

The function of the error handler in a compiler is to flag errors that arise during the various stages of the compilation process. Additionally, it is responsible for reporting errors to the user and implementing a recovery strategy. [17]

As illustrated in Figure 1, the error handler is linked to all phases of the compiler, as errors may arise in any of these phases.

In the event of an error being identified, several potential scenarios may arise. These include:

- The error is detected, the user is informed of it, and the analysis is terminated. In this instance, the compiler will only report the initial error encountered, despite the potential for subsequent errors to exist.
- Identify an error and communicate this information, subsequently implementing a recovery plan. In this instance, the compiler persists in its analysis of the program despite the identification of an error. As a result, the user is able to be informed of the various errors that exist.
- Detect an error, report it, and implement a solution. This particular scenario is uncommon in the context of compilers, primarily due to the inherent complexity associated with the implementation of effective error correction strategies.

A discussion of the various types of errors that may occur during the compilation process is presented below:

- **Lexical errors:** These are identified by the scanner. Such errors are typically the result of character substitution, erroneous character inclusion, or character omission. Examples of such errors include the utilization of a string that is not permitted within the language in question, or the misspelling of a comment in accordance with the linguistic norms of the language.
- **Syntactic errors**: These are identified by the syntactic analyzer. Similarly, as with lexical errors, these can be attributed to the substitution, omission, or inclusion of lexemes that violate the syntactic rules of the language. An example would be the omission of the default block within the switch in a language in which its presence is obligatory.
- **Semantic errors**: These are identified by the semantic analyzer. Such errors are the result of a failure to adhere to the established semantic rules of the language in question. An illustration of this would be the utilization of an undeclared variable in the context of a language that requires the declaration of variables. Another potential error is the declaration of the same variable twice.
- **Execution errors:** These errors manifest during the execution phase and are typically associated with arithmetic operations or data input and output errors. One illustrative example is a division by zero. In this phase, the compiler is no longer able to detect these errors. Consequently, it is crucial to contemplate this type of error and exercise control over it during the synthesis phase.
- **Errors that may manifest at any stage of the compilation process**: In this instance, the errors could be attributed to deficiencies in the compiler design.

As previously stated, one of the primary responsibilities of the error handler is to communicate any encountered errors to the user. In order to ensure the efficacy of error messages, it is essential to adhere to certain recommendations during the design process.

Firstly, the error message should identify the precise location of the error by providing the line number. Secondly, the message should be written in a manner that is accessible and user-friendly. Thirdly, the message should contain as much information as possible related to the error. [18]

2.2 Object Oriented Programming

In computer science, object-oriented programming (OOP) represents a model of software design that is centered on the concept of objects, as opposed to the more traditional approach of focusing on functions and logic. An object can be defined as a data field that possesses distinctive attributes and behaviors.

The origins of this model can be traced back to the 1960s, with the development of the Simula language by Ole-Johan Dahl and Kristen Nygaard. This language was the first to introduce the concepts of class, object, and inheritance.

Subsequently, in the 1970s, the term "object-oriented" was coined with the creation of the Smalltalk language. This language implemented the model in its currently known form, incorporating the transmission of messages between objects, encapsulation, and inheritance.

In the 1980s, object-oriented programming gained popularity with the emergence of languages such as C++, which incorporated object-oriented features into the C programming language. Another notable development was Objective-C, which combined the Smalltalk and C languages. In the 1990s, Java and Python played a pivotal role in the consolidation of object-oriented programming. Java became a particularly influential language in the field of software development. Furthermore, the Python programming language exhibited a gradual adaptation of this paradigm.

As early as 2000, Microsoft launched C#, which was influenced by C++ and JavaScript, as the core language for its .NET platform. Over time, JavaScript also underwent an evolution towards OOP with the introduction of ES6 in 2015. In the present era, OOP remains a dominant paradigm in software development, undergoing adaptations to accommodate new technologies and trends while retaining its significance within the industry. [19]

In OOP, the principal development element is the "object". An object is an entity that represents an idea through data and the operations that can be performed on it. Objects are instances of classes, which are templates that define the common characteristics and behaviors that objects will have. This approach allows for greater modularity, flexibility, and reusability of the code. Derived from this unique approach are the following fundamental features of object-oriented programming [20]:

- **Encapsulation**

  It is defined as the grouping of data and related operations into a single unit. This mechanism enables the regulation of access to the aforementioned information, preventing its direct retrieval from external sources.

  In order to gain access to this information, it is necessary to utilize specific methods. This makes it possible to safeguard the internal state of objects and to regulate access to information with precision.

  The Code 2 example employs encapsulation to safeguard the license plate data (`$plate`) from direct and uncontrolled access from external sources. This approach ensures that any alteration or access is conducted through the prescribed methods, thereby facilitating a more secure and consistent management of the object data.

```
class Car {
    private $plate;
    // Constructor
    public function __construct($marca) {
        $this->plate = $plate;
    }
    //Method to obtain the license plate
    public function getPlate() {
        return $this->plate;
    }
    //Method to change the car's license plate
    public function changePlate($newPlate) {
        $this->plate = $newPlate;
    }
}
```

Code 2 Encapsulation example

- **Abstraction**

  This fundamental characteristic enables the presentation of an object's essential functionality without the necessity of detailed knowledge regarding its implementation. In this manner, internal processes that are irrelevant to the observer can be concealed. This allows developers to effortlessly augment the functionality of objects or implement modifications when required.

  In example Code 3, the abstraction is employed to delineate a generic structure for notifications within the abstract class Notification. The abstract method `send()` defines the structure of the abstract class Notification. This method states that each type of notification (such as SMS or Email) must implement its own send version. Thanks to this structure, it is straightforward to add any new type of notification. Furthermore, it also hides the details of the operation of each type of notification.

```
abstract class Notification {
    protected $recipient;
    //Constructor
    public function __construct($recipient) {
        $this->recipient = $recipient;
    }
    // Abstract method to send the notification
    abstract public function send();
    // Common method to get the recipient
    public function getRecipient() {
        return $this->recipient;
    }
}
// SMS class that extends Notification
class SMS extends Notification {
    public function send() {
        return "Sending SMS to " . $this->recipient;
    }
}
// Email class that extends Notification
class Email extends Notification {
    public function send() {
        return "Sending Email to " . $this->recipient;
    }
}
```

Code 3 Abstraction example

- **Inheritance**

  It defines the relationship between classes. It permits one class to derive from another, thereby acquiring its attributes and methods. This feature is useful for creating hierarchies and sharing common behaviors between related objects, which increases the reuse of code but also the complexity of it.

  As illustrated in Code 4, the vehicle structure is created through the use of inheritance, which establishes a hierarchical structure. The base class, Vehicle, defines common properties and methods, such as the brand.

  Inheritance allows the derived classes, namely Car and Truck, to reuse code from the Vehicle class without redefining it. Each derived class also introduces specific, unique methods: Car includes the `drive()` method, while Truck has the `loadCargo()` method. This approach encourages the

reuse of code and facilitates the modular design of the code base, thereby facilitating maintenance and extension of the code base.

```
class Vehicle {
    protected $brand;
    public function __construct($brand) {
        $this->brand = $brand;
    }
    public function getBrand() {
        return $this->brand;
    }
}
// Derived class Car
class Car extends Vehicle {
    public function drive() {
        return "Driving a car.";
    }
}
// Derived class Truck
class Truck extends Vehicle {
    public function loadCargo() {
        return "Loading cargo.";
    }
}
```

Code 4 Inheritance example

- **Polymorphism**

  This concept enables an operation to be executed in a variety of possible ways, depending on the context in which it is utilized. This suggests that a single method may exhibit different behaviors when invoked by distinct objects. This feature helps developers avoid duplicating code and to arrange it in a more straightforward manner.

  Code 5 illustrates the concept of polymorphism through the base class, Shape, and its derived classes, Circle and Rectangle. Each class implements its own `area()` method.

  The `printArea()` function is applicable to any object of the Shape type. Consequently, the same method can operate in different shapes (such as Circle or Rectangle) without requiring knowledge of their specific details. When the function is invoked, the method designated as appropriate for the given object is executed.

  This approach fosters flexibility and extensibility. The addition of a new shape would entail the implementation of a new `area()` method, without any alteration to the existing code.

The primary distinction between functional programming (FP) and object-oriented programming lies in their differing approaches to code structuring and management. In contrast to object-oriented programming, functional programming is a function-centric approach. Each function receives data, processes it, and returns a result without any alterations to its internal state. The principal benefit of this approach is the ability to perform parallel processing and to trace the flow of data. In contrast, OOP groups data into objects, allowing for modifications to the state of these objects and interactions between them.

```
class Shape {
    public function area() {
        return 0;
    }
}
// Derived class for Circle
class Circle extends Shape {
    private $radius;
    public function __construct($radius) {
        $this->radius = $radius;
    }
    public function area() {
        return pi() * pow($this->radius, 2);
    }
}
// Derived class for Rectangle
class Rectangle extends Shape {
    private $width;
    private $height;
    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }
    public function area() {
        return $this->width * $this->height;
    }
}
// Function to print the area of any shape
function printArea(Shape $shape) {
    echo "The area is: " . $shape->area() . "\n";
}
```

Code 5 Polymorphism example

Furthermore, whereas PF places an emphasis on higher-order functions and function composition to address complex problems in a modular manner, OOP relies on code reuse and entity modeling through concepts such as inheritance and polymorphism. This renders FP more appropriate for tasks that require mathematical operations or data processing, whereas OOP is more conducive to applications that manage multiple entities with evolving states. [21]

The use of OOP offers several advantages, including the ability to modularize code, reuse code, and scale applications. The features of encapsulation facilitate the maintenance and security of code, while others, such as polymorphism, allow for greater flexibility. Furthermore, the ability to create independent modules facilitates collaboration among development teams, thereby enhancing overall productivity.

However, it also has some disadvantages. These are derived from its excessive focus on data, which can result in the role of algorithms and functions being relegated. This in turn can make implementation difficult in certain contexts. Furthermore, aspects such as inheritance can complicate the code and increase its complexity exponentially. [22]

In the course of developing this project, an object-oriented programming methodology was employed, utilizing the PHP language. In certain instances, it has been combined with procedural programming.

2.3 PHP (PHP Hypertext Preprocessor)

PHP is a free open-source server-side programming language that is utilized for the development of web applications. At the present time, it is the most popular language utilized for back-end development. It is used by a multitude of robust content management systems, including WordPress. It is notable for its user-friendliness and the extensive set of sophisticated capabilities that it offers to more experienced programmers.

The technology in question was originally developed under the name PHP/FI, a set of scripts written in C. These were created by Rasmus Lerdorf, who shared and published the inaugural version of PHP in 1995. Subsequently, the language was rewritten and enhanced with database support and improved interaction with web servers. The next version of the software, PHP 3, was released in 1997. Following the establishment of the Apache Foundation, PHP was incorporated into one of its projects. In 2000, version 4.0 was released, which contributed to the language's growing popularity due to the incorporation of new features, including object-oriented programming. In 2004, version 5.0 of the PHP software was released, featuring significant enhancements to address identified bugs and improve performance. Ultimately, following the release of PHP 7, the latest version, PHP 8, was launched in 2020. [23]

Presently, as evidenced by the usage statistics presented by W3Techs [24], PHP is used on 77.5% of the websites currently in operation. Notable examples of websites utilizing this technology include Facebook.com and Wikipedia.org. Given its extensive use, the technology has attracted a large developer community, resulting in the availability of numerous libraries and frameworks that facilitate development work.

PHP is a server-side programming language, which means that the program is executed on the web server. Figure 10 illustrates the process by which PHP functions. It can be defined as a request-response model. When web content is accessed via a browser, the server transmits a request to the installed PHP interpreter, which executes the code. Subsequently, the web server sends the result to the browser using HTML, which is interpreted and displayed to the user. [25]

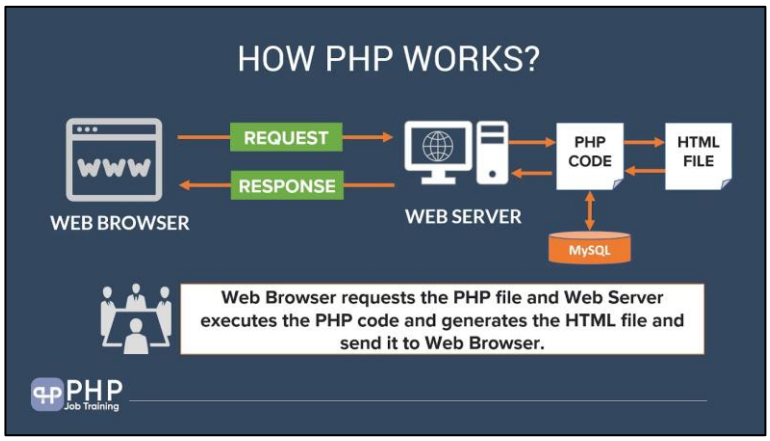

Figure 10 How PHP works

The main functionality of the PHP programming language is its capacity to generate dynamic server-side content. Moreover, it is compatible with a multitude of servers and operating systems. Because of these characteristics,

PHP is an optimal technology for the development of web applications, given its straightforward syntax and its compatibility with technologies such as MySQL.

One of the primary advantages is that it is free and open source, which reduces development costs. In addition, due to its popularity, it has extensive documentation and a supportive community.

However, PHP also has a few drawbacks. On a security level, it has historically been prone to vulnerabilities if good coding practices are not implemented. There are also maintenance problems in more complex projects if a proper structure is not used from the start. [26]

These characteristics and associated advantages make PHP a fundamental technology in the development of the Draco web system, and a role it plays in this project as well.

To develop PHP code, it is necessary to configure the development environment. This entails installing a web server that is compatible with the PHP language and a database management system, should one wish to utilize the latter. In this project, we have utilized the WampServer web server and the MySQL database management system.

The following is an example of a .php document, which illustrates the conventional structure of a web page. These documents can contain HTML, CSS, and JavaScript code in conjunction with PHP, as demonstrated in Code 6.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <?php echo "PHP code"; ?>
    </body>
</html>
```

Code 6 Example .php file

2.4 HTML 5 (HyperText Markup Language)

HyperText Markup Language (HTML) is a computer language that is used to create web pages. It is the principal language utilized in the construction of web pages that make up the World Wide Web (WWW). This language is used to delineate the content and fundamental structure of web pages.

This technology originated in 1980 with the proposal of a new system for sharing documents by the physicist Tim Berners-Lee, a CERN worker. However, the first HTML document was not published until 1991. The foundational document, entitled "HTML Tags", provided an initial description of the language.

In the 1990s, further developments were made to the language. The first official standard was published in 1995 by the Internet Engineering Task Force (IETF) under the designation "HTML 2.0".

In 1996, the W3C (World Wide Web Consortium) assumed responsibility for the publication of HTML standards. Following the release of HTML 4.01, this activity ceased. In response, companies such as Apple, Mozilla, and Opera formed the WHATWG (Web Hypertext Application Technology Working Group), an organization focused on the HTML 5 standard.

In 2008, the W3C published the first official draft. Due to this momentum, W3C resumed standardizing HTML, publishing HTML 5 in October 2014, the most recent version to date. [27]

The primary function of HTML is to organize the content of a web page, including any embedded content or references. The simplicity of HTML allows it to be written in any text document, for which any text editor that supports the .html extension may be used to create and save the document. These documents are interpreted by web browsers, such as Chrome, Edge, Firefox, Mozilla, Opera, and Safari, which display web content.

Any HTML document is composed of a multitude of HTML elements. An HTML element is delineated by a start tag, a content tag, and an end tag, as exemplified by the following syntax:

```
<tag> content </tag>
```

The available tags encompass a broad range, dependent on the content and the intended meaning of the information. [28]

Table 5 illustrates some of the most utilized and significant tags.

Tag	Meaning
<body>	Represents the content; there can only be one per document.
<nav>	Defines the content of the website's navigation section.
<main>	Defines the main content, only one is allowed per document.
<section>	Defines a section.
<aside>	Allows for the inclusion of additional website content.
<h1> <h2> <h3>…	Used as header titles in the document.
<header>	Defines the headers of the webpage.
<footer>	Defines the webpage footer.
<p>	Used to write paragraphs of text.
	Creates ordered lists.
	Creates unordered lists.
<figure>	Used to include an embedded figure.
	Creates a container.
<a>	Allows the addition of hyperlinks.
	Defines the accompanying text as important.
 	Inserts a line break.
<img>	Includes an image.
<table>	Defines a table.
<th> <td>	Defines table cells.
<form>	Defines a form.

Tag	Meaning
`<label>`	Used to add a title or name to other elements.
`<input>`	For data input in a form.
`<button>`	Represents a button in a form.

Table 5 HTML tags

The example that follows (Code 7) represents a basic HTML document, which serves to illustrate the fundamental structure of a web page:

```
<html lang="en">
    <head>
        <title>Page title example </title>
    </head>
    <body>
        <h1>Heading 1</h1>
        <p>Paragraph</p>
    </body>
</html>
```

Code 7 HTML page structure

HTML is a free and open-source language that can be utilized with ease in any text editor, as previously stated. Furthermore, since it is the standard language for the Internet, it is guaranteed that the majority of web browsers will be able to interpret such documents. Additionally, HTML is relatively straightforward to learn thanks to the abundance of information available about the language.

However, HTML has certain limitations. It does not allow for the creation of dynamic content, and often it is necessary to make use of other languages, such as CSS or JavaScript, to enhance the quality of web content designs.

This work uses HTML 5, the latest published standard. It includes numerous advances over previous versions, most notably the inclusion of multimedia elements like `<audio>` and `<video>`, the redefinition of certain elements, new tags for simpler structuring with semantic information, client-side local storage, integration of scalable vector graphics, MathML for mathematical formulas, and APIs (Application Programming Interfaces). [29]

This version improves the compatibility and adaptability of web content. Consequently, the DRACO web system is made more accessible from devices other than computers, including tablets and cell phones.

2.5 CSS 3 (Cascading Style Sheets, version 3)

Cascading Style Sheets (CSS) is a style language that describes the way Hypertext Markup Language (HTML) documents are displayed. It provides instructions on the representation of each element and its structure on the screen.

The concept of style sheets originated in 1970, driven by the necessity to apply distinct formatting styles to electronic documents. However, they subsequently gained notable importance with the rise of the Internet and the subsequent development and utilization of the HTML language.

W3C assumed responsibility for the standardization of web-related technologies. Several proposals were put forth with the aim of standardizing style sheets, including CHSS (Cascading HTML Style Sheets) and SSP (Stream-Based Style

Sheet Proposal). Ultimately, CSS was selected, and the first official publication was in 1996 under the name CSS Level 1. Currently, the most recent official specification is CSS3, which introduces new functionality to those present in CSS2.1, allowing compatibility with the previous version. [30]

CSS enables a web developer to specify the desired visual presentation and user interface for a web page. Figure 11 shows a diagram of how CSS works. The browser loads the HTML document, converts it into a DOM (document object model), and then searches for the resources linked to that document, including the CSS. Subsequently, the style sheet rules are parsed and the instructions are applied to the elements. Subsequently, the content is rendered by the browser and displayed on the screen. [31]

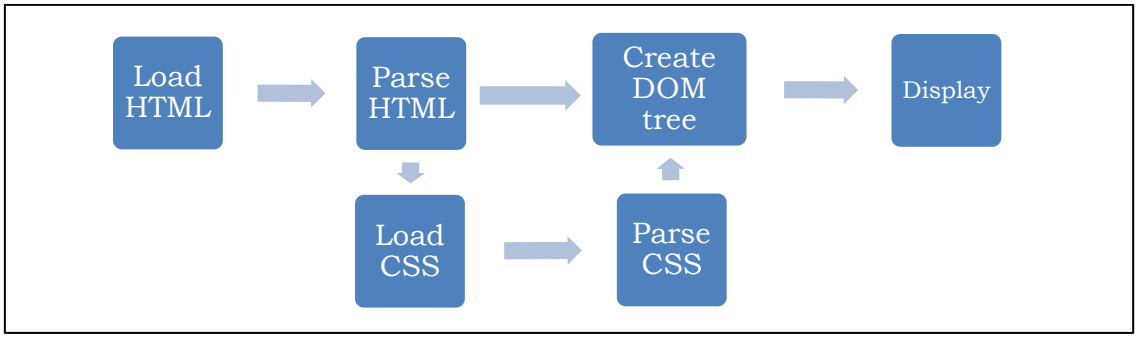

Figure 11 How CSS works

CSS enables the modification of numerous properties of web page elements. Among the most useful and widely used of these are those that apply color to text and backgrounds, those that format text, those that modify the font, and those that apply specifically to lists and tables. Table 6 provides an illustration of the various properties associated with the text of a web page.

Style	Description
`width`	Width of element.
`background-color`	Background color of element.
`color`	Color of text in element.
`text-align`	Horizontal text alignment of element.
`border`	Border thickness and style of element.
`padding`	Padding (white space around content) of element.

Table 6 Common CSS properties

There are three distinct methods for incorporating CSS code into a web document. [32]

- **External CSS**

  The file comprises a set of CSS rules that can be applied to an HTML document. To utilize this technique, it is necessary to define a `<link>` tag within the HTML files in which the CSS will be applied, referencing the stylesheet in the following manner:

  ```
  <link rel="stylesheet" href="styleSheet.css">
  ```

- **Internal CSS**

  This mode involves insertion of CSS code into the HTML file. To accomplish this, it is essential to utilize `<style>` tags within the section of the document

header delineated by the <head> tag. Code 8 exemplifies the use of internal CSS code.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <style>
        body {
              background-color: blue;
        }
        h1 {
              color: red;
              padding: 60px;
        }
        </style>
    </head>
    <body>
        <p>Example </p>
    </body>
</html>
```

Code 8 Internal CSS example

- **Inline CSS**

  This style enables the individual styling of a specific HTML tag. The ability to easily insert rules for a given element is a notable advantage. However, due to concerns regarding readability and page load time, this approach is not widely recommended.

  This is achieved by utilizing the style attribute within an HTML tag, as illustrated in the following example:

```
<h1 style="color:white;">Inline CSS example</h1>
```

The Cascading Style Sheets language offers several advantages when developing web content. It is a highly flexible language that provides extensive control over the style of HTML elements. Additionally, it enables the separation of a web page's structure and presentation, facilitating the maintenance of a clean and structured code.

However, learning, and mastering CSS is a challenging process, and inconsistency in style between different browsers is a notable drawback. [33]

In the development of this project, CSS version 3 has been employed for the styling of various elements within the DRACO web system. Regarding the manner of incorporating CSS code, the three methods mentioned above have been integrated according to the particular requirements of each situation.

2.6 SQL (Structured Query Language)

SQL is a programming language designed for use with relational databases. It enables database management systems (DBMS) to perform a wide range of operations, including the creation of tables, the insertion, updating and deletion of records, and the execution of queries.

The language was initially conceived in the early 1970s under the designation SEQUEL. The first version of this language was developed by Donald D. Chamberlin and Raymond F. Boyce at IBM. The designation SEQUEL was

derived from the acronym "Structured English Query Language". The objective of the language was to facilitate the manipulation of relational databases.

In 1981, IBM released the first commercial version of the product, which was subsequently renamed SQL. This technology rapidly became the standard for handling relational databases, being adopted by major vendors such as Oracle, Sybase, and Microsoft.

Years later, the ANSI (American National Standards Institute) and ISO (International Organization for Standardization) published the official SQL language standards. The language has undergone continuous evolution to the present day. Currently, it is a crucial language in the management of relational databases and has become a fundamental tool for companies in all sectors with regard to data management. [34]

SQL is an interpreted language, which means that it is processed by a server machine before proceeding to return a result. As stated above, SQL is a language designed for use with relational databases. In this type of database, the information is organized into interrelated tables through the use of keys.

Various types of database operation can be performed using SQL. For this purpose, it has numerous commands that offer a wide variety of options for developers. The following are the three main groups of commands. [35]

- **Data Definition Commands**

  These allow you to define how information is stored. This group includes functions for creating and deleting tables, defining columns and primary and foreign keys. Some examples are:

  - `CREATE DATABASE`: creates a database.
  - `DROP DATABASE`: deletes a database.
  - `CREATE TABLE`: creates a new table.
  - `ALTER TABLE`: modifies an existing table.
  - `DROP TABLE`: deletes an existing table.

- **Data Manipulation Commands**

  These allow you to manipulate data in a database. It includes data insertion, update and delete operations. The most important commands are:

  - `INSERT`: adds new data.
  - `UPDATE`: updates the value of existing records.
  - `DELETE`: erases existing data.
  - `REPLACE`: modify or replace data.

- **Data Query Commands**

  These allow data to be accessed and retrieved in a controlled manner. The main commands are:

  - `SELECT`: selects the columns to be retrieved.
  - `WHERE`: determines the selected records to be displayed according to the desired condition.
  - `JOIN`: joins several tables to obtain crossed data.
  - `GROUP BY`: separates records into specific groups.
  - `ORDER BY`: sorts the records according to the desired order.

In conclusion, SQL represents a fundamental tool for the management of relational databases. It enables efficient and rapid processing of large data volumes. As a widely accepted standard, it facilitates interoperability between

different systems and makes it relatively easy for new developers to learn the language.

However, it also has some disadvantages, such as its limitation to tabular structures. Furthermore, query performance may be affected in the case of poorly optimized databases.

In the development of this work, the SQL language was used for all issues related to the interaction with the DRACO web system database and information about the source language.

2.7 Gamification

Gamification is a technique that employs the concepts and mechanics of games in non-game contexts with the objective of motivating and improving user participation and learning.

The origins of gamification can be traced back to several instances where companies employed reward mechanisms. One of the most prominent examples was the introduction of a point program by American Airlines in 1981. The objective of this system was to attract customers in exchange for redeemable rewards, thus fostering customer loyalty.

The term "gamification" was first used in 2002 by programmer Nick Pelling. In the 21st century, it has undergone a notable evolution and experienced a surge in popularity. [36]

The efficacy of gamification is now well established and, as a result, this technique has been adopted in a number of fields, including sports, health, business, and education. Notable examples of successful implementations include: Kahoot, Duolingo, and Starbucks Rewards.

The application of gamification techniques enables users to motivate and concentrate. This is accomplished through the incorporation of diverse game elements. As illustrated in Figure 12, the author Kevin Werbach identifies three distinct levels. [37]

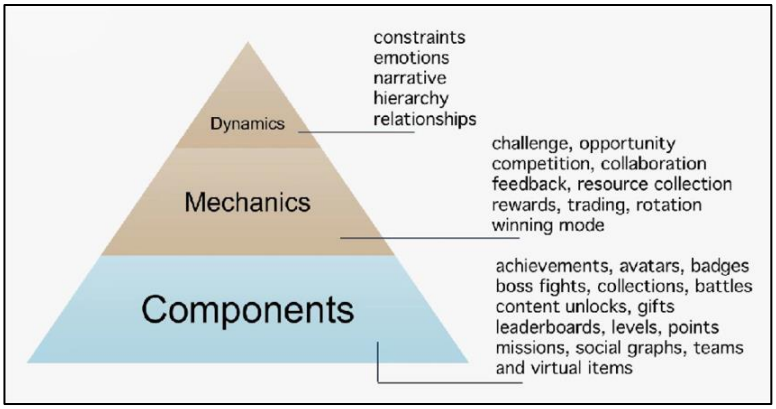

Figure 12 Pyramid model of gamification elements

- **Components**

  This constitutes the base of the pyramid. The formation of this concept is derived from the integration of game design elements and resources. Some of these include:

  - Avatars: a visual representation of the user.

- Levels: sequential stages that facilitate advancement.
- Rankings: A representation of the user's classification.
- Points: A quantitative representation of progress.
- Badges: A visual representation of achievements.

- **Mechanics**

  This represents the intermediate level of the pyramid. The formation is comprised of the fundamental components of the game. These mechanics are responsible for stimulating the dynamics. A variety of mechanical principles are employed, including:

  - Rewards: offered in a variety of forms, including tangible items such as medals or intangible benefits.
  - Missions: specific objectives assigned to the user, with a designated timeframe for completion.
  - Challenges: tasks that require a certain level of skill or knowledge to complete.

- **Dynamics**

  This represents the highest level of the pyramid. The term refers to the user's behavior and the needs that the game satisfies. Several dynamics may be identified, including:

  - Emotions: including the user's satisfaction in achieving challenges.
  - Status: the acknowledgment of the user's attainment of a specific level within the system within the user community.
  - Progression: the user's perception of advancement towards the attainment of their objectives.

Incorporating gamification into the educational process offers a multitude of benefits. It fosters user participation, improves participation, offers immediate feedback, and cultivates a constructive atmosphere surrounding the field of education.

Nevertheless, the technique is challenging to implement correctly. Achieving optimal equilibrium is often a significant challenge. Furthermore, if the technique is not applied correctly, it can potentially compromise the benefits previously outlined.

This project is framed within the DRACO web system, which is designed to facilitate learning for students through the use of gamification as its primary mechanism.

3 Problem Statement

3.1 Introduction

This document introduces the Software Requirements Specification (SRS) for the implementation of a new functionality within the DRACO web platform. The objective is to incorporate an automated practice evaluation system for the *Traductores de Lenguajes* course.

This system is based on the development of a generic language compiler that will be integrated into DRACO to automatically correct student submissions. The present work contributes to this broader project by implementing several core components of the compiler, including the intermediate code design (quadruples), quadruples compiler, and the analysis of the assembly code. Other components, such as the source code analyzer, are being developed in parallel by other contributors.

This SRS has been validated by the DRACO system administrator and is considered free of critical errors, within the usual reservations in the context of software development.

The structure and content of this document follow the guidelines established in the IEEE Recommended Practice for Software Requirements Specifications, ANSI/IEEE Standard 830-1998 [38].

3.1.1 Purpose

The purpose of this document is to define the functional and non-functional requirements of the software components developed in this Master Thesis, which are to be integrated into the DRACO web platform.

The intended audience of this document includes the system developers responsible for integrating these components, as well as the DRACO system administrator. Given that the document has already been validated, it is expected to serve primarily as a reference during the design and implementation phases.

3.1.2 Scope of the System

This document outlines the requirements for the development of key components of a generic compiler to automatically correct student assignments for the *Traductores de Lenguajes* course within the DRACO platform. The system includes:

- Intermediate language design (quadruples)
- Quadruples compiler (translation from quadruples to assembly)
- Assembly code analyzer.

The target users are students of the *Traductores de Lenguajes* course. This project is limited to the components mentioned above and does not include other parts of the broader compiler system, which are being developed separately.

3.1.3 Glossary

This section defines key terms, acronyms and abbreviations used throughout this document.

3.1.3.1 Definitions

Term	Definition
Assembly Code	A low-level programming language that represents machine instructions in a human-readable form
Compiler	A program that analyzes and translates code written in a high-level language into a low-level language
Intermediate Code	A representation of the source code that is independent of both the programming language and the target machine. It is used as an intermediate step during compilation
Intermediate Code Generation	Compilation phase in which the source program is translated into an abstract, platform-independent representation, facilitating further analysis and transformation
Lexical Analysis	Compiler phase responsible for converting a sequence of characters into a sequence of tokens by identifying lexical patterns
Object Code Generation	Final compilation phase in which the intermediate representation is translated into low-level assembly or machine code, producing executable object code
Quadruples	An intermediate representation of code used by the compiler, consisting of a set of four elements that represent operations, operands, and results
Semantic Analysis	Phase of the compiler that verifies the correctness of the program's meaning
Syntactic Analysis	Process of checking the structural correctness of the token sequence according to the grammar of the language
Token	A unit of meaning extracted from the code during lexical analysis.

Table 7 Definition of terms used in the problem statement chapter

3.1.3.2 Acronyms and abbreviations

Acronym / Abbreviation	Meaning
DB	Database
DFA	Deterministic Finite Automaton
DRACO	*Dinámica de Refuerzo para el Aprendizaje de Compiladores* which means Reinforcement Dynamics for Compiler Learning
PDL	*Procesadores de Lenguajes*

Acronym / Abbreviation	Meaning
PHP	PHP Hypertext Preprocessor
SRS	Software Requirements Specification
TDL	*Traductores de Lenguajes*

Table 8 Acronyms and abbreviations used in the problem statement chapter

3.1.4 References

This section lists the documents and resources referenced throughout the specification and development of the software components described in this chapter. These references provide the structural foundation, integration context, and technical support necessary for the implementation.

The following key references were used:

- *IEEE Recommended Practice for Software Requirements Specifications* (IEEE Std 830-1998), which outlines the structure and guidelines for writing software requirements specifications. This standard has been followed to ensure consistency and clarity in this document.
- The *DRACO Web Platform*, which serves as the environment where the developed components will be integrated. The platform is available at: https://dlsiis.fi.upm.es/draco/.
- The *ENS 2001 Assembler Language User Guide*, which describes the syntax and usage of the assembly language used for student submissions in the course *Traductores de Lenguajes*. This guide can be obtained at: https://dlsiis.fi.upm.es/traductores/Documentos/ENS2001.pdf.

A complete list of all sources and materials cited throughout this work can be found in the Bibliography chapter.

3.1.5 Overview

This document includes a general description of the problem to be solved, the functional and non-functional requirements for the components developed in this Master Thesis. The document also defines design constraints, expected system performance, and technological attributes. Each section provides the necessary details for understanding the scope, goals, and constraints of the project.

3.2 General Description

3.2.1 Product Perspective

This Master Thesis focuses on the development of specific components within the broader project of building a generic compiler, which is intended to be integrated into the Practice section of DRACO web platform to automate the correction of TDL practices.

The generic compiler project includes several stages, from source code analysis to assembly code execution. The present work covers the design and implementation of three essential components: the design of an intermediate representation language based on quadruples, the development of a compiler that analyses and translates quadruples into assembly code, and an analyzer of assembly code.

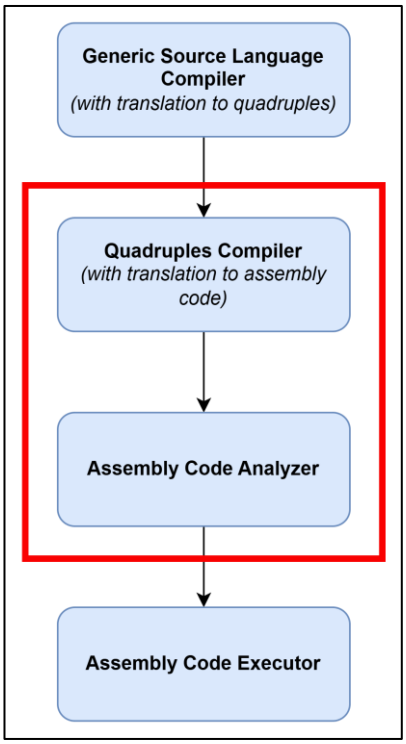

Figure 13 Modules of the generic language compiler

These components are a central part of the generic compiler and are designed to interoperate with the rest of the compiler modules. Figure 13 illustrates the architecture of the general compiler project, with the components addressed in this Master Thesis highlighted. Other modules are being developed by other contributors.

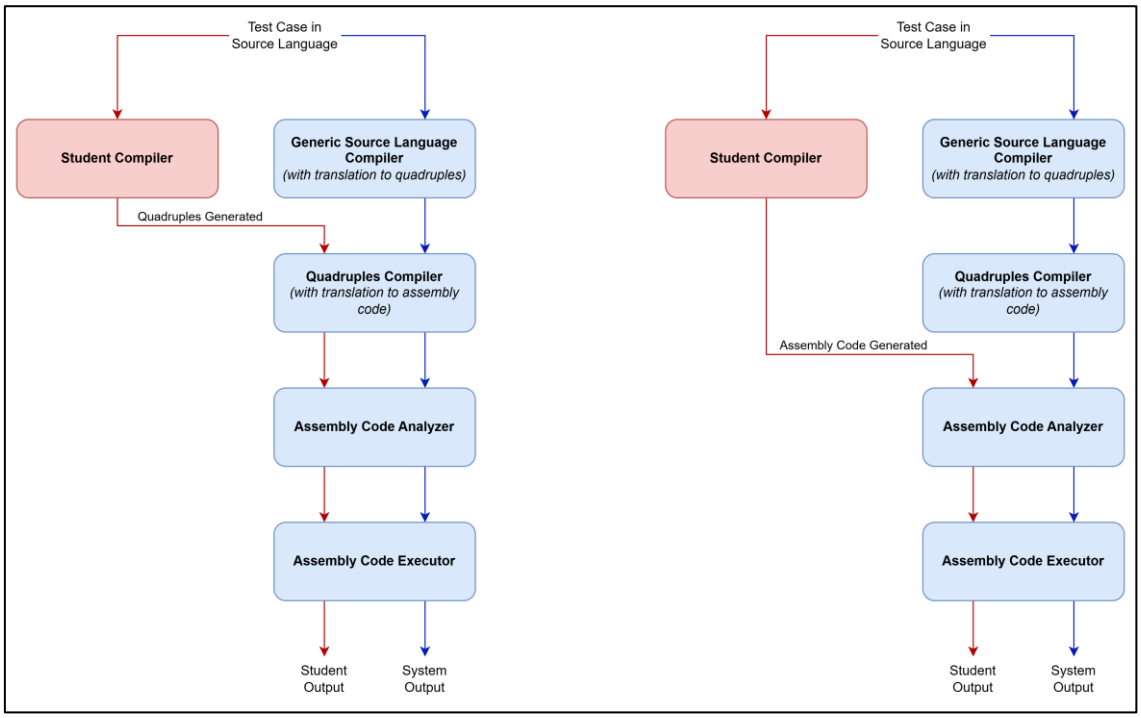

Figure 14 Operational diagram of the activities supported by the system

To clarify how the system operates, the following describes the two types of student activities supported. Both activities are based on a common reference

execution, generated automatically from a test case written in the source language. This test case is compiled by DRACO through all compilation phases until executable ENS code is produced and run, generating the reference output.

In the quadruples-based activity, the student uploads a file containing quadruples, which are analyzed and translated into ENS code. This ENS code is executed, and the result is compared with the reference output.

In the assembly-based activity, the student directly uploads an ENS file, which is then analyzed and executed, and its output is also compared with the reference.

The full operational flow of both activities is shown in Figure 14.

3.2.2 System Functions

The system developed in this Master Thesis provides a set of core functionalities which can be summarized as follows:

- **Quadruples Analysis:** The system reads and parses the input file containing the intermediate representation of a program in the form of quadruples, ensuring its correctness.
- **Assembly Code Generation:** The system translates the validated quadruples into assembly code, specifically in a extended version of ENS 2001.
- **Assembly Code Analysis:** Assembly code undergoes lexical, syntactic, and semantic analysis. This ensures that the code adheres to the rules and structures of the modified ENS 2001 language specification, checking for errors at each level of analysis.

3.2.3 User Characteristics

The system is designed to cater to the needs of different user groups, each with specific characteristics and roles. The primary users of the system are as follows:

- **Students:** The main users of the system are students enrolled in the *Traductores de Lenguajes* course. They will submit their programming assignments in the form of code, which will be analyzed automatically by the system.

  Students are expected to have basic knowledge of programming concepts and familiarity with the specific assembler language (modified ENS 2001) used in the course.

- **Teachers:** The instructors responsible for the course will use the system to monitor student submissions and ensure the correctness of the automated evaluations.

  Teachers are expected to have a deeper understanding of programming languages, compilers, and the specific language being used in the course. They are also responsible for configuring the system, including the definition of the language to be used. Additionally, they will provide any necessary clarifications and adjustments based on the results of the automated evaluations.

- **Developers:** Technical personnel responsible for maintaining and integrating the system into the DRACO platform.

Developers will have advanced knowledge of programming, software development, and the internal workings of DRACO. They will be tasked with implementing and maintaining the components of the compiler system, ensuring its seamless integration with other modules of DRACO.

3.2.4 Constraints

The development and implementation of the system are subject to various constraints, including technological limitations, predefined formats, and integration requirements within the DRACO platform. The key constraints are as follows:

- **Technology Base:** The system will be developed using PHP, in line with the existing technological stack of the DRACO platform. This decision is driven by the need for compatibility and ease of integration into the web-based environment. Consequently, the system's design, architecture, and implementation must adhere to the capabilities and limitations of PHP.
- **File Format and Standards:** The system must operate within the constraints of the file formats defined by the DRACO platform for both the quadruples and assembly code.

  The quadruples file format specification is a critical element, as the system will need to generate, process, and interpret these files according to the established standards used by the course.

  Similarly, the assembly code generated and analyzed must comply with a modified version of the ENS 2001 assembly language, ensuring uniformity in the student submissions and evaluation process.
- **Web-based Environment:** The system will function within a web-based platform, which introduces constraints related to performance, security, and accessibility. The system must be optimized for web deployment, security protocols must also be followed to ensure that the platform is safe for students to use, especially regarding data privacy and code integrity.
- **Database Integration:** The system will be integrated into the DRACO platform's database, which stores student data, assignment submissions, and feedback. The design and functionality of the system must align with the existing database structure to facilitate smooth data retrieval and storage operations. Any modifications to the database must be carefully managed to avoid disrupting other parts of the DRACO system.

These constraints define the boundaries within which the system will operate, guiding the development process and ensuring compatibility with the broader DRACO platform.

3.2.5 Assumptions and dependencies

The development of the system relies on several key assumptions and dependencies:

- **Predefined File Formats:** The quadruples and assembly code file formats, central to the system's functionality, are assumed to remain consistent with the existing specifications provided by DRACO. Any changes to these formats could require modifications to the system's functionality.

- **Version of ENS 2001 Assembly Language:** The system will operate under the assumption that the modified version of the ENS 2001 assembly language used in this project will be adhered to consistently. The system's compiler and analyzer components will be designed to process this specific variant, and any deviations from this version may cause inconsistencies in evaluation.
- **PHP Environment:** The system assumes that the PHP environment in which it will be deployed is stable and consistent with the version used during development. Changes in the PHP version or configuration may affect the system's performance and require adjustments to the codebase.
- **Availability of the DRACO Platform:** The system assumes that the DRACO platform will be available and operational during the development and deployment phases.

These assumptions and dependencies provide the foundational context for the system's design and integration, ensuring that the development process stays aligned with the expected operating conditions and external factors.

3.3 Specific Requirements

This section details the specific requirements of the system developed within the scope of this project. These requirements define the expected behavior, performance constraints, technological dependencies, and quality attributes of the components designed and implemented. The aim is to ensure that the system fulfills its intended functionality.

3.3.1 Functional Requirements

This section presents the functional requirements of the system, specifying the essential behaviors it must exhibit to meet its objectives. Each requirement is defined clearly and verifiably, following a consistent structure to support the design and evaluation processes.

3.3.1.1 Quadruples Analysis

R1.1: Design of the Quadruple Language

A comprehensive and expressive quadruples language must be designed to represent intermediate code. This language shall support a wide range of operators and expressions necessary for translating higher-level constructs into an intermediate form.

The design of the language should be documented in the TDL subject quadruples format document for students.

R1.2: Lexical Analysis of Quadruples

The system shall perform lexical analysis on the quadruples provided in the input file. It must tokenize each quadruple and ensure that all lexical units conform to the expected patterns for operators, operands, and results as defined in the quadruples format. To achieve this, a scanner must be designed and implemented.

R1.3: Syntax Analysis of Quadruples

The system shall perform syntactic analysis to verify that the structure of each quadruple adheres to the grammar rules defined for the quadruples language,

including the correct number and order of components. To achieve this, a parser must be designed and implemented.

R1.4: Semantic Analysis of Quadruples

The system shall perform semantic analysis to verify the correctness of operations within the quadruples, ensuring, for instance, that operand types are compatible with the specified operator and that symbolic references are properly declared and used. To achieve this, a semantic analyzer must be designed and implemented.

R1.5: Error Detection and Reporting

The system shall detect lexical, syntactic, and semantic errors within the quadruples. For each detected error, the system shall generate a clear error message that includes the nature of the error and the line number where it occurred. This information should be presented in a user-friendly manner, allowing students to quickly identify and resolve issues.

R1.6: Error Recovery

In the event that any lexical, syntactic, or semantic error is detected during the analysis, the analyzer must be capable of recovering and continuing to analyze and detect more errors until the end of the document.

3.3.1.2 Assembly Code Generation

R2.1: Design and Adaptation of the ENS 2001 Assembly Language

The necessary modifications to the ENS 2001 assembly language must be defined and documented to support the correct translation of the designed quadruple language.

R2.2: Translation to Assembly Code

The system must translate each quadruple analyzed without errors by the quadruples analyzer into the specified assembly language. This ensures that code generation is performed on valid and consistent intermediate representations.

R2.3: Output File Generation

The system will generate a new file containing the resulting assembly code. This file shall follow the format of the assembly code document specified in the ENS 2001 manual, taking into account the defined modifications.

3.3.1.3 Assembly Code Analysis

R3.1: Lexical Analysis of Assembly Code

The system shall perform lexical analysis on the assembly code provided in the input file. It must identify and classify tokens such as instructions, registers, labels, and constants according to the defined lexical rules of the adapted ENS 2001 assembly language.

R3.2: Syntactic Analysis of Assembly Code

The system shall validate the syntactic structure of each instruction in the assembly code. This includes verifying that the instruction formats, operands, and label declarations follow the grammar defined for the modified ENS 2001 language.

R3.3: Semantic Analysis of Assembly Code

The system shall perform semantic verifications to ensure the correctness of instruction usage in the program. It must detect issues such as the use of undeclared labels, invalid register references, and improper operand types, among other semantic inconsistencies.

R3.4: Context-Aware Analysis Based on File Origin

The system shall distinguish between assembly code files submitted by students and those generated automatically by the quadruples-to-assembly translator. This contextual information must be taken into account during the analysis process, adapting error detection accordingly.

R3.5: Error Detection and Reporting in Assembly Analysis

The system shall identify and report any lexical, syntactic, or semantic errors found in the assembly code. Each error must be accompanied by a clear and descriptive message, including the corresponding line number, to facilitate the debugging process.

R3.6: Error Recovery in Assembly Analysis

In case of errors, the analyzer must continue processing the remaining lines of the assembly code to detect additional issues. This ensures comprehensive feedback is provided to the student in a single analysis run.

3.3.2 Performance Requirements

This section defines the performance constraints that the system must satisfy to ensure a responsive and reliable experience during its use within the DRACO web platform.

Although the number of enrolled students may reach up to 100, simultaneous access by all users is highly unlikely, as activities are not scheduled for specific dates or times. However, the system must be capable of handling the most demanding scenarios in terms of input size and execution time to guarantee usability and scalability.

R4: Quadruples File Analysis

The system shall complete the lexical, syntactic, and semantic analysis of a quadruples file containing up to 150 lines in no more than 3 seconds.

R5: Quadruples-to-Assembly Code Translation

The system shall complete the translation from quadruples to assembly code, generating output of up to 300 lines, in no more than 3 seconds.

R6: Assembly Code File Analysis

The system shall complete the lexical, syntactic, and semantic analysis of an assembly code file containing up to 300 lines in no more than 3 seconds.

3.3.3 Technological Requirements

The technological requirements for the system ensure that it is developed, deployed, and integrated smoothly within the existing DRACO web platform.

These requirements cover the use of specific technologies, compatibility with the current infrastructure, and adherence to security standards, all of which are

crucial for maintaining system functionality and performance while ensuring its integration with other components of the platform.

R7: Compatibility with PHP and Web Server Environment

Since DRACO is based on PHP, the new components must be written in PHP to ensure seamless integration with the platform. The system must also be compatible with the PHP version currently deployed on the DRACO web server and function smoothly within the existing server setup.

R8: Integration with DRACO Database

The system must interact with the existing DRACO database. It should support reading and writing operations to relevant tables, adhering to the data models and formats already in use. Additionally, the system must handle database transactions efficiently to ensure consistent and accurate data operations.

R9: No External Dependencies

To ensure minimal complexity and ease of maintenance, the system should not rely on external libraries or dependencies that are not already included in the DRACO environment. The use of PHP and MySQL as core technologies should be sufficient for implementing all functionalities required by the system, minimizing the need for additional setup or configuration.

3.3.4 Attributes

This section outlines the key quality attributes that the system must meet, focusing on its maintainability, extensibility, reliability, and scalability.

These attributes are essential to ensure the long-term success of the system, particularly in a context where future modifications and additions may be required, or when other developers need to interact with the system to integrate additional modules into the overall project.

R10: Maintainability

The system must be designed in a way that facilitates its maintenance over time. As other developers will need to understand and work on the developed components to integrate them into the generic language compiler project and the DRACO platform, it is crucial that the code is well-structured and documented.

Additionally, it should follow standard coding conventions to ensure that the code remains easily understandable and modifiable. This approach will ensure that any future modifications can be carried out effectively.

R11: Extensibility

The system must be flexible and allow for easy integration of future components. As new modules or features are added to the generic language compiler project, the system should be designed in a way that facilitates these extensions.

The architecture should enable the addition of new components without requiring significant changes to the core functionality of the system, ensuring smooth adaptation to future project requirements.

R12: Reliability

The system must be reliable and capable of recovering appropriately from failures or errors. In the event of an error, the system must ensure that it can

recover in a controlled manner without data loss or significant disruption to its operation.

Error handling and recovery mechanisms must be implemented to ensure the system's stability and minimize the impact of failures, allowing users to experience the least possible disruption in case of unexpected situations.

R13: Scalability

The system must be scalable and maintain good performance even in high-demand scenarios. Although simultaneous access by all users is not anticipated, the system must be capable of handling up to 100 concurrent users.

Additionally, the system should efficiently handle large input files so that performance is not significantly affected as the number of users or data size increases.

4 Development

This chapter delineates the design, implementation, and testing process of the components developed for this Master Thesis. An incremental and iterative methodology was employed, whereby each module was constructed from the requirements previously defined and aligned with the objectives of the project.

4.1 Design

This section presents the design process of the key components developed in this Master Thesis: the design of the quartet language, the quadruples compiler, and the assembly code analyzer.

4.1.1 Quadruples Language

The intermediate quadruples language plays a central role in the development of the generic language compiler for the DRACO platform. This representation serves as a bridge between the generic source language —whose design lies outside the scope of this project— and the target assembly language.

Within the broader project, the translation from source code to quadruples is a fundamental stage, followed by the subsequent translation from quadruples to assembly code. The design of this intermediate language underpins both of these stages and constitutes one of the core contributions of this Master Thesis.

In addition to its role in the compilation process, this intermediate language is also the format that students will use to test the intermediate code generators they develop as part of their coursework. Students will upload the quadruple code they have produced, and the system will automatically analyze and verify its correctness.

The design of the quadruples language was guided by the need for flexibility and expressiveness, with the aim of supporting the translation of a generic source language whose precise definition remains outside the scope of this Master Thesis. For this reason, the intermediate language had to be capable of representing a wide range of operations and programming constructs.

To address this requirement, the language includes a comprehensive set of operators covering arithmetic, logical, relational, memory access, assignment, and control flow operations. This operator set was carefully selected to ensure that the language could represent the necessary semantics of a wide variety of source programs without requiring future modifications.

Each quadruple is composed of four elements: operator, argument 1, argument 2, and result, following the general structure:

```
(OPERATOR, ARGUMENT1, ARGUMENT2, RESULT)
```

The operators are defined in a closed table and represent actions from the source language. The arguments when not required, can be omitted or replaced with hyphens (-).

The elements ARGUMENT1, ARGUMENT2, and RESULT may represent variables, constants, labels, or intermediate values, and are expressed as tuples:

```
{CLASS, value}
```

or, when representing non-local variables:

```
{CLASS, depth, offset}
```

The students' intermediate code generator must produce a plain text file that contains all the quadruples produced. This file must follow the formatting rules below:

- Each line contains a single quadruple.
- Empty lines and comments starting with // are allowed.
- The quadruple must follow the previous syntax mentioned (allowing spaces and tabs between elements)

To ensure the correct use of the language, detailed documentation was created, covering both the language specification and the format that students must follow when generating the quadruples file, this document is provided in chapter Annexes: Quadruples File Format.

The documentation includes the list of valid operators, the expected structure of operands, and the file formatting requirements (structure, encoding, line terminators, etc.). The proposed format was based on the reference document used in the course for the token format in *Procesadores de Lenguajes* (PDL) assignments [39].

An example of valid quadruples is shown in Code 9.

```
( INPUT_ENT,    ,      , {VAR_GLOBAL , -22})
(suma, {VAR_GLOBAL , -22} , { VAR_TEMP, +2} , {VAR_LOCAL , -1})
(PRINT_ENT,    ,   - ,  {VAR_NOLOCAL,  2 , 3})
```

Code 9 Quadruples example

The system is designed to allow students to define their own naming descriptions for operators and argument classes according to their preferences.

These customized names are configured directly in the DRACO web platform, where students must register the chosen identifiers. Internally, the system associates each user-defined name with a unique identifier, allowing the compiler to interpret the quadruples in a consistent manner regardless of the specific terminology selected by each group.

To support the feature that allows students to define their own reserved words for operators and argument classes, dedicated configuration interfaces will be designed and implemented in DRACO web platform. However, the development of these interfaces is beyond the scope of this work. [40]

This project focuses solely on the design of the required database modifications, which consist of the addition of two new tables to the existing system schema. The structure and fields of these tables are shown in Figure 15.

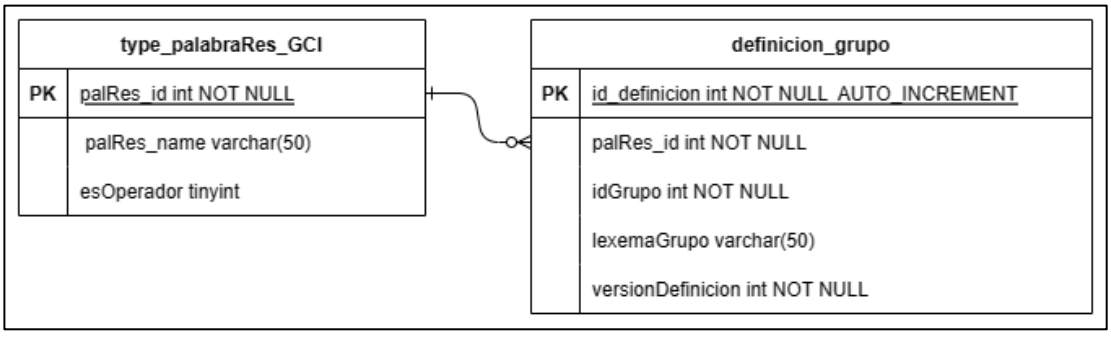

Figure 15 Database schema for custom token definitions for quadruples language

Table "type_palabraRes_GCI":

This table contains the complete list of valid system-defined tokens for quadruples and argument classes.

- **palRes_id:** Primary key that uniquely identifies each token.
- **palRes_name:** Default name used by the system for the token.
- **esOperador:** Boolean field indicating whether the token is an operator or an argument class.

Table "definicion_grupo":

This table stores the custom definitions created by each students group.

- **id_definicion:** Primary key uniquely identifying each entry.
- **palRes_id:** Foreign key referencing the palRes_id field in the type_palabraRes_GCI table.
- **idGrupo:** Identifier of the group defining the token.
- **lexemaGrupo:** Reserved word that the group will use for the corresponding token.
- **versionDefinicion:** Version number assigned to the token definition. It is a decimal value that increases progressively from 0. This versioning mechanism allows groups to redefine their token mappings while preserving older definitions used in previous tests or submissions.

This design ensures that each group's personalized terminology can be consistently interpreted by the compiler, while also maintaining traceability across different definition versions.

4.1.2 Quadruples Compiler

This chapter delineates the development of the Quadruples compiler. It encompasses both the compiler design perspective and the software design perspective of the analyzer

The quadruples compiler is responsible for reading the quadruples file, analyzing its contents, translating the quadruples into assembly code, and detecting and reporting any errors found.

4.1.2.1 Compiler Design

This section describes the design of the quadruples analyzer compiler. It delineates the analysis phases of a compiler: the lexical, the syntax and the semantic analyzer. It also includes the object code generator and specifies error cases. The intermediate code generator has been omitted because the code to be analyzed is already in this form.

4.1.2.1.1 Scanner

The following is a comprehensive documentation on the design and structure of the scanner of the quadruples compiler. It details fundamental aspects such as tokens definition, lexical grammar, representation by a deterministic finite automaton, corresponding semantic actions, and error handling.

4.1.2.1.1.1 Tokens Definition

Table 9 presents the definition of tokens for the scanner of the quadruples compiler.

Token	Description	Example
<OpenParenthesis, ->	(	(HALT, , ,)
<CloseParenthesis, ->	)	(HALT, , ,)
<OpenBracket, ->	[	(ETIQ, {et, "end"}, ,)
<CloseBracket, ->	]	(ETIQ, {et, "end"}, ,)
<Comma, ->	,	(GoTo,-,-,{Et , "Main"})
<Dash, ->	-	(INPUT_ENT,-,, {VAR_GLOBAL,-22})
<Plus, ->	+	(PARAM, {VAR_TEMP, +6}, ,)
<String, lexeme>	String. The string itself is stored in the attribute.	(GoTo,-,-,{Et, "Main"})
<Integer, value>	Decimal digits. The integer value is stored in the attribute.	(PARAM, {VAR_LOCAL, 0}, ,)
<CR, ->	Carriage return	(RETURN, , ,) \n (HALT, , ,)
<EOF, ->	End of file	(HALT, , ,) eof
<Operator, n>	Token utilized for the reserved words that have been defined by the DRACO practice group for the operators of the quadruples. These tokens constitute a table of operators, with the attribute *n* serving as an index.	(PARAM, {VAR_TEMP, 6}, ,)
<Argument, m>	Token utilized to represent the reserved words defined by the practice group in DRACO for quadruples argument types. These tokens constitute a table of argument classes, wherein the *m* attribute serves as an index.	(PARAM, {VAR_TEMP, 6}, ,)

Table 9 Defined quadruples compiler tokens

In the tokens <Operator , n> <Argument , m> each reserved word defined by the group of practices in DRACO (operators and argument classes) follows the pattern letter followed by any number of letters, digits, or underscores. The pattern is defined as `l ( l | d | _ )*`.

It is imperative that these words not only comply with this pattern, but also that the DB is consulted to verify that they match any definitions made by the group.

In the event of a match, the token is generated. In the absence of such a definition, the token is deemed a lexical error.

4.1.2.1.1.2 Lexical Grammar

Figure 16 provides the regular grammar of the scanner of the quadruples compiler.

```
S → del S | - | ( | ) | { | } | , | eof | + | l A | " C | / B |
    d E | \r F | \n G
A → l A | d A | _ A | λ
C → c1 C | "
B → / D
D → c2 D | \n S | \r S | eof S
E → d E | λ
F → \n | λ
G → \r | λ
```

Figure 16 Regular grammar of the scanner of the quadruples compiler

One of the key design decisions in the lexical grammar is the treatment of line breaks. To ensure compatibility across different operating systems, a single CR token is defined to represent line endings. This approach abstracts away the differences between encodings (\n, \r\n, \r), allowing the analyzer to handle input files uniformly regardless of their origin.

4.1.2.1.1.3 Deterministic Finite Automaton (DFA)

Figure 17 shows the deterministic finite automaton of the scanner of the quadruples compiler obtained from the grammar.

The designed deterministic finite automaton consists of 21 states, 13 of which are final (indicated by a double circle). State 0 is the initial state, and from there, input symbols from the source file are processed to identify the tokens defined in the lexical grammar.

It is worth noting that the handling of comments is implemented through transitions from state 0 to state 14 and then to state 15, triggered by the recognition of the two forward slashes '//'. In state 15, all symbols are read until a newline character or end-of-file is found, causing a transition back to state 0 to resume lexical analysis.

The automaton also explicitly supports the different newline encodings used by major operating systems: '\n' (Unix/Linux), '\r' (classic Mac OS), and '\r\n' (Windows). Specifically, it includes a transition from state 0 to state 18 upon detecting a '\r' character. From state 18, if a '\n' is read, the transition goes to state 20; otherwise, it proceeds to state 19. Both states are final and allow for correct recognition of both '\r' and '\r\n' sequences. Additionally, a direct transition from state 0 to state 6 is defined upon reading a '\n', from which the automaton transitions to states 19 or 20, also marked as final.

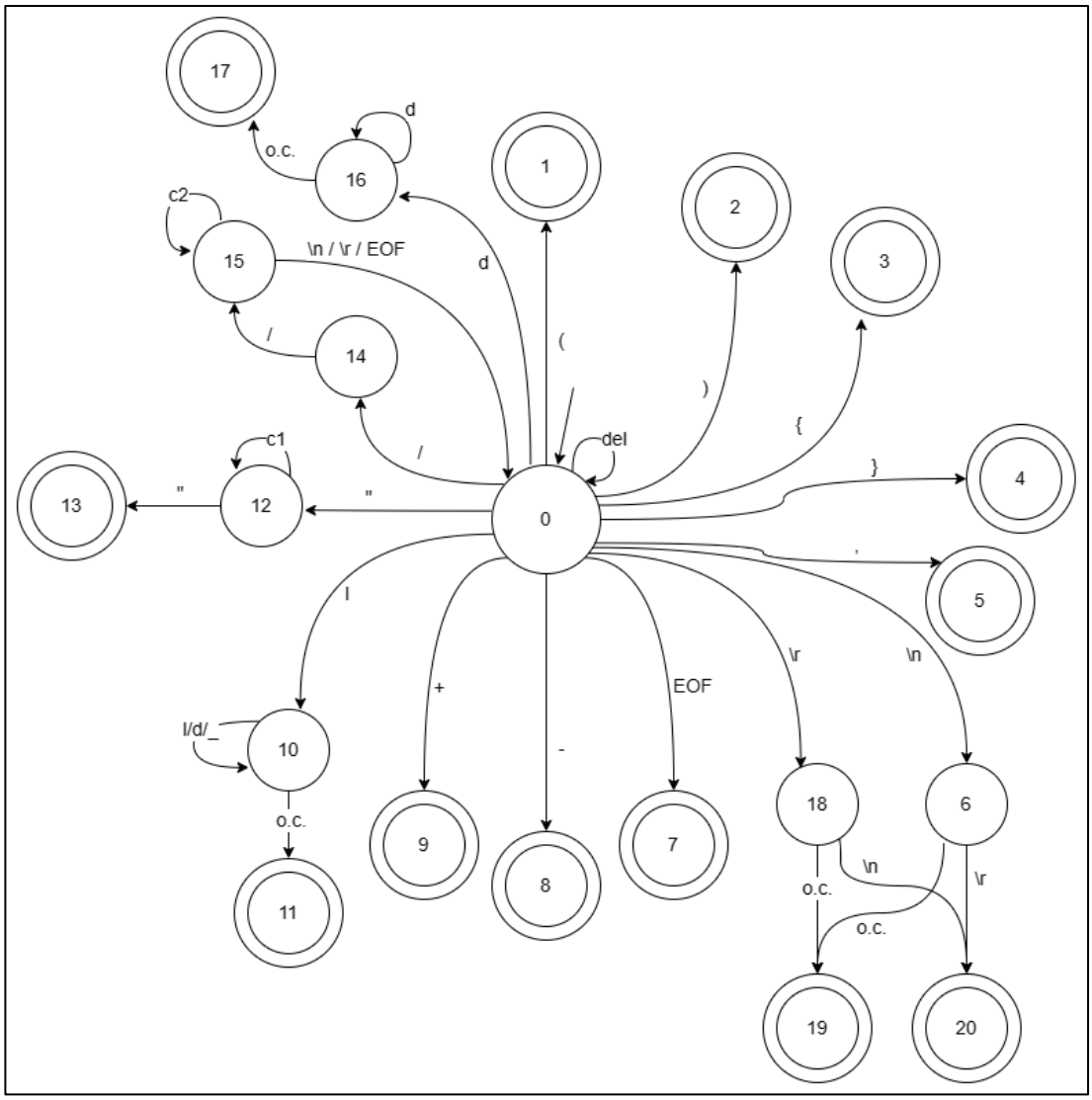

Figure 17 Deterministic finite automaton of the scanner of the quadruples compiler

4.1.2.1.1.4 Semantic Actions

Table 10 lists the semantic actions associated with each transition between states of the scanner of the quadruples compiler.

Transition	Semantic Action
0-0	`Ac1: Read`
0-1	`Ac2: Read; Gen_Token(OpenParenthesis, -)`
0-2	`Ac3: Read; Gen_Token(CloseParenthesis, -)`
0-3	`Ac4: Read; Gen_Token(OpenBracket, -)`
0-4	`Ac5: Read; Gen_Token(ClosedBracket, -)`
0-5	`Ac6: Read; Gen_Token(Comma, -)`
0-7	`Ac8: Gen_Token(EOF, -)`
0-8	`Ac9: Read; Gen_Token(Dash, -)`
0-9	`Ac10: Read; Gen_Token(Plus, -)`
0-10	`Ac11: Lexeme = l; Read`

Transition	Semantic Action
10-10	Ac12: Lexeme = Lexeme + l/d/_; Read

10-11	Ac13:
	index= searchOperator(Lexeme)
	If (index != null)
	    Gen_Token(Operator, index)
	Else{
	    index= searchArgumentType(Lexeme)
	    If (index!=null)
	        Gen_Token(Argument, index)
	    Else
	        Error(20, lexeme)
	}

0-12	Ac14: Lexeme = empty; Read;charCounter=0
12-12	Ac15: Lexeme = Lexeme + c1; Read; charCounter++

12-13	Ac16:
	Read;
	If (charCounter >= stringMaximumLength)
	    Error(26)
	Else
	    Gen_token(String, Lexeme)

0-14	Ac1: Read
14-15	Ac1: Read
15-15	Ac1: Read
15-0	No action
0-16	Ac17: Value= d; Read
16-16	Ac18: Value = Value * 10 + d; Read

16-17	Ac19:
	If (Value > 32767)
	    Error(21)
	Else
	    Gen_Token(Integer, Value)

0-6	Ac1: Read
0-18	Ac1: Read
6-19	Ac20: Line++; Gen_Token(RC, -)
6-20	Ac7: Read; Line++; Gen_Token(RC, -)
18-20	Ac7: Read; Line++; Gen_Token(RC, -)
18-19	Ac20: Line++; Gen_Token(RC, -)

Table 10 Semantic actions of the scanner of the quadruples compiler

The following functions and variables are invoked during the execution of semantic actions defined in the grammar:

- **lexeme**: Stores the lexeme currently being processed by concatenating characters as they are read.
- **index**: Holds the numeric identifier associated with an operator or an argument type. A null value indicates that no matching element was found.

- **charCounter**: Counts the number of characters read when processing a string.
- **stringMaximumLength**: Represents the maximum string length allowed by the compiler.
- **value**: Stores the numeric value obtained from reading digits.
- **searchOperator(lexeme), searchArgumentType(lexeme):** search the lexeme entered as parameter in the reserved word tables defined by the practice group. These functions return the position of that word in the table or null otherwise.

4.1.2.1.1.5 Error Cases

Table 11 indicates the lexical errors of the quadruples compiler and the message that will be provided to the user.

Error Code	Message
20	The reserved word used does not exist in the group definition. Word read: {word}
21	The number exceeds the allowed range of integers.
22	{c} character was not expected.
23	A string cannot contain a line break.
24	A string cannot contain the end of file.
25	To open a comment, it is necessary to use //. The character: {c} was not expected.
26	The string exceeds the maximum allowed length.

Table 11 Quadruples compiler lexical errors

4.1.2.1.2 Parser

In this section, the design of the parser of the quadruples compiler is presented. The syntax grammar, in addition to the error cases, is subsequently delineated.

4.1.2.1.2.1 Syntax Grammar

Figure 19 presents the syntax grammar of the parser of the quadruples compiler. Figure 18 illustrates the correspondence between the symbols employed in the grammar and the tokens that have been defined.

```
( ≡ <OpenParenthesis , - >      ) ≡ <CloseParenthesis , - >
{ ≡ <OpenBracket , - >          } ≡ <CloseBracket , - >
, ≡ <Comma , - >                - ≡ <Dash, ->
+ ≡ <Plus, ->                   cad ≡ <String , lexeme>
ent ≡ <Integer , value>  rc ≡ <RC , - >
eof ≡ <EOF, - >                 opr ≡ <Operator, n >
arg ≡ <Argument, m >
```

Figure 18 Equivalence between tokens and grammatical terminal symbols

```
P → S P | eof
S → cr |( opr , A , A , A) cr
A → { arg , VAL } | - | λ
VAL → cad | NUM NUM'
NUM → ent | - ent | + ent
NUM'→ λ | , NUM
```

Figure 19 Syntax grammar of the parser of the quadruples compiler

LL(1) Grammar Verification:

P → Empty intersection:

First(SP) = {cr, (}

First(eof) = {eof}

S → Empty intersection:

First(cr) = {cr}

First((opr,A,A,A)cr) = {(}

A → Empty intersection:

First({arg,VAL}) = {{}

First(-) = {-}

First (λ) = {λ}

Follow(A) = {, , ,)}

VAL → Empty intersection:

First(cad) = {cad}

First(NUM NUM') = {ent, -, +}

NUM → Empty intersection:

First(ent) = {ent}

First(-ent) = {-}

First(+ent) = {+}

NUM' → Empty intersection:

First (λ) = {λ}

Follow(NUM') = {)}

First(,NUM) = {,}

The syntax grammar meets LL(1) conditions.

4.1.2.1.2.2 Error Cases

Table 12 below delineates the error cases that the parser of the quadruples compiler is capable of detecting.

Error Code	Message
100	The {c} element was not expected. The {c} element was expected.

Table 12 Quadruples compiler syntax errors

4.1.2.1.3 Semantic Analyzer

The upcoming section outlines the design of the semantic analyzer for the quadruples compiler. It details the semantic verifications performed, the syntax grammar annotated with these verifications, and the error cases that can be identified.

4.1.2.1.3.1 Semantic Verifications

Table 13 and Table 14 delineate the semantic verifications executed by the semantic analyzer of the quadruples compiler.

Operator	Semantic Verification
(PRINT_CAD, , , result)	- Must only have argument 3 - Argument 3 cannot be cte_ent or label
(PRINT_CAR, , , result)	- Must only have argument 3. - Argument 3 cannot be cte_ent or label - If argument 3 is cte_cad it must be length 2 (character + null character)
(PRINT_ENT, , , result)	- Must only have argument 3 - Argument 3 cannot be a cte_cad or a label
(INPUT_CAD, , , result) (INPUT_ENT, , , result) (INPUT_CAR, , , result)	- Must only have argument 3 - Argument 3 cannot be cte_ent, cte_cad or a label
(SUMA, arg1, arg2, result) (RESTA, arg1, arg2,result) (MUL, arg1, arg2,result) (DIV, arg1, arg2,result) (MOD, arg1, arg2,result) (EXP, arg1, arg2,result) (AND, arg1, arg2, result) (OR, arg1, arg2, result)	- Must have 3 arguments - None of the three arguments can be label or cte_cad - Argument 3 cannot be cte_ent
(CONCAT, arg1, arg2,result)	- Must have 3 arguments - No argument can be a cte_ent or label - Argument 3 cannot be a cte_cad
(MENOS, arg1, ,result) (NOT, arg1, , result) (ASIG, arg1, , result) (PTR_ASIG, arg1, result)	- Must only have arguments 1 and 3 - No argument can be cte_cad or label - Argument 3 cannot be cte_ent
(ASIG_DIR, arg1, , result) (ASIG_PTR, arg1, , result)	- Must only have arguments 1 and 3 - No argument can be cte_cad, cte_ent or label
(ASIG_CAD, arg1, , result) (PTR_ASIG_CAD, arg1, , result)	- Must only have arguments 1 and 3 - No argument can be a cte_ent or label - Argument 3 cannot be a cte_cad
(ASIG_PTR_CAD, arg1, , result)	- Must only have arguments 1 and 3 - No argument can be cte_ent, cte_cad or label
(GOTO, , , label)	- Must only have argument 3 - Argument 3 must be a label - The label must be defined
(GOTO_IG, arg1, arg2, label) (GOTO_DIST, arg1, arg2, label) (GOTO_MAY, arg1, arg2, label) (GOTO_MEN, arg1, arg2, label) (GOTO_MAY_IG, arg1, arg2, label) (GOTO_MEN_IG, arg1, arg2, label)	- Must have 3 arguments - Arguments 1 and 2 cannot be cte_cad or label - Argument 3 must be a label - The label must be defined

Operator	Semantic Verification
(PARAM, op1, ,) (PARAM_REF, op1, ,) (PARAM_CAD, op1, ,)	- Must only have argument 1 - Argument 1 cannot be cte_cad, cte_ent or label
(CALL, label, ,)	- Must have only argument 1 - Argument 1 must be a label - The label must be defined
(CALL_FUN, label, , result) (CALL_FUN_CAD, label, , result)	- Must only have arguments 1 and 3 - Argument 1 must be a label - Argument 3 cannot be cte_ent, cte_cad or labe - The label must be defined
(RETURN, , ,)	- Must have no arguments.
(HALT, , ,)	- Must not have any arguments - There must be at least one HALT in the program
(RETURN_ENT, , , result)	- Must only have argument 3 - Argument 3 cannot be cte_cad or label
(RETURN_CAD, , , result)	- Must only have argument 3 - Argument 3 cannot be cte_ent or label
(ETIQ, label, ,)	- Must only have argument 1 - Argument 1 must be a label - The same label cannot be defined twice

Table 13 Semantic verifications of operators for the semantic analyzer of the quadruples compiler

Argument	Semantic Verification
{VAR_GLOBAL, offset} {VAR_LOCAL, offset} {VAR_TEMP, offset} {PAR, offset} {PAR_REF, offset} {CTE_ENT, integer}	- Arguments must be a numeric value
{CTE_CAD, string}	- The argument must be a string value
{ET, label}	- The argument must be a string value - The label must be declared
{VAR_NOLOCAL, depth, offset}	- Non-local variables are not supported by the system

Table 14 Semantic verifications of arguments for the semantic analyzer of the quadruples compiler

4.1.2.1.3.2 Translation Schema

Figure 20 presents the syntax grammar annotated in blue with the semantic verifications of the semantic analyzer of the quadruples compiler.

```
P → S P

P → eof
{
   notDeclaredList = difference(labelUsedList, labelDeclaredList)
   foreach(notDeclaredList : labelNotDeclared){
      error(227)
```

```
   }
   if(!inList("main"), labelDeclaredList){
      error(229)
   }
   if(!haltQuadruple){
      error(226)
   }
}
```

S → cr

S →(opr , A_1 , A_2 , A_3) cr

```
{
S.type = ERROR
if(A1.type != EMPTY || A2.type != EMPTY || A3.type != EMPTY){
   switch (opr.type){
      case PRINT_CAD:
         if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY){
            error(205)
         }
         else if(A3.type == CTE_ENT){
            error(206)
         }
         else if(A3.type == ET){
            error(207)
         }
         else{
            S.type = OK
         }
         break

      case PRINT_CAR:
         if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY){
            error(205)
         }
         else if(A3.type == CTE_ENT){
            error(206)
         }
         else if(A3.type == ET){
            error(207)
         }
         else if(strlen(A3.value) != 1){
            error(225)
         }
         else{
            S.type = OK
         }
         break

      case PRINT_ENT:
         if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY){
            error(205)
         }
         else if(A3.type == CTE_CAD){
            error(208)
         }
         else if(A3.type == ET){
            error(207)
         }
         else{
            S.type = OK
```

```
      }
      break
   case INPUT_ENT:
   case INPUT_CAD:
   case INPUT_CAR:
      if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(205)
      }
      else if(A3.type == CTE_CAD){
         error(208)
      }
      else if(A3.type == CTE_ENT){
         error(206)
      }
      else if(A3.type == ET){
         error(207)
      }
      else{
         S.type = OK
      }
      break
   case SUMA:
   case RESTA:
   case MUL:
   case DIV:
   case MOD:
   case EXP:
   case AND_OP:
   case OR_OP:
      if(A1.type == EMPTY || A2.type == EMPTY || A3.type == EMPTY){
         error(209)
      }
      else if(A1.type==CTE_CAD||A2.type==CTE_CAD||A3.type==CTE_CAD){
         error(210)
      }
      else if(A1.type == ET || A2.type == ET || A3.type == ET){
         error(211)
      }
      else if(A3.type == CTE_ENT){
         error(206)
      }
      else{
         S.type = OK
      }
      break
   case CONCAT:
      if(A1.type == EMPTY || A2.type == EMPTY || A3.type == EMPTY){
         error(209)
      }
      else if(A1.type==CTE_ENT|| A2.type==CTE_ENT|| A3.type==CTE_ENT){
         error(212)
      }
      else if(A1.type == ET || A2.type == ET || A3.type == ET){
         error(211)
      }
      else if(A3.type == CTE_CAD){
         error(208)
      }
```

```
      else{
         S.type = OK
      }
      break

   case MENOS:
   case NOT:
   case ASIG:
   case PTR_ASIG:
      if(A1.type == EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(213)
      }
      else if(A1.type == CTE_CAD || A3.type == CTE_CAD){
         error(210)
      }
      else if(A1.type == ET || A3.type == ET){
         error(211)
      }
      else if(A3.type == CTE_ENT){
         error(206)
      }
      else{
         S.type = OK
      }
      break

   case ASIG_DIR:
   case ASIG_PTR:
      if(A1.type == EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(213)
      }
      else if(A1.type == CTE_CAD || A3.type == CTE_CAD){
         error(210)
      }
      else if(A1.type == CTE_ENT || A3.type == CTE_ENT){
         error(212)
      }
      else if(A1.type == ET || A3.type == ET){
         error(211)
      }
      else{
         S.type = OK
      }
      break

   case ASIG_CAD:
   case PTR_ASIG_CAD:
      if(A1.type == EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(213)
      }
      else if(A1.type == CTE_ENT || A3.type == CTE_ENT){
         error(212)
      }
      else if(A1.type == ET || A3.type == ET){
         error(211)
      }
      else if(A3.type == CTE_CAD){
         error(210)
      }
      else{
         S.type = OK
```

```
      }
      break

   case ASIG_PTR_CAD:
      if(A1.type == EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(213)
      }
      else if(A1.type == CTE_ENT || A3.type == CTE_ENT){
         error(212)
      }
      else if(A1.type == CTE_CAD || A3.type == CTE_CAD){
         error(210)
      }
      else if(A1.type == ET || A3.type == ET){
         error(211)
      }
      else{
         S.type = OK
      }
      break

   case GOTO_OP:
      if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY){
         error(205)
      }
      else if(A3.type != ET){
         error(214)
      }
      else{
         if (!in_array(A3.value, labelUsedList))
            labelUsedList[] = A3.value
         S.type = OK
      }
      break

   case GOTO_IG:
   case GOTO_DIST:
   case GOTO_MAY:
   case GOTO_MEN:
   case GOTO_MAY_IG:
   case GOTO_MEN_IG:
      if(A1.type == EMPTY || A2.type == EMPTY || A3.type == EMPTY)
         error(209)
      else if(A1.type == CTE_CAD || A2.type == CTE_CAD)
         error(215)
      else if(A1.type == ET || A2.type == ET)
         error(216)
      else if(A3.type != ET)
         error(214)
      else{
         if (!in_array(A3.value, labelUsedList))
            labelUsedList.add(A3.value)
         S.type = OK
      }
      break

   case PARAM:
   case PARAM_REF:
   case PARAM_CAD:
      if(A1.type == EMPTY || A2.type != EMPTY || A3.type != EMPTY)
         error(217)
```

```
   else if(A1.type == CTE_CAD)
      error(218)
   else if(A1.type == CTE_ENT)
      error(220)
   else if(A1.type == ET)
      error(219)
   else
      S.type = OK
   break

case CALL:
   if(A1.type == EMPTY || A2.type != EMPTY || A3.type != EMPTY)
      error(217)
   else if(A1.type != ET)
      error(221)
   else{
      if (!in_array(A1.value, labelUsedList))
         labelUsedList.add(A1.value)
      S.type = OK
      }
   break

case CALL_FUN:
case CALL_FUN_CAD:
   if(A1.type == EMPTY || A2.type != EMPTY || A3.type == EMPTY)
      error(213)
   else if(A1.type != ET)
      error(221)
   else if(A3.type == ET)
      error(207)
   else if(A3.type == CTE_ENT)
      error(206)
   else if(A3.type == CTE_CAD)
      error(208)
   else{
      if (!in_array(A1.value, labelUsedList))
         labelUsedList.add(A1.value)
      S.type = OK
   }
   break

case RETURN_OP:
case HALT:
   if(A1.type != EMPTY || A2.type != EMPTY || A3.type != EMPTY)
      error(222)
   else{
        haltQuadruple = true
      S.type = OK
   }
   break

case RETURN_ENT:
   if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY)
      error(205)
   else if(A3.type == ET)
      error(207)
   else if(A3.type == CTE_CAD)
      error(208)
   else
      S.type = OK
   break
```

```
      case RETURN_CAD:
         if(A1.type != EMPTY || A2.type != EMPTY || A3.type == EMPTY)
            error(205)
         else if(A3.type == ET)
            error(207)
         else if(A3.type == CTE_ENT)
            error(206)
         else
            S.type = OK
         break

      case ETIQ:
         if(A1.type == EMPTY || A2.type != EMPTY || A3.type != EMPTY)
            error(217)
         else if(A1.type != ET)
            error(221)
         else if(array_search(A1.value, labelDeclaredList) != false)
            error(224)
         else{
            labelDeclaredList.add(A1.value)
            S.type = OK
         }
         break

      default:
         error(223)
   }
}
}
```

A → { arg , VAL }

```
{
A.type = ERROR
A.Value = null
if(arg.type == VAR_NOLOCAL)
   error(228)
else if(val.type == TIPO_DOSNUM)
   error(201)
else if(argumentType == ET || argumentType == CTE_CAD){
   if(val.type == TIPO_CAD){
      A.type = arg.type
      A.Value = VAL.value
   }
   else
      error(202)
}
Else{
   if(val.type == TIPO_NUM){
      A.type = arg.type
      A.Value = VAL.value
   }
   else
      error(203)
}
}
```

A → - {A.type = EMPTY; A.value = null} | λ {A.type = EMPTY; A.value = null}

VAL → cad {VAL.type = TIPO_CAD; VAL.value = cad.string}

VAL → NUM NUM’ {VAL.type = NUM’.type; VAL.value = NUM.value}

```
NUM → ent {NUM.value = ent.value} | - ent {NUM.value = - ent.value} | + ent
{NUM.value = ent.value}
NUM'→ λ {NUM'.type = TIPO_NUM } | , NUM {NUM'.type = TIPO_DOSNUM }
```

Figure 20 Translation schema of the semantic analyzer of the quadruples compiler

4.1.2.1.3.3 Error Cases

The following Table 15 presents a comprehensive overview of the cases of semantic errors identified by the quadruples analyzer, accompanied by the respective messages.

Error Code	Message
200	A non-local variable requires two numeric values
201	The argument must have a value
202	The argument value must be a string
203	The argument value must be a number
204	The quadruple has an error in its arguments
205	The quadruple must have a third argument
206	The third argument cannot be an integer constant
207	The third argument cannot be a label
208	The third argument cannot be a string constant
209	The quadruple must have three arguments
210	No argument can be a string constant
211	No argument can be a label
212	No argument can be an integer constant
213	The quadruple must have first and third argument
214	The third argument must be a label
215	The first and second arguments cannot be a string constant
216	The first and second arguments cannot be a label
217	The quadruple must have first argument
218	The first argument cannot be a string constant
219	The first argument cannot be a label
220	The first argument cannot be an integer constant
221	The first argument must be a label
222	The quadruple must not have any arguments
223	Unrecognized quadruple operator
224	The label has already been defined before. Label: {label}
225	Only one character can be printed
226	At least one execution stop quadruple must be generated
227	The label used has not been declared. Label: {label}

Error Code	Message
228	Non-local variables are not supported by the checker
229	A main label must be defined in the program

Table 15 Quadruples compiler semantic errors

4.1.2.1.4 Assembly Code Generation

The assembly code generation phase constitutes the final step of the quadruples compiler, responsible for translating the sequence of quadruples into executable assembly code. This process involves interpreting each quadruple —defined in the Quadruples Language section— and generating the corresponding low-level instructions in the assembly language described in the section Analysis of the Language.

The translation process is based on a template-driven approach. Each quadruple consists of an operator and up to three arguments, and for each supported operator, a translation template has been defined. These templates describe the corresponding assembly instructions that must be generated, including both the core operation and auxiliary instructions for moving data, managing labels, or handling errors.

Below are some representative examples of how certain quadruples are translated into assembly code:

Quadruples	Translation Template
`(PRINT_CAD, -, -, result)`	`WRSTR result`
`(SUMA, arg1, arg2, result)`	`ADD arg1, arg2` `MOVE .A, result`
`(GOTO_IG, arg1, arg2, label)`	`CMP arg1, arg2` `BZ /label`
`(MENOS, arg1, -, result)`	`MOVE arg1, result` `NEG result`
`(INPUT_CAR, -, -, result)`	`INCHAR result`

Table 16 Example of quadruples translation

There are also some more complex translations, as demonstrated in Figure 21, which are accompanied by a diagram to facilitate understanding.

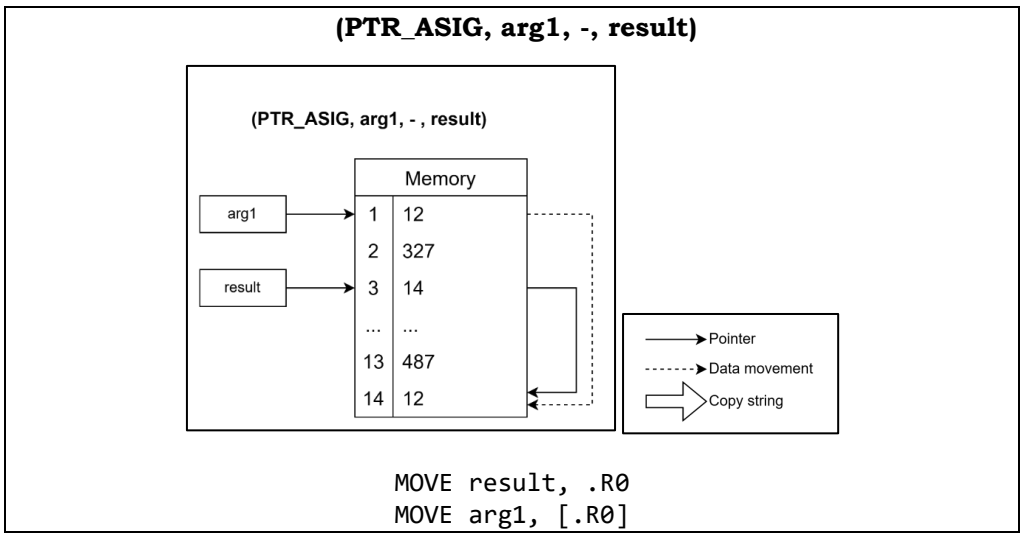

Figure 21 PRT_ASIG quadruple translation

The complete set of translation templates for all supported quadruple operators is provided in chapter Annexes: Translation Template for Quadruples, including detailed handling of both simple and complex cases.

Argument translation is also handled according to predefined rules, converting each argument into its corresponding assembly operand based on its class. Each argument in a quadruple is represented as a tuple `{Class, Value}`. The assembly code generator uses this structure to determine how to access either the value or the address of the argument, depending on the instruction's needs. The translation rules are strictly defined for each argument class, as described below:

Global variables

Stored in a separate memory vector, accessed through the `IY` register.

To retrieve the value:

```
#offset[.IY]
```

The value is stored in `#offset[.IY]`

To retrieve the address:

```
ADD #offset, .IY
MOVE .A, .R5
```

The address is stored in `.R5`

Local variables, temporaries, and value parameters

Stored in the simulated memory vector of the activation record, accessed via `IX` register.

To retrieve the value:

```
#offset[.IX]
```

The value is stored in `#offset[.IX]`

To retrieve the address:

```
ADD #offset, .IX
MOVE .A, .R6
```

The address is stored in `.R6`

Reference parameters

Stored within the activation record memory, accessed using `IX` register:

To retrieve the value:

```
ADD #offset, [.IX]
MOVE .A, [.R7]
```

The value is stored in `.R7`

To retrieve the address:

ADD #offset, .IX

MOVE .A, .R7

The address is stored in `.R7`

String constants

Each string constant requires generating a label. The label is declared with a `DATA` instruction

```
label1234: DATA "value"
```

Label value at: `/label1234`

Integer constants

Translated using immediate addressing mode: `#value`

Labels

Translated directly using the slash-prefixed label notation: `/label`

Due to the fact that only the intermediate code produced by students is available to the compiler, and not the internal structure of activation records, several challenges arise when generating executable assembly code. In particular, function-related operations cannot be fully resolved at compile time because the memory layout and size of activation records are unknown.

To address this limitation, the design introduces a special mechanism that affects both the translation process and the execution phase. Specifically, quadruples involving function calls, parameters, and return statements are translated using custom assembly instructions. These instructions are interpreted directly by the assembly executor, which is capable of simulating activation records and their associated operations.

Activation records are not stored in a stack, but instead are going to be simulated during execution. Each activation record will include:

- A return address (label).
- A program counter (PC).
- A dynamic memory vector that simulates the variable space (parameters, locals, and temporaries).
- A return value.

These attributes are accessed through the `IX` register (or optionally `SP`), which points to the base of the simulated memory. Addressing in the generated code relies on offsets relative to these registers.

The custom instructions defined in section Analysis of the Language are used in the translation of function-related quadruples. Each instruction is associated with specific addressing modes depending on its usage.

To illustrate the use of these custom instructions, Figure 22 shows the translation of a `CALL_FUN` quadruple, where a function is invoked and an integer return value is stored after the call.

```
(CALL_FUN, labelFunction, -, result)
DRACO_GENRETORNO_RA /returnLabel
DRACO_GUARDARIX_RA
DRACO_MOVEIX_RA
BR /myFunction
/returnLabel:
DRACO_GUARDARVALORDEVUELTO_RA result
DRACO_RESTAURARIX_RA
DRACO_ELIMINAR_RA
```

Figure 22 CALL_FUN quadruple translation

4.1.2.2 Solution Design

This section describes the solution from the perspective of the software design of the quadruples compiler.

The compiler has been structured according to a modular, object-oriented approach. The proposed architecture segments the system into clearly differentiated components, each encapsulating the functionality corresponding to one of the main phases of compilation: lexical analysis, syntactic-semantic analysis, and assembly code generation. This organization fosters maintainability, component reuse, and clarity in inter-module interaction.

4.1.2.2.1 System Overview

The UML class diagram, Figure 23, provides a global overview of the system's software structure and the relationships among its main modules:

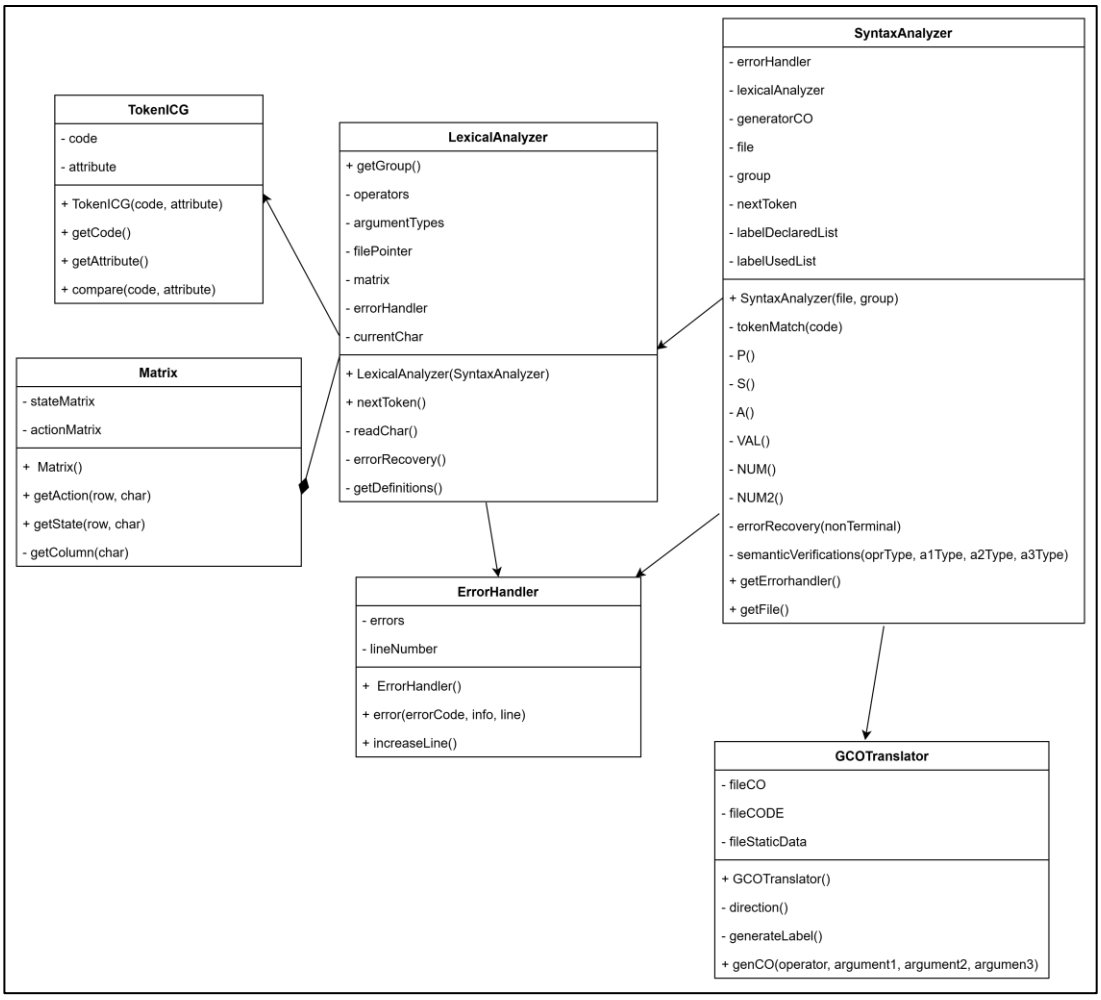

Figure 23 Quadruples compiler UML diagram

4.1.2.2.2 Class-Level Software Design

The following section describes each of the main - classes implemented in the compiler, specifying their primary role within the system and summarizing the most relevant methods provided by each.

- **`TokenICG`**

This class represents tokens generated during lexical analysis as objects with internal attributes identifying the token's code and attribute. It provides

accessor methods, functions `getCode()` and `getAttribute()`, and the comparison method `compare()` to support token equivalence checks.

- **`Matrix`**

The `Matrix` class implements the state transition and action table of the lexical automaton. Based on the current state and input symbol, it determines the next state, function `getState()`, and the action to execute using the function `getAction()`. This auxiliary class is used exclusively by the lexical analyzer.

- **`LexicalAnalyzer`**

The `LexicalAnalyzer` object performs lexical analysis of the quadruples source code. Its operation is based on character-by-character reading of the input file via the `readChar()` method, querying the `Matrix` class to determine the behavior of the automaton at each step. Once a complete lexeme is recognized, it generates an instance of `TokenICG` and returns it to the syntactic analyzer, all within the `nextToken()` method.

Additionally, the class includes the `getDefinitions()` method, which retrieves the custom token names defined by students, ensuring semantic consistency during analysis. In the event of lexical errors, `LexicalAnalyzer` notifies the `ErrorHandler` class and applies an error recovery mechanism via the `errorRecovery()` function, enabling the analysis to resume from a stable state.

- **`SyntaxAnalyzer`**

This class is responsible for syntactic and semantic analysis of the source code. It implements a recursive descent parser with one method per non-terminal symbol: `A()`, `P()`, `S()`, `NUM()`, `NUM2()`, `VAL()`. It requests tokens from `LexicalAnalyzer`, analyzes them based on the defined grammar, and performs semantic checks using the `semanticVerifications()` method.

If the current quadruple is valid, it invokes the `genCO()` method of the `GCOTranslator` object to translate it into assembly code. If syntactic or semantic errors occur, `SyntaxAnalyzer` uses `ErrorHandler` to report them and executes a recovery routine using the function `errorRecovery()` to continue analysis without terminating the entire process.

- **`GCOTranslator.php`**

The `GCOTranslator` object implements the synthesis phase of the compiler. It is responsible for translating validated quadruples into assembly code. It receives a quadruple and applies a translation template through the method `genCO()`, defined and documented in the annex of this document (Translation Template for Quadruples). This template establishes a precise mapping between the quadruple's operator and operands and the corresponding instructions in the target assembly language used in DRACO.

- **`ErrorHandler.php`**

The `ErrorHandler` class provides a centralized interface for reporting lexical, syntax, and semantic errors detected during analysis. It abstracts error handling logic from the analyzers and delivers clear and structured messages to the user via the `error()` method.

4.1.2.2.3 Component Interaction

The compiler's execution flow begins with the `SyntaxAnalyzer` object, which serves as the main orchestrator. It requests tokens from the `LexicalAnalyzer` object, which in turn relies on the `Matrix` class to govern the automaton's

behavior. Once a valid token is identified, `LexicalAnalyzer` encapsulates it in a `TokenICG` instance and returns it to `SyntaxAnalyzer`.

With the received tokens, `SyntaxAnalyzer` validates both syntactic and semantic correctness. If the quadruple is valid, it is forwarded to the `GCOTranslator` object, which generates the corresponding assembly code using the predefined translation template.

Whenever lexical, syntactic, or semantic errors are detected, control is delegated to the `ErrorHandler` component, which is responsible for reporting the detected error.

Both `LexicalAnalyzer` and `SyntaxAnalyzer` implement error recovery mechanisms that allow the analysis to continue once a stable state is reached, thus avoiding a complete interruption of the compilation process.

The diagram, Figure 24, illustrates the dynamic interaction between the main system objects during the compilation of a quadruple.

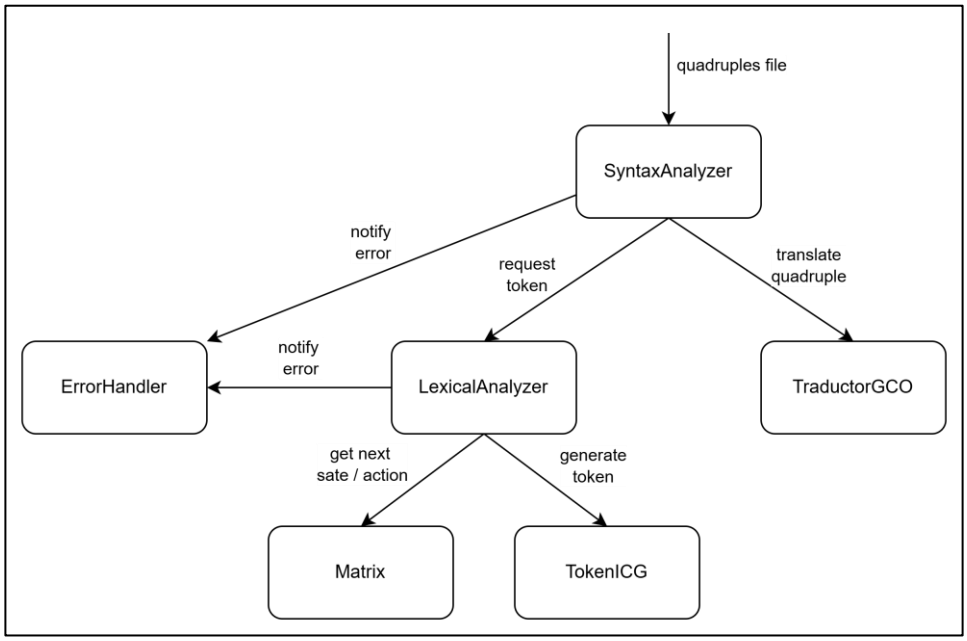

Figure 24 Quadruples compiler component interaction diagram

4.1.2.2.4 Design Considerations

The object-oriented design adopted enables clear encapsulation of the compiler's core concepts. Each class maintains its own attributes and methods, adhering to the principles of single responsibility and low coupling. The relationships between components are clearly defined, and their communication is handled exclusively through objects, ensuring system integrity and modularity.

Moreover, this design is easily extensible and facilitates integration within the DRACO system, allowing each compiler component to evolve independently without compromising the overall functionality.

4.1.3 Assembly Code Analyzer

This section delineates the developmental process of the second component of the project: the assembly code analyzer. It encompasses the design from both perspectives: the compiler design and the solution design.

The objective of this component is to examine and present the identified faults in the assembly code under analysis, with such code originating either directly from a DRACO system user or from the assembly code generated by the quadruples compiler.

4.1.3.1 Analysis of the Language

To date, ENS2001 has been utilized in the practical work of the *Traductores de Lenguajes* course. This application integrates an Assembler that is a subset of the IEEE 694 standard, as well as a Simulator capable of executing programs assembled for that particular implementation of the standard.

This section undertakes an analysis of the assembly language that will be employed in the aforementioned practices and that will be analyzed by the Assembly Code Analyzer of this project. The language under analysis is based on the language used by ENS2001, with minor modifications. [1]

The fundamental aspects of the language relevant to the design of the parser for this language are presented below:

- The word width is 16 bits and the memory is 64K. The address range is from address 0 to 65535.
- The register bank contains the following registers: `PC, SP, SR, IX, IY, A, R0... R9`.
- The source code is comprised of a series of lines that adhere to the following format: `[[Label ':'] Instruction] [';' Comment]`
- After the END instruction no more code will be analyzed. The `END` instruction cannot be preceded by a label.
- Empty lines and lines containing exclusively comments are permissible. It is imperative that each instruction is entered on a separate line.
- Labels must commence with a letter, followed by one or more letters, digits, or underscores. It is imperative to note that no other characters may be used in the designation of a label. It should also be noted that the designation of a label is case-sensitive.
  In instances where code has been generated by the DRACO system, will be permissible to employ labels that commence with an underscore.
- In contrast to the constraints imposed by ENS 2001, the utilization of mixed upper- and lower-case letters in the writing of instructions and registers is permitted.
- The format of the instructions is as follows: `mnemonic [Operand1 [, Operand2]]`. A mnemonic is defined as a word or abbreviation that refers to the operation.
  Operands can be one of the designated addressing modes. However, not all instructions support all possible addressing modes.
- The available addressing modes and their format are shown below:
  - Immediate addressing → `#Integer`
  - Direct addressing to register → `.Register`
  - Direct memory addressing → `/Integer` or `/Label`
  - Indirect addressing → `[.Register]`
  - Relative to index register addressing → `#Integer[.Register]`
  - Program counter addressing → `$Integer` or `$Label`

- The language facilitates the writing of numbers in decimal and hexadecimal adhering to the format: 0xHexadecimal number.
- The language presents a subset of macroinstructions. Expressions are defined as sequences of numbers operating within the conventional operator precedence rules, encompassing the standard arithmetic operators: addition, subtraction, negative, multiplication, division, modulus operator, and parentheses.
  The range of integer values allowed in expressions is [-32768, 65535], which may be interpreted either as two's complement integers or as unsigned absolute values in the range depending on the context in which they are used.
- In addition to the instructions delineated in the ENS 2001 user manual, the language employed incorporates the following new custom instructions utilized by the DRACO system:

  - **DRACO_CREA_RA**
    Creates a new activation record if it has not been created yet. This instruction does not take arguments.

  - **DRACO_MOVE_RA argument**
    Stores the given argument into the next available memory slot of the activation record.
    Addressing modes supported: immediate, register, memory, indirect, relative.

  - **DRACO_GENRETORNO_RA argument**
    Stores the return label in the activation record's return address field.
    Addressing modes supported: memory, program-counter-relative, indirect.

  - **DRACO_GUARDARIX_RA**
    Saves the current IX value into the activation record's control pointer (PC) field.

  - **DRACO_MOVEIX_RA**
    Sets the IX register to point to the current activation record.

  - **DRACO_RESTAURARIX_RA**
    Restores IX register from the PC field of the activation record.

  - **DRACO_ELIMINAR_RA**
    Deletes the current activation record after the function finishes execution.

  - **DRACO_GUARDARVALORDEVUELTO_RA argument**
    Stores the function's return value into the destination specified by the argument.
    Addressing modes supported: register, memory, indirect, relative.

  - **DRACO_GUARDARVALORDEVUELTOCAD_RA argument**
    Stores the function's return value (string) into the destination specified by the argument.
    Addressing modes supported: register, memory, indirect, relative.

- **DRACO_BR_DIRRET_RA**
  Performs a jump to the return address stored in the activation record.

- **DRACO_MOVEVALORDEVUELTO_RA argument**
  Copies the given integer argument into the activation record's return field.
  Addressing modes supported: immediate, register, memory, indirect, relative.

- **DRACO_MOVEVALORDEVUELTOCADENA_RA argument**
  Copies the given string argument into the activation record's return field.
  Addressing modes supported: register, memory, indirect, relative.

- The subsequent tables offer clarification on the interpretation of numbers in different contexts. Table 17 indicates the interpretation of integers according to the addressing mode, while Table 18 provides the interpretation according to the macro instructions.

  By default, numeric values are interpreted literally as written. That is, positive values are treated as unsigned (e.g., 5 → 5), and negative values are treated as signed (e.g., -1 → -1) without applying binary reinterpretation.

  However, certain addressing modes or instruction types may override this behavior, applying specific interpretations such as two's complement decoding or treating negative values as pre-encoded unsigned representations. These cases are detailed in the following tables.

  As an internal extension of the original ENS 2001 language, the DRACO system allows the use of values in the extended range [-32768, 65535] for specific addressing modes: index register and PC addressing.

  This extension is applied only at the system level and is not available to students, who must adhere to the original language specification.

Immediate addressing	Direct memory addressing	Index register addressing	Program counter addressing
Allowed integer range: [-32768, 65535]		Allowed integer range for the system: [-32768, 65535] Allowed integer range for students: [-128, 255]	
Values >32767 are interpreted as signed integers using two's complement representation	Values <0 are treated it as an unsigned 16-bit value	Values >127 (for students) or >32767 (for the system) as signed integers using two's complement representation	
Example: `MOVE #40000, .R1` `.R1` → -25536	Example: `MOVE #33, /-1` Address 65535 → 33	Example: `BR $129` Is interpreted as `BR $-127`	

Table 17 Interpretation of integers in addressing modes of assembly code language

RES	EQU	ORG	DATA
Allowed integer range: [-32768, 65535]			
Values <0 are treated it as an unsigned 16-bit value			Values >32767 are interpreted as signed integers using two's complement representation
Example: `ORG -1000` Assembles code at address 64536			Example: `Label1: DATA 40000` `Label1` value → -25536

Table 18 Interpretation of integers in macroinstructions of assembly code language

4.1.3.2 Compiler Design

This chapter documents the design of the language compiler specified in the previous section. Its primary objective is to verify the correctness of the assembly code, through the analysis phases of a compiler: lexical, syntax, and semantic analysis. The system also defines and handles potential error cases to ensure the input conforms to the expected structure.

The generation of machine code (synthesis phase) is not part of the current work, as the assembler language used already represents the object-level code for execution in DRACO. However, a future extension could incorporate translation to binary-level machine code, placing instructions directly into memory.

4.1.3.2.1 Scanner

A comprehensive documentation of the design of the scanner of the assembly code analyzer compiler is provided below. This documentation includes a detailed description of the tokens definition, the lexical grammar, the deterministic finite automaton, the semantic actions, the error cases, and the transition state matrix.

4.1.3.2.1.1 Tokens Definition

Table 19 presents the tokens defined along with a description and a code example highlighting the symbol that generates the token.

Token	Description	Example
`< CR, - >`	Carriage return	`ADD #0,.IY \n` `MOVE .A, .R5`
`< EOF, - >`	End of file	`MOVE .A, .R5` ***eof***
`< Label, lexeme >`	Name of a label.	`BR /main`
`< String, lexeme >`	String	`Label1: DATA "Hello"`
`< Colon, - >`	:	`Label1: DATA "Hello"`
`< Comma, - >`	,	`MOVE .A, .R5`
`< OpenBracket, - >`	[	`CMP [.R1], #0`
`< ClosedBracket, - >`	]	`CMP [.R1], #0`
`< Register, n >`	.PC, .SP, .SR, .IX, .IY, .A, .R0… .R9	`CMP [.R1], #0`
`< Hash, - >`	#	`CMP [.R1], #0`

Token	Description	Example
< Dollar, - >	$	BR **$**3
< Integer, value >	Decimal or hexadecimal digits. An integer value is stored at the attribute.	BR $3
< Plus, - >	+	ORG 4+2
< Dash, - >	-	ORG 4-2
< Multiplication, - >	*	ORG 4*2
< Slash, - >	/	BZ /label1 ORG 4/2
< OpenParen, - >	(	ORG **(**4+2)*7
< ClosedParen, - >	)	ORG (4+2**)**7
< Module, - >	%	ORG (4%2)*7
< ORG, - >	Macroinstruction ORG	**ORG** 100
< EQU, - >	Macroinstruction EQU	Label: **EQU** 100
< END, - >	Macroinstruction END	**END**
< RES, - >	Macroinstruction RES	Label: **RES** 100
< DATA, - >	Macroinstruction DATA	Label: **DATA** 100
< NOP, - >	Instruction NOP	**NOP**
< HALT, - >	Instruction HALT	**HALT**
< MOVE, - >	Instruction MOVE	**MOVE** .A, .R5
< PUSH, - >	Instruction PUSH	**PUSH** #0xFF
< POP, - >	Instruction POP	**POP** #12
< ADD, - >	Instruction ADD	**ADD** #100, .IY
< SUB, - >	Instruction SUB	**SUB** #100, .IY
< MUL, - >	Instruction MUL	**MUL** #100, .IY
< DIV, - >	Instruction DIV	**DIV** #100, .IY
< MOD, - >	Instruction MOD	**MOD** #100, .IY
< INC, - >	Instruction INC	**INC** .R1
< DEC, - >	Instruction DEC	**DEC** .R1
< NEG, - >	Instruction NEG	**NEG** .R4
< CMP, - >	Instruction CMP	**CMP** [.R1], **#0**
< AND, - >	Instruction AND	**AND** .R1, #0
< OR, - >	Instruction OR	**OR** .R1, #0
< XOR, - >	Instruction XOR	**XOR** .R1, #0
< NOT, - >	Instruction NOT	**NOT** .R1
< BR, - >	Instruction BR	**BR** /main
< BZ, - >	Instruction BZ	**BZ** /label1
< BNZ, - >	Instruction BNZ	**BNZ** /label1
< BP, - >	Instruction BP	**BP** /label1

Token	Description	Example
< BN, - >	Instruction BN	**BN** /label1
< BNV, - >	Instruction BNV	**BNV** /label1
< BC, - >	Instruction BC	**BC** /label1
< BNC, - >	Instruction BNC	**BNC** /label1
< BE, - >	Instruction BE	**BE** /label1
< BO, - >	Instruction BO	**BO** /label1
< CALL, - >	Instruction CALL	**CALL** /function1
< RET, - >	Instruction RET	**RET**
< INCHAR, - >	Instruction INCHAR	**INCHAR** .R1
< ININT, - >	Instruction ININT	**ININT** .R1
< INSTR, - >	Instruction INSTR	**INSTR** /1024
< WRCHAR, - >	Instruction WRCHAR	**WRCHAR** .R1
< WRINT, - >	Instruction WRINT	**WRINT** .R1
< WRSTR, - >	Instruction WRSTR	**WRSTR** /1024
< DRACO_CREA_RA, - >	Instruction DRACO_CREA_RA	**DRACO_CREA_RA**
< DRACO_MOVE_RA, - >	Instruction DRACO_MOVE_RA	**DRACO_MOVE_RA** .R2
< DRACO_GENRETORNO_RA, - >	Instruction DRACO_GENRETORNO_RA	**DRACO_GENRETORNO_RA** /1000
< DRACO_GUARDARIX_RA, - >	Instruction DRACO_GUARDARIX_RA	**DRACO_GUARDARIX_RA**
< DRACO_MOVEIX_RA, - >	Instruction DRACO_MOVEIX_RA	**DRACO_MOVEIX_RA**
< DRACO_RESTAURARIX_RA, - >	Instruction DRACO_RESTAURARIX_RA	**DRACO_RESTAURARIX_RA**
< DRACO_ELIMINAR_RA, - >	Instruction DRACO_ELIMINAR_RA	**DRACO_ELIMINAR_RA**
< DRACO_GUARDARVALORDEVUELTO_RA, - >	Instruction DRACO_GUARDARVALORDEVUELTO_RA	**DRACO_GUARDARVALORDEVUELTO_RA** .R9
< DRACO_GUARDARVALORDEVUELTOCAD_RA, - >	Instruction DRACO_GUARDARVALORDEVUELTOCAD_RA	**DRACO_GUARDARVALORDEVUELTOCAD_RA** .R8
< DRACO_BR_DIRRET_RA, - >	Instruction DRACO_BR_DIRRET_RA	**DRACO_BR_DIRRET_RA**

Token	Description	Example
< DRACO_MOVEVALORDEVUELTO _RA, - >	Instruction DRACO_MOVEVALORDE VUELTO_RA	**DRACO_MOVEVALORDEVUELTO _RA** .R3
< DRACO_MOVEVALORDEVUELTO CADENA_RA, - >	Instruction DRACO_MOVEVALORDE VUELTOCADENA_RA	**DRACO_MOVEVALORDEVUELTO CADENA_RA** .R2

Table 19 Defined assembly code analyzer compiler tokens

4.1.3.2.1.2 Lexical Grammar

Figure 25 shows the regular grammar of the scanner of the assembly code analyzer.

S → del S | _ A | l A | $d_{1\text{-}9}$ B | 0 H |" C | . D | eof | : | , | [|] | # | $ | + | - | * | / | (|) | % | \n E | \r F | ; G

A → l A | d A | _ A | λ ;labels, instructions and macroinstructions

B → d B | λ ;numbers

C → c1 C | " ;strings

D → l K ;registers

E → \r | λ ;carriage return

F → \n | λ ;carriage return

G → c2 G | eof S | \n S | \r S ;comments

H → x I | d B | λ ;numbers

I → h J ;hexadecimal number

J → h J | λ ;hexadecimal number

K → l | d | λ ;registers

c1: any character except ", line breaks, and EOF.

c2: any character except line breaks, and EOF.

h: hexadecimal digit.

Figure 25 Regular grammar of the scanner of the assembly code analyzer

4.1.3.2.1.3 Deterministic Finite Automaton (DFA)

Figure 26 shows the deterministic finite automaton of the scanner of the assembly code analyzer.

As can be observed, this automaton follows the same design approach used in the lexical analyzer of the quadruples compiler to handle comments and the different newline encodings. The only difference is that, in this case, comments start with a semicolon '**;**' instead of double slashes '//'.

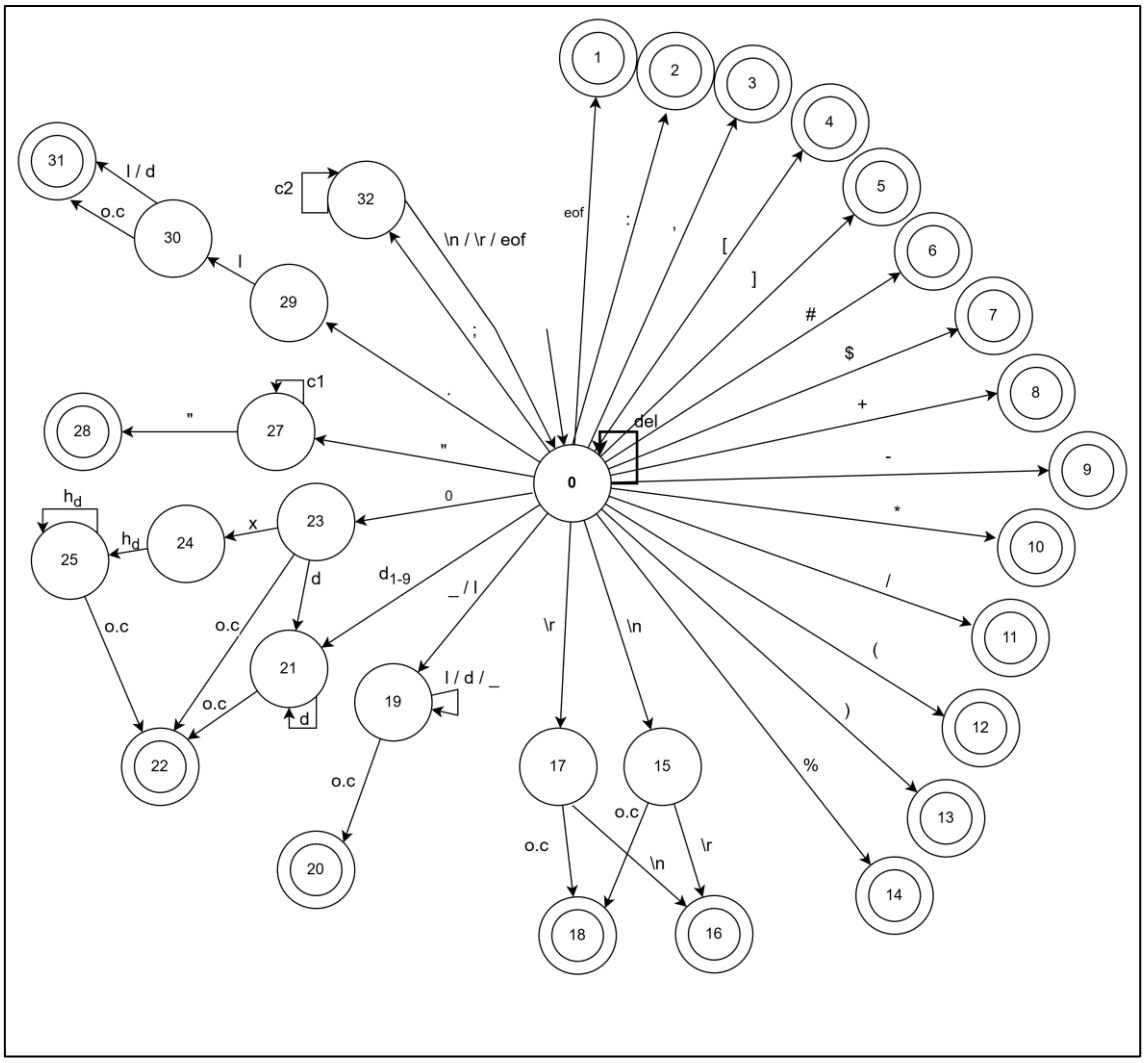

Figure 26 Deterministic finite automaton of the scanner of the assembly code analyzer

4.1.3.2.1.4 Semantic Actions

Table 20 shows the semantic actions defined for the state transitions of the scanner. In addition to the actions specified in the table, in case of any unspecified transition, the error 1001 will be thrown.

Transition	Semantic Action
0-0	`Ac1: Read`
0-1	`Ac2: Gen_Token(EOF, -)`
0-2	`Ac3: Read; Gen_Token(Colon, -)`
0-3	`Ac4: Read; Gen_Token(Comma, -)`
0-4	`Ac5: Read; Gen_Token(OpenBracket, -)`
0-5	`Ac6: Read; Gen_Token(ClosedBracket, -)`
0-6	`Ac7: Read; Gen_Token(Hash, -)`
0-7	`Ac8: Read; Gen_Token(Dollar, -)`
0-8	`Ac9: Read; Gen_Token(Plus, -)`
0-9	`Ac10: Read; Gen_Token(Dash, -)`
0-10	`Ac11: Read; Gen_Token(Multiplication, -)`

Transition	Semantic Action
0-11	Ac12: Read; Gen_Token(Slash, -)
0-12	Ac13: Read; Gen_Token(OpenParen, -)
0-13	Ac14: Read; Gen_Token(ClosedParen, -)
0-14	Ac15: Read; Gen_Token(Module, -)
0-15	Ac1
15-16	Ac16: Read, Line count++; Gen_Token(RC, -)
15-18	Ac17: Line count++; Gen_Token(RC, -)
0-17	Ac1
17-16	Ac16: Read; Line count++; Gen_Token(RC, -)
17-18	Ac17: Line count++; Gen_Token(RC, -)

0-19	Ac18:
	lexeme=_/l
	reservedDraco= false
	If(lexeme == "_"){
	reservedDraco=true
	}
	Read

19-19	Ac19: lexeme=lexeme + l/d/_; Read

19-20	Ac20:
	If(isStudent() AND reservedDraco){
	ERROR(1002)
	}
	Else{
	Index = searchInstruction (lexeme)
	If (index!=null){
	If(isStudent() AND palabraDraco(index))
	ERROR(1009)
	Else
	Gen_Token(lexeme, -) ;Specific token of the macro/instruction
	}
	Else{
	Gen_Token(Label, lexeme)
	}
	}

0-21	Ac21: Value=d; Read
21-21	Ac22: Value=Value*10+d; Read

21-22	Ac23:
	If (Value>65535){
	ERROR(1003)
	}
	Else{
	Gen_Token(Integer, Value)
	}

0-23	Ac24: Value = 0; Read
23-21	Ac22: Value=Value*10+d; Read
23-22	Ac25: Gen_Token(Integer, Value)

Transition	Semantic Action
23-24	Ac1
24-25	Ac26: Value= hexToDec(h_d); Read
25-25	Ac27: Value= Value * 16 + hexToDec(h_d); Read
25-22	Ac23
0-27	Ac28: Read; Lexeme = empty
27-27	Ac29: Lexeme = Lexeme + c1; Read
27-28	Ac30: Read; Gen_Token(String, lexeme)
0-29	Ac1
29-30	Ac31: Lexeme = l; Read

30-31 → l/d	Ac32:
	Lexeme = Lexeme + l/d
	Read
	index = searchRegister(lexeme)
	If (index != null){
	  Gen_Token(Register, index)
	}
	Else{
	  Error (1004)
	}

30-31 → o.c	Ac33:
	index = searchRegister(lexeme)
	If (index != null){
	  Gen_Token(Register, index)
	}
	Else{
	  Error (1004, “El registro <lexeme> no existe”)
	}

0-32	Ac1
32-32	Ac1
32-0	No action

Table 20: Semantic actions of the scanner of the assembly code analyzer

4.1.3.2.1.5 Error Cases

Table 21 details the lexical errors detected by the scanner.

Error Code	Message
1001	The character {c} was not expected.
1002	Wrong instruction or label format, cannot start with character: _ .
1003	The number exceeds the allowed range.
1004	The register {lexeme} does not exist.
1005	Hexadecimal number expected, {c} character was not expected.
1006	The end of file was found inside a string.
1007	Strings cannot contain line breaks.

Error Code	Message
1008	Invalid register name. Existing registers are: PC, SP, SR, IX, IY, A, R0… R9
1009	Invalid instruction or label.

Table 21 Assembly code analyzer lexical errors

4.1.3.2.2 Parser

The following section details the design of the object code analyzer. This section is exclusively concerned with the design of the syntax grammar and the specification of syntax errors.

4.1.3.2.2.1 Syntax Grammar

The following Figure 27 shows the syntax rules of the assembly code analyzer.

```
P → S P | end
S → L A cr
L → label : | λ
A → M | I | λ
M → M1 E | data ED
M1 → org | equ | res
E → J E'
E' → + J E' | - J E' | λ
J → K J'
J' → * K J' | / K J' | % K J' | λ
K → ( E ) | INT
ED → - integer ED1 | integer ED1 | string ED1
ED1 → , ED| λ
I → N | O | T
N → nop | halt | draco_crea_ra | draco_guardarix_ra | draco_moveix_ra |
draco_restaurarix_ra | draco_eliminar_ra | draco_br_dirret_ra | ret
O → O1 D1 | O2 D2 | O3 D3 | O4 D4
T → move D1 D2| T1 D1 D1
O1 → push | wrchar | wrint | draco_move_ra | draco_movevalordevuelto_ra
O2 → pop | inc | dec | neg | not | inchar | inint |
draco_guardarvalordevuelto_ra | draco_guardarvalordevueltocad_ra |
draco_movevalordevueltocadena_ra
O3 → br | bz | bnz | bp | bn | bv | bnv | bc | bnc | be | bo | call |
draco_genretorno_ra
O4 → instr | wrstr
T1 → add | sub | div | mul | mod | cmp | and | or | xor
D1 → register | / LABELINT | [ register ]| # DD1
DD1 → label | INT DD2
DD2 → λ | [ register ]
D2 → register | / LABELINT | [ register ] | # INT [ register ]
D3 → / LABELINT | $ LABELINT | [ register ]
```

```
D4 → / LABELINT | [ register ]| # INT [ register ]
LABELINT → label | INT
INT → - integer | integer
```

Figure 27 Syntax grammar of the parser of the assembly code analyzer

LL(1) Grammar Verification:

P → terminal "end" does not appear in any other rule ensuring that the intersection is empty.

L → terminal "label" does not appear in any other rule ensuring that the intersection is empty.

A → Empty intersection:

First(M) = { org, equ, res, data }

First(I) = { nop, halt, push, pop… terminals of the instructions }

First (λ) = { λ }

Follow(A) = { cr }

M → Empty intersection:

First(M1) = { org, equ, res}

First(data) = { data }

M1 → Empty intersection, each rule begins with a different terminal.

E' → Empty intersection:

First(+ J E') = { + }

First(- J E') = { - }

First (λ) = { λ }

Follow(E') = {), cr }

J' → Empty intersection:.

First(*KJ') = { * }

First(/KJ') = { / }

First(%KJ') = { % }

First (λ) = { λ }

Follow(J') = {), cr }

K → Empty intersection:.

First ((E)) = { (}

First (INT) = { -, integer }

ED → Empty intersection, each rule begins with a different terminal.

ED1 → Empty intersection:.

First(,ED) = { , }

First (λ) = { λ }

Follow(ED1) = {), cr }

I → Empty intersection, all its rules begin with different terminal symbols corresponding to different instructions.

N → No intersection. Different terminals of the instructions.

O → Empty intersection, all its rules begin with different terminal symbols corresponding to different instructions.

T → Empty intersection, all its rules begin with different terminal symbols corresponding to different instructions.

O1 → Empty intersection, each rule begins with a different terminal.

O2 → Empty intersection, each rule begins with a different terminal.

O3 → Empty intersection, each rule begins with a different terminal.

O4 → Empty intersection, each rule begins with a different terminal.

T1 → Empty intersection, each rule begins with a different terminal.

D1 → Empty intersection, each rule begins with a different terminal.

DD1 → Empty intersection, each rule begins with a different terminal.

DD2 → Empty intersection:

First([register]) = { [}

First (λ) = { λ }

Follow(DD2) = { register , / , [, # , cr }

D2 → No intersection, different terminals.

D3 → Empty intersection, each rule begins with a different terminal.

D4 → Empty intersection, each rule begins with a different terminal.

LABELINT → Empty intersection, each rule begins with a different terminal.

INT → Empty intersection, each rule begins with a different terminal.

The syntax grammar meets LL(1) conditions.

4.1.3.2.2.2 Error Cases

Table 22 details the syntax errors detected by the parser.

Error Code	Message
1100	The {c} element was not expected. The {c} character was expected.

Table 22 Assembly code analyzer syntax errors

4.1.3.2.3 Semantic Analyzer

The following section presents the design of the semantic analyzer of the object code analyzer. This section delineates the semantic verifications that are performed, the syntax grammar that is annotated with the semantic verifications, and the error cases that can be detected.

4.1.3.2.3.1 Semantic Verifications

The following semantic verifications are required by the compiler to ensure the proper analysis of the assembly code. A detailed description of each verification is provided in Table 23.

Type	Semantic Verification
Labels	A label that is used must be declared at some point.
	The same label cannot be declared several times.
Memory size	Instructions cannot be written above memory address 65535.
Registers	Relative to index register addressing only allows the following registers: `IX, IY, SP.`
	The `SP` register can only be used by the DRACO system, not by students.
Integers	Integer values <-32768 are not allowed.
	The resulting value of the expressions must be in the range [-32768, 65535].
Integers in addressing modes	For students, only the range [-128, 255] is allowed for relative to index register and program counter addressing.
	Bifurcations to addresses outside the memory limits [0, 65535] are not allowed.
	For students, bifurcations to a remote label are not allowed more than 127 positions forward or 128 positions backward.
RES Macroinstruction	The sum of the address and the amount of memory to be reserved cannot exceed the memory limit of 65535.

Table 23 Semantic verifications of the semantic analyzer of the assembly code analyzer

4.1.3.2.3.2 Translation Schema

Figure 28 presents the syntax grammar annotated in blue with the semantic verifications of the semantic analyzer.

```
P → {address = 0} S P

P → end
{
    foreach(labelUsed){
        if(!label IN labelDeclared)
            error (1209)
    }
    foreach(relativeBifurcationsList : relativeBifurcation){
        labelContent = searchContentLabelDeclared(relativeBifurcation.name)
        if(!isDraco && labelContent!= null){
            dirDifference = relativeBifuraction.address - labelContent
            if(dirDifference < -127 OR dirDifference > 128)
                error (1200)
        }
    }
}

S → L A cr
{
    if(address > 65535)
        error(1201)
    if (L.label != empty && A.type == equ){
        labelDeclared.setContent(L.label, A.value)
    }
}
```

```
L → label :
{
    if(labelDeclared(label.name))
        error(1202)
    else
        labelDeclared.add(label.name; address)
    L.label = label.name
}
```

```
L → λ {L.label = empty}
```

```
A → M {A.type = M.type; A.value = M.value} | I {A.type = emtpy; A.value =
empty } | λ {A.type = emtpy; A.value = empty}
```

```
M → M1 E
{
    if(E.value > 65535 OR E.value <-32768)
        error(1203)
    else{
        if(M1.type==org){
            address=E.value
        }
        Elseif(M1.type==res){
            If(address + E.value > 65535)
                Error(1204)
            Else
                address += E.value
        }
        M.type = M1.type
        M.value = E.value
    }
}
```

```
M → data ED {M.type = data; M.value = empty}
```

```
M1 → org {M1.type=org} | equ {M1.type=equ} | res {M1.type=res}
```

```
E → J {E'.value = J.value} E' {E.value = E'.value}
```

```
E' → + J {E1'.value = E'.value + J.value} E'1 {E'.value = E1'.value}
```

```
E' → - J {E1'.value = E'.value - J.value} E'1 {E'.value = E1'.value}
```

```
E' →  λ {E'.value = empty}
```

```
J → K {J'.value = K.value} J' {J.value = J'.value}
```

```
J' → * K {J1'.value = J'.value * K.value} J'1{J'.value = J1'.value}
```

```
J' → / K {J1'.value = J'.value / K.value} J'1{J'.value = J1'.value}
```

```
J' → % K {J1'.value = J'.value % K.value} J'1{J'.value = J1'.value}
```

```
J' → λ {J'.value = empty}
```

```
K → ( E ) {K.value = E.value}
```

```
K → INT {K.value = INT.value}
```

```
ED → - integer
{
    if(integer.value > 32768)
        error(1205)
    Else
        address ++
} ED1
```

```
ED → integer
{
    if(integer.value > 65535)
        error(1205)
    Else
        address ++
} ED1

ED → string {address += (string.length + 1)} ED1

ED1 → , ED| λ

I → N {address ++} | O | T

N → nop | halt | draco_crea_ra | draco_guardarix_ra | draco_moveix_ra |
draco_restaurarix_ra | draco_eliminar_ra | draco_br_dirret_ra | ret

O → O1 {address ++} D1 {address ++}

O → O2 { address ++} D2 { address ++}

O → O3 { address ++} D3 { address ++}

O → O4 { address ++} D4 { address ++}

T → move { address ++} D1 { address ++} D2
{
    if(D1.flagByte OR D2.flagByte)
        address ++
}

T → T1 { address ++} D1 { address ++} D1
{
    if(D1₁.flagByte OR D1₂.flagByte)
        address ++
}

O1 → push | wrchar | wrint | draco_move_ra |  draco_movevalordevuelto_ra

O2 → pop | inc | dec | neg | not | inchar | inint |
draco_guardarvalordevuelto_ra | draco_guardarvalordevueltocad_ra |
draco_movevalordevueltocadena_ra

O3 → br | bz | bnz | bp | bn | bv | bnv | bc | bnc | be | bo | call |
draco_genretorno_ra

O4 → instr | wrstr

T1 → add | sub | div | mul | mod | cmp | and | or | xor

D1 → register {D1.flagByte = false}

D1 → / LABELINT {D1.flagByte = true}

D1 → [ register ] {D1.flagByte = false}

D1 → # DD1 {D1.flagByte = DD1.flagByte}

DD1 → label {labelUsed.add(label.name, address); DD1.flagByte = true}

DD1 → INT {DD2.value = INT.value} DD2 {DD1.flagByte = DD2.flagByte}

DD2 → λ {DD2.flagByte=true}

DD2 → [ register ]
{
    if (register.type != (.IX OR .IY OR .SP)){
        error (1206)
    }
    if (!isDraco){
        if (register.type == .SP)
```

```
            error (1207)
        if(DD2.value > 255 OR DD2.value < -128))
            error(1208)
    }
    DD2.flagByte = false
}
```

D2 → register {D2.flagByte=false}

D2 → / LABELINT {D2.flagByte=true}

D2 → [register] {DD2.flagByte=false}

D2 → # INT [register] {DD2.flagByte=false}

```
{
    if (register.type != (.IX OR .IY OR .SP)){
        error (1206)
    }
    if (!isDraco){
        if (register.type == .SP)
            error (1206)
        if(INT.value > 255 OR INT.value < -128))
            error(1211)
    }
}
```

D3 → / LABELINT | [register]

D3 → $ LABELINT

```
{
    if(LABELINT.value != empty){
        if(!isDraco &&(LABELINT.value > 255 OR LABELINT.value < -128))
            error(1208)
        finalAddress = address + 1 + LABELINT.value
        if(finalAddress > 65535 OR finalAddress < 0)
            error(1210)
    }
    else
        relativeBifurcationsList.add(LABELINT.labelName, address+1)
}
```

D4 → / LABELINT | [register]

D4 → # INT [register]

```
{
    if (register.type != (empty OR .IX OR .IY OR .SP)){
        error (1206)
    }
    if (!isDraco){
        if (DD1.type == .SP)
            error (1207)
        if(INT.value != empty && (INT.value > 255 OR INT.value < -128))
            error(1211)
    }
}
```

LABELINT → label {labelUsed.add(label.name, address); LABELINT.nameLabel = label.name; LABELINT.value = empty}

LABELINT → INT {LABELINT.value = INT.value; LABELINT.nameLabel = empty}

INT → - integer

```
{
    if(integer.value > 32768)
```

```
        error(1205)
    Else
        INT.value = - integer.value
}
INT → integer
{
    if(integer.value > 65535)
        error(1205)
    Else
        INT.value = integer.value
}
```

Figure 28 4.3.1.2.3.2 Translation Schema of the semantic analyzer of the assembly code analyzer

4.1.3.2.3.3 Error Cases

Table 24 details the semantic errors detected by the semantic analyzer.

Error Code	Message
1200	Bifurcation to an address outside the allowed range.
1201	Memory limit exceeded.
1202	The same label cannot be declared again.
1203	Value of the expression outside the allowed range.
1204	Memory reserve exceeds the allowed memory limit.
1205	Integer outside the allowed range.
1206	Relative to index register addressing allows only the following registers: IX, IY
1207	Relative to index register addressing does not allow the SP register.
1208	Bifurcation relative to PC out of range.
1209	The label used has not been declared.
1210	Bifurcation to an address outside the memory boundaries.
1211	Relative addressing outside the allowed range.

Table 24 Assembly code analyzer semantic errors

4.1.3.3 Solution Design

The design of the Assembly Code Analyzer is based on the same modular and object-oriented approach employed in the quadruple compiler (Solution Design), adapted in this case to an architecture focused exclusively on analysis. Since its purpose is to validate already generated assembly code, no translation or code generation phase is included.

4.1.3.3.1 System Overview

The system is composed of several PHP classes that follow the same structural and collaborative model described for the quadruple compiler. Each component has a clearly defined responsibility in the analysis process, and error handling is centralized in a dedicated module.

The UML class diagram, Figure 29, presents the overall structure of the system:

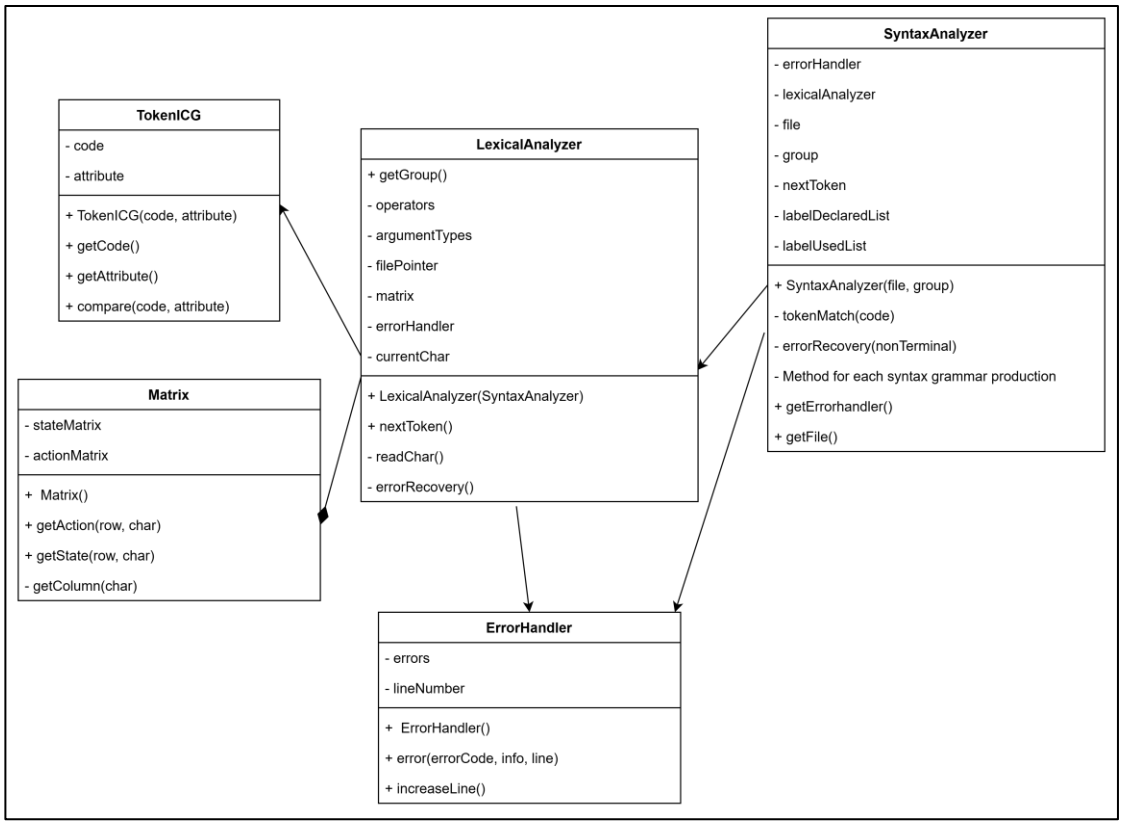

Figure 29 Assembly code analyzer UML diagram

4.1.3.3.2 Class-Level Software Design

The classes defined for this analyzer are:

- **`TokenEnsamblador`**
  Represents the lexical tokens of the assembly language, with methods for requesting and comparing their attributes.
- **`Matrix`**
  Implements the transition and action table for the finite-state machine used in lexical analysis.
- **`LexicalAnalyzer`**
  Responsible for scanning the source code character by character, recognizing tokens using `Matrix`, and invoking `ErrorHandler` in case of lexical errors. It includes the `errorRecovery` method to resume analysis after a fault.
  Unlike the quadruple compiler, this class does not include a `getDefinitions()` method, since there is no custom token naming functionality in the assembly code context.
- **`SyntaxAnalyzer`**
  Processes the token stream and verifies its syntax and semantic correctness. It also includes the logic for semantic analysis, ensuring that instructions, labels, and operands conform to the language's constraints. In this design, semantic verifications are embedded within the parser, following the typical recursive descent structure, and are invoked during the parsing of each construct. Furthermore, the class implements a error recovery mechanism, allowing the parser to continue analyzing the remaining input.

- `ErrorHandler`
  Manages and reports errors detected during any phase of analysis.

The core methods and functionalities of each class follow the same pattern already described in the quadruples compiler.

4.1.3.3.3 Component Interaction

The component interaction follows the same flow as previously described, with the sole difference that `SyntaxAnalyzer` does not interact with any translator module. It requests tokens from `LexicalAnalyzer`, which relies on `Matrix` to analyze the source code and encapsulate recognized tokens in `TokenEnsamblador` objects.

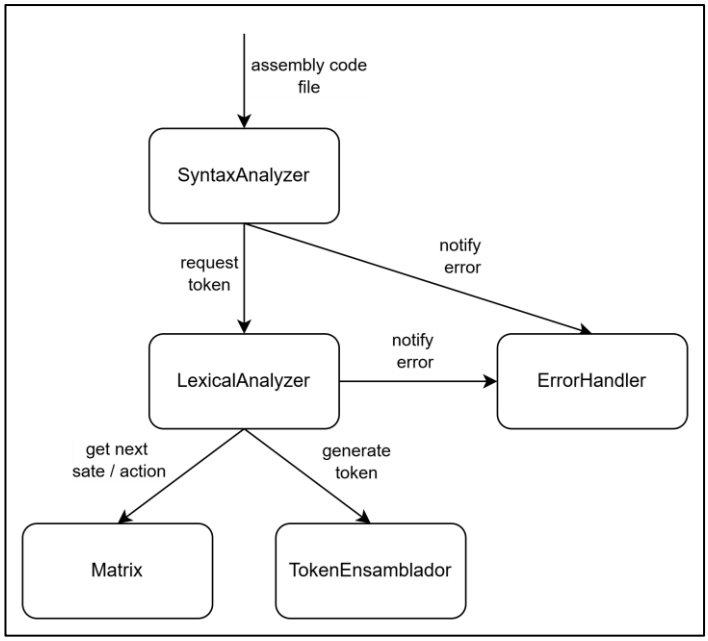

Figure 30 Assembly code analyzer component interaction diagram

When an error occurs, both the lexical and syntax analyzers report the issue through `ErrorHandler` and apply recovery procedures to resume analysis.

Figure 30 illustrates the interaction between the main components during the analysis process.

4.1.3.3.4 Design Considerations

Reusing the previously established architecture ensures consistency across the system and significantly reduces development effort. By preserving the same class architecture and naming conventions as the quadruples compiler, the design promotes consistency across the DRACO ecosystem, facilitating maintenance and easing future enhancements.

4.2 Implementation

This section details the most relevant aspects of the implementation of the developed components. It describes some of the key elements involved in the implementation of the analysis and translation phases of the quadruples compiler and the assembly code analyzer, with particular emphasis on the technical decisions made during development.

It is worth noting that the analysis phases of both modules, lexical, syntactic, and semantic, share a very similar structure and behavior. This is because both the compiler design and the proposed software solution are based on the same conceptual foundation.

Despite their structural similarities, the two modules differ in the specific grammars and semantic actions they implement. Each analyzer uses a different lexical and syntactic grammar adapted to the structure of its target language, and the semantic actions are tailored to the validation rules and meaning of each context.

The following subsections present some of the most significant implementation decisions.

4.2.1 Transition Matrix of the Lexical Analyzer

The lexical analyzer logic for the assembly code has been implemented using a class named Matrix, which is responsible for modeling the deterministic finite automaton through two key structures: the state matrix (`$stateMatrix`) and the action matrix (`$actionMatrix`). Both matrices represent the automaton's behavior based on the current state and the type of character read.

The detailed design of these matrices was previously developed as part of the lexical automaton analysis, covering all possible combinations of state transitions and associated actions. Figure 31 shows a representative excerpt from this design matrix, which served as the direct basis for its code implementation.

Thus, depending on the current state (first column) and the character or set of characters read from the file (first row), the left cell provides the new state, while the right cell indicates the action to be executed or the error code to be returned.

	space tab		–		letter - {A-F,x}		x		A-F		digit$_{1\text{-}9}$		0	
0	0	1	19	18	19	18	19	18	19	18	21	21	23	24
15	18	17	18	17	18	17	18	17	18	17	18	17	18	17
17	18	17	18	17	18	17	18	17	18	17	18	17	18	17
19	20	20	19	19	19	19	19	19	19	19	19	19	19	19
21	22	23	22	23	22	23	22	23	22	23	21	22	21	22
23	22	25	22	25	22	25	24	1	22	25	21	22	21	22
24		1005		1005		1005		1005	25	26	25	26	25	26
25	22	23	22	23	22	23	22	23	25	27	25	27	25	27
27	27	29	27	29	27	29	27	29	27	29	27	29	27	29
29		1008		1008	30	31	30	31	30	31		1008		1008
30	31	33	31	33	31	32	31	32	31	32	31	32	31	32
32	32	1	32	1	32	1	32	1	32	1	32	1	32	1

Figure 31 Excerpt from the design of the transition and state matrix of the lexical analyzer for the assembly code analyzer

To improve code maintainability and readability, the various lexical character types are defined using constants within the class itself. These constants allow for clear and controlled indexing of both the columns of the state matrix and the action matrix. Likewise, the automaton states are also defined as named constants, which contributes to more readable and maintainable code.

In the constructor of the class, the state matrix and the action matrix are initialized as two-dimensional arrays. For example, in Code 10, the representation of the state transitions from state 0 is shown.

```
$this->stateMatrix = array(
      self::ESTADO_0 => array(
      self::DELIMITADOR => 0,
      self::BARRA_BAJA => 19,
      self::OTRAS_LETRAS => 19,
      self::X => 19,
      self::LETRAS_AF => 19,
      self::DIGITOS => 21,
      self::CERO => 23,
      …
```

Code 10 Snippet of the state transition matrix of the lexical analyzer for the assembly code analyzer

In addition to the constructor, the `Matrix` class provides three essential methods used during lexical analysis:

- `getState($state, $column)`: returns the next state based on the current state and the type of character read.
- `getAction($state, $column)`: returns the action code associated with the transition.
- `getColumn($character)`: classifies an individual character into one of the lexical categories defined by the internal constants and returns the corresponding column in the matrix.

  The `getColumn` function plays a critical role in the analysis flow, as it transforms a character read from the source file into a valid index for accessing the matrices. Its implementation relies on a series of chained conditions that evaluate whether the character belongs to a specific group (letters, digits, specific symbols...), returning the corresponding constant value.

  For example, if the character is a letter between 'A' and 'F', it is classified as `LETRAS_AF`. A representative code snippet of this function is included below, Code 11.

```
private function getColumn($char){
    $column = self::OTRO_CARACTER;
    if ($char=="\t" || $char == " ") {
        $column = self::DELIMITADOR;
    }
    elseif (preg_match('/^[A-Fa-f]$/', $char)){
        $column = self::LETRAS_AF;
    }
    …
    return $column;
}
```

Code 11 Excerpt from the implementation of the getColumn function

4.2.2 Database Reading of Student-Defined Entries

In the context of the lexical analyzer of the quadruples compiler, a functionality has been incorporated that allows the system to adapt to the custom definitions of reserved words configured by each student practice group.

This functionality is implemented through the `getDefinitions($group)` method, whose purpose is to retrieve from the DRACO database the names assigned by the practice group to the operators and argument classes. This method is executed at the beginning of the lexical analysis of the quadruples file.

The function establishes a connection to the database and checks whether the $group parameter is null. If so, it assigns the value 0, which corresponds to the system's default group, containing the standard definitions used by the platform. It then performs an SQL query to retrieve the relevant fields from the `draco_gci.definicion_grupo` and `draco_gci.type_palabrares_gci` tables, linked via the `palRes_id` identifier.

For each entry, the query returns the reserved word identifier, the lexeme defined by the group, and a boolean flag (`esOperador`) that indicates whether the term corresponds to an operator or to an argument class. The results are processed and stored in two internal structures of the class: `operators` and `argumentTypes`, which are later used to validate and classify tokens during the lexical analysis.

The implementation of this functionality is shown in Code 12.

```
private function getDefinitions($group){
   $db_link = connect_db();
   if(is_null($group)){
      $group=0;
   }
   $query_result = mysqli_query($db_link, "SELECT dg.palRes_id, lexemaGrupo,
esOperador FROM draco_gci.definicion_grupo dg INNER JOIN
draco_gci.type_palabrares_gci tp ON dg.palRes_id=tp.palRes_id
WHERE idGrupo= ".$group." ORDER BY 'dg'.'palRes_id' ASC");
   $definitions = mysqli_fetch_all($query_result);
   for($i = 0; $i<sizeof($definitions); $i++){
      if($definitions[$i][2]==1){
         $this->operators[$definitions[$i][0]]=$definitions[$i][1];
      }
      else{
         $this->argumentTypes[$definitions[$i][0]]=$definitions[$i][1];
      }
   }
}
```

Code 12 GetDefinitions method

4.2.3 Semantic Verifications Performed by the Semantic Analyzer

In the rule P of the Translation Schema of the semantic analyzer of the quadruples compiler, a final phase of semantic verifications is incorporated to validate key aspects that cannot be guaranteed during earlier stages of analysis.

One of the fundamental verifications consists of ensuring that all labels used in the program have been declared. To perform this check, two internal lists are maintained: `labelUsedList` and `labelDeclaredList`. The PHP function `array_diff` is used to identify labels that have been referenced but not defined. The resulting labels from this operation are reported as semantic errors.

Another important verification is the presence of the main label. This check is implemented using the `in_array` function, which evaluates whether the label is included in the list of declared labels.

Finally, the program must include at least one quadruple of type `HALT`, which is essential to mark the end of execution. This condition is checked using a boolean variable that is updated during the analysis.

These verifications serve as the final stage of the semantic analysis before the subsequent translation to assembly code.

The code corresponding to these verifications is shown in fragment Code 13.

```
$notDeclared = array_diff($this->labelUsedList, $this-> labelDeclaredList);
foreach($notDeclared as $labelError){
   $this->errorHandler->error(227, $labelError, false);
}

if (!in_array("main", $this->labelDeclaredList)) {
   $this->errorHandler->error(229, "", false);
}

if($this->halt===false){
   $this->errorHandler->error(226, "", false);
}
```

Code 13 Final semantic verifications of the quadruples compiler

4.2.4 Error Recovery During Analysis

In the lexical analyzers, parsers and semantic analyzers developed, mechanisms have been implemented to allow the analysis process to continue after an error is detected. This recovery strategy has been applied both in the quadruples compiler and in the assembly code analyzer, using a common approach in both cases.

In the lexical analyzers, recovery is implemented through the `errorRecovery()` function, which is invoked when an invalid transition is encountered in the automaton matrix. The function reads characters sequentially until it reaches a delimiter, a line break, or the end of the file. Once this process is complete, the automaton returns to its initial state and resumes the analysis.

Code 14 shows the implementation of the `errorRecovery` function used in the lexical analyzer of the assembly code analyzer.

```
private function errorRecovery(){
   while($this->currentChar != "\t" && $this->currentChar != " " && $this-
>currentChar !==false && $this->currentChar != "\n" && $this->currentChar !=
"\r"){
      $this->readChar();
   }
   return Matrix::ESTADO_0;
}
```

Code 14 ErrorRecovery method in the lexical analyzer of the assembly code analyzer

In the syntactic analyzers, the `errorRecovery($noTerminal)` method handles error recovery when a token is found that does not match the expected one by the grammar. To achieve this, the Follow set associated with the non-terminal in the production where the error occurred is consulted, along with the line break (`RC`) and end-of-file (`EOF`) tokens. The analyzer skips tokens until it finds one belonging to this set, at which point it resumes the analysis from a stable state.

As for semantic error recovery, it does not require a specific implementation. Thanks to the design of the syntax-directed translation scheme, semantic validations are embedded within the parsing process itself. Therefore, when a semantic error occurs, it is reported immediately without interrupting the overall analysis flow.

Code 15 presents the implementation of the `errorRecovery` function in the parser of the quadruples compiler.

```
private function errorRecovery($noTerminal){
   switch ($noTerminal){
      case "P":
         $noTerminal=array();
         break;
      case "S":
         $noTerminal=array(TokenICG::PARENTESIS_ABRIR);
         break;
      case "A":
         $noTerminal=array(TokenICG::COMA, TokenICG::PARENTESIS_CERRAR);
         break;
      case "NUM2":
      case "VAL":
         $noTerminal=array(TokenICG::LLAVE_CERRAR);
         break;
      case "NUM":
         $noTerminal=array(TokenICG::COMA, TokenICG::LLAVE_CERRAR);
         break;
      default:
   }
   array_push($noTerminal, TokenICG::RC, TokenICG::EOF);
   while(!in_array($this->nextToken->getCode(), $noTerminal)){
      $this->nextToken = $this->lexicalAnalyzer->nextToken();
   }
}
```

Code 15 ErrorRecovery method in the syntax analyzer of the quadruples compiler

4.2.5 Quadruples Translator

The translation of quadruples into assembly code has been implemented through the `genCO` function, which serves as the central component of the process.

This function uses a switch block that maps each operator of the intermediate language to its corresponding assembly instruction template (annex Translation Template for Quadruples). In each case, the function generates the required sequence of translation using the arguments of the quadruple as operands, and writes it directly to the output file using `fprintf` statements.

A key element in the implementation of `genCO` is the use of the auxiliary function `direction`, which abstracts access to either the value or the memory address of an argument, depending on the context. This function takes as input an argument and an optional parameter that specifies whether the memory address should be obtained instead of the value. If the address is requested, the necessary code is generated to compute it and store it in a temporary register. Otherwise, the function returns the expression required to access the value directly, which may be a constant, an indexed memory access, or a label.

The use of `direction` adds flexibility and reusability to the translation templates, allowing a single structure to adapt to different types of arguments without duplicating logic or introducing manual exceptions.

```
function genCO($operator, $arg1=null, $arg2=null, $arg3=null){
   $translationCO="";
   switch ($operator){
      case PRINT_CAD:
         $translationCO= "WRSTR ";
         $direccion = $this->direction($arg3);
         $translationCO.=$direccion;
         break;
      …
   }
   fprintf($this->fileCode, $translationCO."\n");
}
```

Code 16 Excerpt from genCO function

```
private function direction($arg, $referenceDirection=false){
   $translationArg="";
   switch ($arg[0]){
      case VAR_GLOBAL:
         if($referenceDirection){
               $translation= "ADD #".$arg[1].",.IY \n";
               $translation.="MOVE .A, .R5 \n";
               fprintf($this->fileCode, $translation);
               $translationArg=".R5";
         }
         else {
               $translationArg = "#" . $arg[1] . "[.IY]";
         }
         break;
      …
   }
   return $translationArg;
}
```

Code 17 Excerpt from direction function

This approach, based on a centralized switch and a context-aware addressing function, has resulted in a clear, maintainable implementation aligned with the design established for the quadruples translator.

Fragments Code 16 and Code 17 respectively show representative parts of the implementation of the `genCO` and `direction` functions described in this section.

4.3 Testing

This chapter presents the testing activities carried out throughout this work. The main objective of this phase was to verify that the system meets the established functional requirements and behaves robustly when faced with different types of input. The following sections describe the criteria followed in the design and execution of the tests, as well as the results obtained and some notable errors that were identified during the process.

4.3.1 Testing Methodology

The testing process applied the control flow testing technique using the statement coverage criterion, whose purpose is to ensure that all executable

instructions in the system are executed at least once during the validation phase. [41]

The choice of this technique is justified by the nature of the components developed, which are predominantly structured around control constructs such as switch blocks and if-else statements. In both the quadruples compiler and the assembly code analyzer, the processing logic is articulated through multiple conditions that lead to different execution paths. For this reason, it was considered essential to design test cases that would force the execution of each branch of the control flow.

The tests were structured around the manual design of input files for the various components of the system. Once processed by the corresponding component, the output was verified and compared against the expected behavior. This comparison allowed for the validation of each component's correct functionality and the detection of deviations from the defined requirements.

The testing process followed an incremental approach, in which each component was validated individually after its implementation. Once a module was completed, specific test cases were designed and executed to verify its behavior in isolation. Subsequently, as the different components were integrated, combined tests were carried out to ensure proper interaction among them and the expected global behavior of the system.

In addition to the tests, during the design of the translation from quadruples to assembly code, the ENS2001 simulator was used to validate the correctness of the defined translation templates (8.2). To this end, the quadruples compiler was executed with sample quadruple inputs, and the resulting assembly code was run in the ENS2001 simulator to verify that it behaved as expected.

4.3.2 Designed Test Cases

To ensure the quality and robustness of the system, a total of 25 test programs were created and distributed across the various components developed: 10 for the quadruples analyzer of the compiler, 8 for the assembly code generator, and 7 for the assembly code analyzer.

Test cases were organized by component, covering independently the lexical, syntactic, and semantic phases of both the quadruples compiler and the assembly analyzer, as well as the object code generation phase. This modular division allowed for the validation of each phase in isolation and ensured that it met the defined requirements before proceeding with integrated testing.

Each test file was manually designed with the aim of evaluating specific system behaviors and verifying the handling of different language constructs. For each component, test cases were included to ensure the execution of all relevant control flow paths, thereby covering all defined lexical symbols, tokens, valid quadruples, and the full set of instructions in the assembly language.

The test files included both valid cases and cases containing intentional errors. Some focused exclusively on correct inputs, others on specific errors, and several combined both types to evaluate the system's behavior in more realistic scenarios. This strategy made it possible to verify error detection in all expected situations, as well as the generation of clear messages and the system's recovery capabilities.

Each test was associated with a predefined expected result. Once the file was processed by the corresponding component, the output was verified against the expected behavior. This systematic verification process allowed for the validation of each module's correctness and the early detection of any deviations from the specification.

4.3.3 Representative Examples and Error Resolution

The following section presents a selection of test cases that proved especially relevant during the system validation phase. These cases helped identify issues in various components, analyze their root causes, and apply the necessary corrections to ensure proper system behavior.

4.3.3.1 Infinite Loop Caused by Incorrect End-of-File Detection

For the initial test performed on the lexical analyzer of the assembly code analyzer, an input file was designed containing multiple lexical symbols to verify the correct recognition of tokens. The purpose of this test was to ensure that all characters were properly processed and that the corresponding tokens were generated according to the lexical grammar.

However, upon executing the test, the system entered an infinite loop and failed to complete the file reading process. After analyzing the program's behavior, it was determined that the lexical analyzer was not correctly identifying the end of file.

The issue was located in the auxiliary function `getColumn()` of the Matrix class, which is responsible for interpreting the characters read. The function used for reading characters was `fgetc()`. In PHP, the end of file can be interpreted either as `false` or as the integer `0`, which caused the `EOF` to be incorrectly treated as the integer `0`, resulting in an infinite loop due to the analyzer never detecting the actual end of the file.

To resolve the error, the evaluation of the read character in the function was modified to use a strict comparison (`===`). By updating the condition, the ambiguity with the `EOF` was eliminated, allowing the lexical analyzer to correctly identify the end of file and properly terminate the reading process. This correction can be seen in Code 18, which shows the change applied in the `getColumn()` function.

```
private function getColumn($char){
    $column = self::OTRO_CARACTER;
    if ($char=="\t" || $char == " ") {
           $column = self::DELIMITADOR;
    }
    …
    elseif ($char === '0'){
           $column = self::CERO;
    }
    …
    elseif ($char === false){
           $column = self::EOF;
    }
    …
    return $column;
}
```

Code 18 Correction of eof detection in getColumn function of the assembly code analyzer scanner

4.3.3.2 Incorrect String Length Calculation with Escape Sequences

During one of the tests designed to validate the lexical analyzer of the quadruples compiler, an input file was used that included string literals as part of the quadruples. The purpose of this test was to verify that strings were correctly recognized and that the system respected the maximum length constraints defined.

While running the test, it was observed that certain strings were incorrectly flagged as exceeding the maximum length, even though their apparent length was within the allowed limit. Upon analyzing the content of the affected strings, it was determined that the issue occurred when the string contained escape sequences such as the newline character (\n). These sequences were being incorrectly counted as two characters instead of one, leading to an inflated total length calculation.

The root of the problem was located in semantic action number 16 of the lexical analyzer. This action is responsible for reading the closing character of a string, checking its length, and, if valid, generating the corresponding token.

```
case 16:
   $this->readChar();
   $escapeSequences = substr_count($string, '\n');
   $escapeSequences += substr_count($string, '\t');
   $escapeSequences += substr_count($string, '\\');
   $escapeSequences += substr_count($string, '\0');
   if(strlen($string)-$escapeSequences>=$this->stringMaximumSize){
      $this->errorHandler->error(26);
   $state = $this->errorRecovery();
   break;
   }
   else {
      return new TokenICG(TokenICG::CADENA, $string);
   }
```

Code 19 Adjustment of string length calculation in semantic action 16 of the quadruples compiler scanner

To fix the issue, the semantic action was modified to explicitly detect the presence of escape sequences in the string. The total count of escape sequences is then used to adjust the length of the string, subtracting one for each sequence, so that each is correctly counted as a single character.

The updated implementation of semantic action number 16 is shown in Code 19.

4.3.3.3 Declaration of a Previously Declared Label in the Quadruples File

In one of the tests performed on the quadruples compiler, an issue was detected related to the declaration of labels. Specifically, an input file was designed that included the same label name declared twice. When the test was executed, the system did not produce any error and allowed the duplicate declaration without reporting any issue.

Upon analyzing the compiler's behavior, it was found that this verification was not considered in the initial design of the semantic analyzer. As a result, no corresponding check had been implemented in the source code, which allowed

the repeated use of labels and potentially compromised the consistency of the translated program.

To resolve the issue, the design of the semantic checks in the quadruples compiler was updated to incorporate the restriction that a label cannot be declared more than once. This verification was explicitly added to the code as part of the validation logic for quadruples that declare labels.

The implemented check verifies whether the label already exists in the list of previously declared labels (`labelDeclaredList`). If so, the corresponding semantic error is triggered. This correction can be found in Code 20, which shows the updated logic for quadruples of type `ETIQ`.

```
case ETIQ:
   if($a1Type[0] == TIPO_VACIO || $a2Type[0] != TIPO_VACIO ||
$a3Type[0]!=TIPO_VACIO){
      $this->errorHandler->error(217);
   }
   else if($a1Type[0]!=ET){
      $this->errorHandler->error(221);
   }
   else if(array_search($a1Type[1], $this->labelDeclaredList)!==false) {
      $this->errorHandler->error(224, $a1Type[1]);
   }
   else{
      $this->labelDeclaredList[]=$a1Type[1];
      $sType=TIPO_OK;
   }
break;
```

Code 20 Semantic verification of duplicate label declarations in the quadruples compiler

5 Results and Conclusions

5.1 Achieved Results

This Master's Thesis has successfully completed the development of the components defined within the generic compiler project for the *Traductores de Lenguajes* course assignments. The work has addressed the three main areas outlined in the project's objectives: the design of the intermediate quadruples language, the development of the compiler that translates this language into assembly code, and the implementation of the assembly code analyzer.

Each of these components has gone through a complete development cycle, including requirements analysis, detailed design, implementation, and functional validation. This approach has led to the construction of a structured and coherent system, prepared for future integration into the DRACO platform and aligned with the overall needs of the project.

The first completed workstream was the definition of the intermediate quadruples language, which students are expected to use in their assignments. This language serves as a bridge format between high-level source code and the final assembly code, facilitating the representation of instructions.

The second component is the quadruples compiler, which enables students to validate the intermediate code they have produced. This compiler performs a series of checks on the quadruples, detecting lexical, syntactic, or semantic errors. As a result of this analysis, the system provides students with a list of detected errors, allowing them to debug and correct the faults in their compiler. In addition to this validation functionality, the compiler also automatically generates the corresponding assembly code from the correct quadruples, thereby completing the synthesis phase of the quadruples compilation process.

Lastly, an assembly code analyzer has been developed to verify that the assembly programs produced by students conform to the rules of the target language. This component performs a thorough three-level analysis—lexical, syntactic, and semantic—enabling the detection of various types of errors. As with the quadruples compiler, the analyzer provides a detailed report of the detected issues.

All modules have been subjected to a set of functional tests designed to evaluate their behavior under various input conditions. Both correct and intentionally erroneous cases —reflecting real usage scenarios— were tested to ensure that the system can accurately detect faults and provide useful feedback to students. The results obtained confirm that the components behave reliably and consistently in line with the established objectives.

Overall, the results achieved demonstrate that the developed system fully meets the intended functionalities and is ready for integration with the remaining components that will be developed as part of the generic compiler project for the *Traductores de Lenguajes* course assignments.

5.2 Impact

The work carried out represents a significant step forward within the framework of the generic compiler project for the *Traductores de Lenguajes* course assignments. Until now, this course did not have any system that allowed students to automatically validate their assignments before submission, which

meant that students had to rely exclusively on their own testing methods, without a dedicated support tool to provide objective and detailed feedback.

This situation contrasts with the reality in the *Procesadores de Lenguajes* course, where a consolidated system already exists to validate the analyzers implemented by students. In this regard, the system developed in this thesis helps to close that gap by offering an analogous and coherent support environment for the *Traductores de Lenguajes* course, thereby improving students' overall learning experience.

From the perspective of the broader project, the components developed —the definition of the intermediate quadruples language, the quadruples compiler with translation to assembly code, and the assembly code analyzer— constitute the functional core of the generic compiler. Their implementation marks a milestone in the progress of the complete system, which still requires the development of other modules, such as the source language compiler and the assembly code executor. The existence of a robust and validated foundation, as achieved in this work, provides the technical groundwork upon which the remaining components will be integrated, bringing the full system closer to operational viability.

In terms of its concrete utility, the system offers students a direct and effective support tool for the development of their assignments. Through the validation capabilities implemented in the various components, students will be able to detect lexical, syntactic, and semantic errors in both the intermediate code and the generated assembly code. This immediate feedback will allow them to debug their compilers more precisely, better understand the nature of the errors they encounter, and improve the overall quality of their implementation. The system thus serves as a comprehensive validation environment that guides students throughout the synthesis phase of the compiler and reinforces their learning through experimentation and iterative correction.

Moreover, the integration of this system into the DRACO platform not only benefits students, but also offers indirect advantages to the teaching staff by enabling detailed monitoring of students' progress. With access to information about the tests carried out by each student, instructors will be able to identify recurring error patterns, common challenges, or problematic areas in the development of assignments. This facilitates informed pedagogical decisions and allows instructors to adjust theoretical explanations based on observed evidence.

In summary, the system developed constitutes a key functional component within the DRACO ecosystem, significantly expanding its scope and usefulness. Its incorporation strengthens the platform's objective of providing active learning support tools and is fully aligned with the principles of autonomy, continuous feedback, and instructional coherence that underpin the generic compiler project.

5.3 Final Conclusions

The completion of this Master's Thesis has been a unique opportunity to consolidate and apply in a practical way the knowledge acquired throughout both my undergraduate and master's studies, particularly in relation to compiler design, as studied in the courses *Procesadores de Lenguajes* and *Traductores de Lenguajes*. Having previously taken these subjects provided me with a solid theoretical foundation to approach this project; however, it has been

through the development of this work that I have truly been able to verify the applicability and real-world value of that knowledge in a structured development environment with detailed objectives.

I would like to emphasize that, having been a student in the *Traductores de Lenguajes* course myself, I am aware of the challenges involved in completing the practical assignments without a validation tool that allows students to independently test their compiler. This personal experience significantly reinforced my commitment to the project and served as a key source of motivation, knowing that the system developed could support the learning process and improve the experience of future students.

This work has also allowed me to directly understand the complexity of undertaking a real development project within the context of a broader collaborative initiative. Throughout the process, various adjustments and decisions had to be made to address evolving requirements and unforeseen issues. In this context, regular meetings with the DRACO platform administrator were essential to validate proposals, align objectives, and revise solutions when necessary. This dynamic has helped me strengthen essential skills such as adaptability and justified technical decision-making.

In short, this Master's Thesis has provided an opportunity to apply not only my knowledge of compiler design, but also the broader set of competencies acquired through the Master's in Software Engineering. Throughout the development process, I have participated in all stages of the software development lifecycle, which has allowed me to build a comprehensive understanding of the process, face real-world challenges, apply learned methodological principles, and assume technical and structural responsibilities characteristic of a professional development environment.

6 Future Lines of Work

The work presented in this document represents a core component of the development project for the DRACO generic compiler. However, the system is not yet complete and offers various opportunities for improvement and expansion, both to finalize the originally intended functionality and to enhance its future pedagogical value and utility. This chapter outlines the main lines of work that could be pursued to complete and further evolve the project.

6.1 Continuation of the Generic Compiler Project

This section presents the main lines of work that remain to be addressed to complete the generic compiler system within the DRACO platform.

6.1.1 Development of the Source Code Compiler

One of the key lines of work to complete the generic compiler project is the development of the source language compiler, responsible for processing the source code of the tests generated by DRACO and transforming it into its corresponding intermediate representation in the form of quadruples. This line of work is already in the design phase, although its implementation has not yet been completed.

This component constitutes the entry point of the generic compiler. Its role is to perform lexical, syntactic, and semantic analysis of the provided source code and, based on that, generate the intermediate code using the quadruples language defined in this Master's Thesis. The design relies on the definition of a generic source language, flexible enough to be adapted each academic year to the learning objectives established by the teaching staff. To this end, it is expected that instructors will be able to introduce the specific characteristics and grammar corresponding to the chosen language, enabling the automation of both the analysis and synthesis phases of the compiler.

This source language compiler will enable the automatic compilation of the test programs generated in DRACO in source language format, so that —together with the remaining components of the generic compiler— it can be translated through to its final execution.

This functionality will allow for the comparison between the execution results of these reference programs and those obtained from the code provided by the students, whether in the form of quadruples or assembly. In doing so, the system will validate the behavior of the compilers developed by the students during the *Traductores de Lenguajes* course practices.

6.1.2 Implementation of the Assembly Code Executor

Another key line of work in the generic compiler project is the development of the assembly code executor, responsible for executing programs after they have been analyzed and validated by the assembly code analyzer. This component must process the assembly instructions and produce an output that reflects the behavior of the program.

Its implementation requires defining an execution model that includes register management, a data stack, a memory area, and control flow mechanisms. Additionally, it will be necessary to implement support for all instructions

defined in the assembly language, including arithmetic operations, bifurcations, memory access, and input/output operations, among others.

The assembly code executor is a fundamental component of the project, as it represents the final stage of the system. Its role is to execute both the programs generated from the source code tests defined in DRACO and compiled by the generic compiler, and the assembly code provided by the students. This will make it possible to compare the outputs of both executions and thereby validate the behavior of the compilers developed by the students. The development of this feature is planned to begin in the upcoming academic year.

6.1.3 Integration of the Components into DRACO

Once all the components of the generic compiler for the *Traductores de Lenguajes* course practices have been developed, it will be necessary to carry out their full integration and verify the correct operation of the system.

This integration involves connecting the source language compiler with the quadruples compiler developed in this work, and linking the assembly code analyzer with the assembly code executor. The entire process must operate in a coordinated manner, enabling the compilation and execution of source code.

In addition to this technical integration, it will be necessary to develop the execution logic that allows this system to be used as part of the activities within DRACO. This includes implementing the execution flow required to manage the input of programs generated by students, their analysis, and the subsequent comparison with the results obtained from the reference tests provided by the system.

The corresponding user interfaces for each of these activities must also be developed. However, a first version of the interfaces has already been implemented, providing the necessary functionality to configure the quadruples language and generate test cases for the analyzers. This development was carried out in parallel as part of the Bachelor's Thesis of Sergiu Petrila. [40]

Second, an activity will be implemented to validate the generation of intermediate code. In this case, the system will provide a sample source code that students must compile using their own compiler, producing a set of quadruples to be submitted to the platform for analysis and execution.

Similarly, an activity will be created to validate the generation of assembly code: the system will offer a source code test, and students must compile it into assembly code, which they will upload to the platform to be executed and compared against the execution of the reference test.

These interfaces must be connected to the entire execution and analysis system to ensure that input, processing, and validation flows function correctly. Additionally, views must be implemented to display the results obtained, the errors detected, and other relevant aspects such as the scoring system associated with each activity.

Finally, specific interfaces must be created for instructors, allowing them to configure various aspects of the activities, such as availability dates, the number of allowed attempts per group, and the difficulty level of the test cases.

6.2 System Evolution and Expansion

Beyond the core components required to complete the generic compiler, there are several additional lines of work that could further enhance the system. These proposals focus on extending its capabilities, increasing its educational value, and broadening its potential applications within the DRACO platform.

6.2.1 Extension of the Defined Languages

A potential line of system evolution involves expanding the expressive capabilities of the source language defined for the course practices. Incorporating new features into this language, such as support for non-local variables or more complex data structures, would make it possible to address more advanced practices within the course, depending on the instructional goals set by the teaching staff.

Such extensions would have a direct impact on the intermediate quadruples language and the assembly language defined in this work, requiring a full revision of all components of the generic compiler. Each module, from the source language compiler to the assembly code executor, would need to be adapted to support the new features, ensuring consistency and correct system behavior throughout the entire process.

Given the complexity involved in updating all components and modifying the entire system to increase the expressiveness of the source language, this project has adopted a design with a sufficiently broad expressive capacity to cover the current set of practices without requiring immediate changes.

However, this line of work could be relevant in the future if the aim is to raise the difficulty level of the course practices or introduce more advanced instructional content. Its implementation would broaden the system's pedagogical scope and allow it to adapt to new educational objectives.

6.2.2 Standalone Assembly Code Execution Tool

A potential line of work derived from this project is the creation of a standalone assembly code execution tool, built from the executor component defined as part of the generic compiler. The goal of this tool would be to provide students with an independent environment in which to execute and validate programs written in the assembly language defined for the *Traductores de Lenguajes* course practices.

The idea is to extract the executor from the main system and integrate it into a separate application that could be distributed independently for local use by students. This tool would allow students to experiment directly with the assembly language, either by manually writing code for learning purposes or by executing the assembly code generated by the compilers they develop during the course. In doing so, it would promote more active and autonomous learning regarding both the assembly language and the runtime behavior of their compiled programs.

Additionally, the tool could incorporate features such as step-by-step execution, visualization of register and memory states, and detailed error messages. These capabilities would offer clearer insight into the execution flow and make it easier to detect and understand errors, thereby enhancing the students' overall learning experience.

6.2.3 Gamification of the Generic Compiler Activities

A complementary line of evolution to the technical development of the generic compiler is the incorporation of gamification elements into the practical activities related to the *Traductores de Lenguajes* course. This proposal aims to increase student motivation by introducing game-like dynamics that promote competitiveness and engagement throughout the process of developing the compilers required for the course practices.

The core idea is to design an experience like a tournament, where the different student groups compete to complete, correctly and as early as possible, the various activities associated with the generic compiler. To achieve this, a points system and group ranking could be established, tracking each group's progress based on the tests they pass and the time at which they succeed.

In addition, it would be useful to incorporate real-time comparative metrics, such as indicating whether a group was the first to pass a given test, how many groups are currently tackling a certain difficulty level, or what stage of development other teams have reached. This type of information would provide a sense of competition and serve as an additional incentive for students to continue advancing in the development of their compilers.

All these features should be integrated into the DRACO, enhancing the existing activities with an additional gamification layer. This integration would help transform the practice validation process into a more engaging and dynamic experience, while preserving the core educational objectives of the course.

7 Bibliography

[1] F. J. Álvarez, “ENS2001 Manual de Usuario,” February 2003. [Online]. Available: https://ens2001.falvarez.es/files/ENS2001-Manual-de-usuario.pdf. [Accessed March 2025].

[2] IMMUNE Technology Institute, “¿Qué es un compilador?,” 20 April 2023. [Online]. Available: https://immune.institute/blog/que-es-un-compilador/. [Accessed January 2025].

[3] Keepcoding, “Invención del compilador: su historia,” [Online]. Available: https://keepcoding.io/blog/invencion-del-compilador-quien-lo-creo/. [Accessed January 2025].

[4] GeeksforGeeks, “Introduction of Lexical Analysis,” 18 October 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-lexical-analysis/. [Accessed January 2025].

[5] A. Pérez and . J. P. Caraça-Valente, *Procesadores de Lenguajes: Análisis Léxico (ETSIInf-UPM). Course Notes,* 2022.

[6] GeeksforGeeks, “Introduction to Syntax Analysis in Compiler Design,” 3 October 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/. [Accessed January 2025].

[7] J. L. Fuertes, *Procesadores de Lenguajes: Análisis Sintáctico (ETSIInf-UPM). Course Notes,* 2022.

[8] GeeksforGeeks, “Semantic Analysis in Compiler Design,” 22 April 2020. [Online]. Available: https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/. [Accessed January 2025].

[9] J. L. Fuertes, *Procesadores de Lenguajes: Análisis Semántico (ETSIInf-UPM). Course Notes,* 2022.

[10] TutorialsPoint, “Compilador Diseño - Tabla de Símbolos,” [Online]. Available: https://www.tutorialspoint.com/es/compiler_design/compiler_design_symbol_table.htm. [Accessed January 2025].

[11] A. Pérez and J. L. Fuertes, *Procesadores de Lenguajes: Tabla de Símbolos (ETSIInf-UPM). Course Notes,* 2022.

[12] GeeksforGeeks, "Intermediate Code Generation in Compiler Design," 16 October 2024. [Online]. Available: https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/. [Accessed February 2025].

[13] A. Pérez, *Traductores de Lenguajes: Generador de Código Intermedio (ETSIInf-UPM). Course Notes,* 2023.

[14] J. L. Fuertes, *Traductores de Lenguajes: Entorno de Ejecución (ETSIInf-UPM). Course Notes,* 2023.

[15] GeeksforGeeks, “Introduction of Object Code in Compiler Design,” [Online]. Available: https://www.geeksforgeeks.org/introduction-of-object-code-in-compiler-design/. [Accessed February 2025].

[16] A. Pérez, *Traductores de Lenguajes: Generador de Código Objeto (ETSIInf-UPM). Course Notes,* 2023.

[17] GeeksforGeeks, “Error Handling in Compiler Design,” 14 July 2023. [Online]. Available: https://www.geeksforgeeks.org/error-handling-compiler-design/. [Accessed January 2025].

[18] J. L. Fuertes, *Procesadores de Lenguajes: Gestor de Errores (ETSIInf-UPM). Course Notes,* 2022.

[19] L. Llamas, “Breve historia de la Programación Orientada a Objetos,” 14 May 2024. [Online]. Available: https://www.luisllamas.es/historia-programacion-orientada-objetos/. [Accessed January 2025].

[20] GeeksforGeeks, “Object Oriented Programming in C++,” 11 October 2024. [Online]. Available: https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/. [Accessed January 2025].

[21] A. McKee, “Programación funcional frente a programación orientada a objetos en el análisis de datos,” 3 May 2024. [Online]. Available: https://www.datacamp.com/es/tutorial/functional-programming-vs-object-oriented-programming?dc_referrer=https%3A%2F%2Fwww.google.com%2F. [Accessed January 2025].

[22] A. Gillis, “Object-oriented programming (OOP),” [Online]. Available: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP. [Accessed February 2025].

[23] PHP, “History of PHP and Related Projects,” [Online]. Available: https://www.php.net/manual/es/history.php. [Accessed January 2025].

[24] W3Techs, “Usage statistics of PHP for websites,” [Online]. Available: https://w3techs.com/technologies/details/pl-php. [Accessed February 2025].

[25] P. Johnson, “How Does PHP Work With The Web Server And Browser?,” 6 September 2023. [Online]. Available: https://foreignerds.com/how-does-php-work-with-the-web-server-and-browser/. [Accessed January 2025].

[26] Geeks for Geeks, “PHP Introduction,” September 2024. [Online]. Available: https://www.geeksforgeeks.org/php-introduction/. [Accessed January 2025].

[27] Uniwebsidad, “Breve historia de HTML,” [Online]. Available: https://uniwebsidad.com/libros/xhtml/capitulo-1/breve-historia-de-html. [Accessed January 2025].

[28] W3schools, “HTML Introduction,” [Online]. Available: https://www.w3schools.com/html/html_intro.asp. [Accessed January 2025].

[29] M. Coppola, “Qué es HTML y cómo utilizarlo,” 6 June 2022. [Online]. Available: https://blog.hubspot.es/website/html#que-es. [Accessed January 2025].

[30] Uniwebsidad, “Breve historia de CSS,” [Online]. Available: https://uniwebsidad.com/libros/css/capitulo-1/breve-historia-de-css. [Accessed January 2025].

[31] Developer.mozilla, “How CSS works,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works. [Accessed January 2025].

[32] B. V., “Tipos de estilos CSS,” 31 January 2023. [Online]. Available: https://www.hostinger.es/tutoriales/tipos-de-estilos-css. [Accessed January 2025].

[33] Diego Santos, "Introducción al CSS," 11 January 2021. [Online]. Available: https://blog.hubspot.es/website/que-es-css#ventajas. [Accessed January 2025].

[34] W3Schools, "SQL History," [Online]. Available: https://www.w3schools.in/sql/history. [Accessed January 2025].

[35] IONOS, “SQL Commands,” 29 November 2022. [Online]. Available: https://www.ionos.es/digitalguide/servidores/configuracion/comandos-sql/. [Accessed January 2025].

[36] K. Josende, “Gamificación en la educación/historiadelagamificacion,” 8 April 2018. [Online]. Available: https://es.wikibooks.org/wiki/Gamificaci%C3%B3n_en_la_educaci%C3%B3n/historiadelagamificacion. [Accessed January 2025].

[37] Y. Nasser, “What is Gamification?,” 23 September 2020. [Online]. Available: https://yamannasser.medium.com/gamification-576452f1c6dd. [Accessed January 2025].

[38] IEEE, "IEEE Recommended Practice for Software Requirements Specifications," October 1998. [Online]. [Accessed March 2025].

[39] "Formato del fichero de tokens para la Práctica," April 2025. [Online]. Available: https://dlsiis.fi.upm.es/procesadores/Documentos/Tokens.pdf.

[40] S. P. Petrila, "Diseño de la configuración y evaluación de comprobadores automáticos y mantenimiento de un sistema de gamificación para la educación," 2024.

[41] GeeksforGeeks, "Control Flow Software Testing," 25 September 2024. [Online]. Available: https://www.geeksforgeeks.org/control-flow-software-testing/. [Accessed March 2025].

[42] Y. Fernández, “Qué es el HTML5 y qué novedades ofrece,” 13 January 2021. [Online]. Available: https://www.xataka.com/basics/que-html5-que-novedades-ofrece. [Accessed January 2025].

8 Annexes

8.1 Quadruples File Format

Note: This annex is written in Spanish, as the course to which it belongs is taught in that language. The document is intended for students and is distributed in Spanish as part of the course materials.

Formato del Fichero con los Cuartetos Obtenidos por el Generador de Código Intermedio

Para facilitar la depuración y la corrección de la Práctica de Traductores de Lenguajes y, como se indica en la especificación de la misma (http://dlsiis.fi.upm.es/traductores/Practica.html), el Traductor deberá entregar obligatoriamente los resultados generados en varios archivos de texto (lista de cuartetos, programa en ensamblador). Se detalla aquí el formato que ha de tener el fichero donde se almacenarán los cuartetos obtenidos por el Generador de Código Intermedio.

El lenguaje no diferencia mayúsculas de minúsculas en los cuartetos, pero sí en el nombre de las etiquetas.

El fichero puede tener líneas vacías. Se permiten comentarios precedidos por dos barras (//). El comentario finaliza con el fin de línea (RC). Los comentarios pueden estar formados por cualquier carácter. Un comentario puede situarse en una línea tras el cuarteto o en una línea que tenga sólo el comentario.

Formato de los Cuartetos

Debe escribirse un único cuarteto por cada línea y un cuarteto no puede ocupar más de una línea. Los cuartetos estarán en el fichero en el orden en que son generados por el Generador de Código Intermedio, es decir, el primer cuarteto del fichero será el primer cuarteto que se haya generado.

Los *cuartetos* tendrán el siguiente formato:

```
del* ( del* Operador del* , del* Arg1 del* , del* Arg2 del* , del*
                        Result del* ) del* RC
```

Donde:

- **del*** → cualquier cantidad de espacios en blanco o tabuladores, o nada.
- **Operador** → nombre del operador especificado por el grupo. Está formado por letras, dígitos y subrayados, siendo el primero siempre una letra. Los operadores se consideran palabras reservadas.
- **Arg1**, **Arg2**, **Result** → argumentos del cuarteto. En caso de que el cuarteto no lo requiera, se indica dejándolo vacío o poniendo un guion (-). La descripción detallada de estos campos se muestra en el siguiente apartado.
- **RC** → salto de línea.

***Nota:** Como se ha dicho, en aquellos cuartetos en los que no se requiera rellenar los argumentos, estos estarán vacíos o podrán llevar un guion (-), pero se mantendrán las comas (,) que separan dichos campos siguiendo el formato establecido.

Ejemplo: `(RETURN, , ,)  (RETURN, -, -, -)`

Formato de los argumentos

Los tres campos se representan mediante tuplas *{Clase , Valor}*. Al igual que los operadores de los cuartetos, el campo *Clase* de los argumentos son palabras reservadas especificadas por cada grupo de prácticas.

Los argumentos tendrán el siguiente formato:

```
del* { del* Clase del* , del* Valor del* } del*
```

Donde:

- **`del`*** → cualquier cantidad de espacios en blanco o tabuladores, o nada.
- **`Clase`** → nombre de la clase del argumento especificado por el grupo en DRACO. Está formado por letras, dígitos y subrayados, siendo el primero siempre una letra. Los nombres de clase se consideran palabras reservadas, y no pueden coincidir con las palabras reservadas de los cuartetos.
- **`Valor`** → es el valor asociado a la clase. En función de la clase, este campo puede tener los siguientes formatos:
  - `número`: `[+|-]d`$^+$ → constante entera con signo opcional (puede usarse para representar un desplazamiento en la Tabla de Símbolos (TS), un valor entero…). El valor absoluto del número no puede ser mayor de 32767.
  - `cadena`: `"c`*`"` → constante de tipo cadena entre comillas dobles (puede usarse para representar una cadena, una etiqueta…). No puede contener ni saltos de línea ni EOF.
  - dos números: *`número`* `del`* `,` `del`* *`número`* → dos números como los descritos anteriormente (puede usarse para representar la profundidad y el desplazamiento en las variables no locales).

Lista de Cuartetos

Seguidamente se indican los cuartetos disponibles y que se pueden utilizar. No es obligatorio usar todos los cuartetos, sino solamente aquellos que sean necesarios para la práctica.

En la tabla siguiente se muestra una breve descripción, una representación en formato de código de 3 direcciones y una representación de cada cuarteto. Para esta representación se utiliza una palabra para el operador (los grupos de prácticas pueden utilizar otras palabras) y los argumentos necesarios para el cuarteto, así como su posición dentro del cuarteto. Deberán respetarse los argumentos usados y su posición dentro de cada tipo de cuarteto.

Descripción	Código 3d	Cuarteto
Operación aritmética SUMA	`x = y + z`	`(SUMA, y, z, x)`
Operación aritmética RESTA	`x = y - z`	`(RESTA, y, z, x)`
Operación aritmética PRODUCTO	`x = y * z`	`(MUL, y, z, x)`
Operación aritmética DIVISIÓN	`x = y / z`	`(DIV, y, z, x)`
Operación aritmética MÓDULO	`x = y mod z`	`(MOD, y, z, x)`

Descripción	Código 3d	Cuarteto
Operación aritmética EXPONENTE	`x = y ^ z`	`(EXP, y, z, x)`
Operación lógica AND	`x = y AND z`	`(AND, y, z, x)`
Operación lógica OR	`x = y OR z`	`(OR, y, z, x)`
Concatenación de cadenas	`x = y concat z`	`(CONCAT, y, z, x)`
Menos unario	`x = - y`	`(MENOS, y, , x)`
Operación lógica NOT	`x = not y`	`(NOT, y, , x)`
Asignación entre datos enteros	`x = y`	`(ASIG, y, , x)`
Asignación entre datos cadena	`x =c y`	`(ASIG_CAD, y, , x)`
Salto incondicional	`goto etiqueta`	`(GOTO, , , etiqueta)`
Salto condicional si iguales	`if x == y goto etiqueta`	`(GOTO_IG, x, y, etiqueta)`
Salto condicional si distintos	`if x != y goto etiqueta`	`(GOTO_DIST, x, y, etiqueta)`
Salto condicional si mayor	`if x > y goto etiqueta`	`(GOTO_MAY, x, y, etiqueta)`
Salto condicional si menor	`if x < y goto etiqueta`	`(GOTO_MEN, x, y, etiqueta)`
Salto condicional si mayor o igual	`if x >= y goto etiqueta`	`(GOTO_MAY_IG, x, y, etiqueta)`
Salto condicional si menor o igual	`if x <= y goto etiqueta`	`(GOTO_MEN_IG, x, y, etiqueta)`
Paso de parámetro entero	`param x`	`(PARAM, x, , )`
Paso de parámetro cadena	`paramc x`	`(PARAM_CAD, x, , )`
Paso de parámetro por referencia	`param& x`	`(PARAM_REF, x, , )`
Llamada a un procedimiento	`call etiqueta`	`(CALL, etiqueta, , )`
Llamada a una función con retorno de entero	`x = callf`	`(CALL_FUN, etiqueta, , x)`
Llamada a una función con retorno de cadena	`x = callfc`	`(CALL_FUN_CAD, etiqueta,,x)`
Retorno	`return`	`(RETURN, , , )`
Retorno de entero	`return x`	`(RETURN_ENT, , , x)`
Retorno de cadena	`returnc x`	`(RETURN_CAD, , , x)`
Imprime un dato entero	`print x`	`(PRINT_ENT, , , x)`
Imprime un dato cadena	`printc x`	`(PRINT_CAD, , , x)`
Lee un dato entero	`input x`	`(INPUT_ENT, , , x)`
Lee un dato cadena	`inputc x`	`(INPUT_CAD, , , x)`
Etiqueta	`etiqueta :`	`(ETIQ, etiqueta, , )`
Asignación de una dirección	`x = &y`	`(ASIG_DIR, y, , x)`
Asignación del valor de un puntero	`x = *y`	`(ASIG_PTR, y, , x)`
Asignación del valor de un puntero que apunta a una cadena	`x =c *y`	`(ASIG_PTR_CAD, y, , x)`
Asignación indirecta	`*x = y`	`(PTR_ASIG, y, , x)`
Asignación indirecta de cadena	`*x =c y`	`(PTR_ASIG_CAD, y, , x)`
Parada de ejecución	`halt`	`(HALT, , , )`

Estructura de los Argumentos

Seguidamente se indica la estructura de los argumentos disponibles para los cuartetos y que se pueden utilizar. No es obligatorio usar todas las clases de

argumentos, sino solamente aquellas que sean necesarias para la práctica. Así mismo, es posible tener varias clases con el mismo nombre.

En la tabla siguiente se muestra una breve descripción de las distintas clases de argumentos con la indicación de los valores permitidos, así como una representación del argumento. Para esta representación se utiliza una palabra para la clase de argumento (los grupos de prácticas pueden utilizar otras palabras) y los valores necesarios.

Descripción clase-valor	Tupla
Variable global - desplazamiento en la TS del ámbito global	`{VAR_GLOBAL, desplazamiento}`
Variable local - desplazamiento en la TS del ámbito local	`{VAR_LOCAL, desplazamiento}`
Variable no local - diferencia de profundidades - desplazamiento en la TS de dicha variable	`{VAR_NOLOCAL, profundidad, desplazamiento}`
Variable temporal - desplazamiento en la TS del ámbito local	`{VAR_TEMP, desplazamiento}`
Parámetro por valor - desplazamiento en la TS del ámbito local	`{PAR, desplazamiento}`
Parámetro por referencia - desplazamiento en la TS del ámbito local	`{PAR_REF, desplazamiento}`
Constante entera - valor de la constante	`{CTE_ENT, entero}`
Constante cadena - la propia cadena	`{CTE_CAD, cadena}`
Etiqueta - la propia etiqueta	`{ET, etiqueta}`

Consideraciones Adicionales

En el fichero debe aparecer el cuarteto `(ETIQ, {ET, "main"},,)` que define la etiqueta "main" y que deberá asociarse al comienzo del programa principal que contenga el fichero.

Las etiquetas deben empezar por una letra que va seguida de otras letras, dígitos o subrayados. Ningún otro carácter será válido en el nombre de una etiqueta. Se recuerda que en el nombre de las etiquetas sí se diferencia entre mayúsculas y minúsculas.

Ejemplos

Se muestran a continuación algunos ejemplos de ficheros que cumplen el formato (en los ejemplos se usan los operadores y nombres de clase mostrados en las tablas anteriores):

- **Ejemplo 1:**

```
( INPUT_ENT,    ,      , {VAR_GLOBAL , -22})     //input x siendo x una variable
global
(suma, {VAR_GLOBAL , -22} , { VAR_TEMP, +2} , {VAR_LOCAL , -1})
           //suma de una variable global y una temporal en una variable local
 (PRINT_ENT,    ,    ,  {VAR_NOLOCAL,  2 , 3})  //print x siendo x una variable
no local
  ( Asig, { CTE_ENT, -12345},      , {Var_Local , 1} )
   (  NOT, {VAR_NOLOCAL , 77, -20},    , {VAR_LOCAL , 2}  )
(NOT, {VAR_LOCAL , 2},  , {VAR_GLOBAL , 2})
(Param_Cad, {VAR_GLOBAL, 23},-,-)
(RETURN_ENT,,, {VAR_TEMP, 32})
```

```
 ( halt,,,)
(ETIQ, {ET, "main"},,)                          //etiqueta del programa principal
(asig, { VAR_TEMP, 2}, , { VAR_LOCAL, 55})
(ASIG, {cte_ent, 00},   , {VAR_LOCAL , 0})
(RETURN, , , )
(PRINT_Cad,  ,  , {CTE_CAD, ""} )
(INPUT_ENT,-,-, {VAR_NoLocal, - 5, + 87})
(RETURN_CAD, - , - , { VAR_Local, -2})
(ASIG, {PAR_REF, 0},  , {var_LOCAL, -321})
(ETIQ, {et , "end" }, ,-)
(ASIG, {Cte_ENT , +23456},,{VAR_LOCAL, 0})
```

- **Ejemplo 2:**

```
// A continuación, se muestra el código intermedio de funcion1:
(GoTo,-,-,{Et , "main"})
(ETIQ ,   { ET , "funcion1" }   ,-    ,   -)
( ASIG, {PAR, 0}, , {VAR_LOCAL, 1})
( PRINT_ent ,    ,  - ,   { VAR_LOCAL ,  1} )
( ASIG, {PAR,0},   , {VAR_LOCAL,2})
( ASIG, {CTE_ENT,5}, , {VAR_LOCAL,3})
(GOTO_May, {VAR_LOCAL , 2 }, {VAR_LOCAL, 3}, {ET, "etiq0"})
( ASIG, {PAR , 0},     , {VAR_LOCAL , 4})
( ASIG, {CTE_ENT, -1},   , {VAR_LOCAL , 5})
( SUMA, {VAR_LOCAL , 4} , { VAR_LOCAL, 5 } , {VAR_TEMP , 6} )
(PARAM , {VAR_TEMP , +06}, , )
(CALL,{ET, "funcion1"},- ,- )
(etiq , { et , "etiq0" } ,,)
  (RETURN,,,)

// Código intermedio del ¡programa principal! que llama a funcion1:
(ETIQ , {ET , "main"} , - , - )
  (ASIG, {CTE_ENT, 0}, , {VAR_LOCAL , 0})
  (PARAM, {VAR_LOCAL , 0}, , )
  (Call,  {et , "funcion1"} ,  -  , -)
  (PRINT_CAD, -, , {CTE_CAD, "<<+ ¡fin! +>>"})
(halt,-,-,-)
```

Seguidamente se muestra un ejemplo de fichero que no cumple el formato (todos los cuartetos son incorrectos):

- **Ejemplo 3:**

```
INPUT_ENT,  ,  , {VAR_GLOBAL , -22}
                                  //ERROR: el cuarteto debe ir entre paréntesis
(NOT, VAR_LOCAL , 2,  , {VAR_GLOBAL , 2} )
                                  //ERROR: faltan llaves en el primer argumento
(PARAM_CAD, {VAR_GLOBAL, 02} )
                        //ERROR: el cuarteto está incompleto, faltan dos campos
(RETURN, {VAR_TEMP, 2}, , )
  //ERROR: en la estructura del cuarteto que representa return x no se usa el
  //       segundo y el tercer campo; el argumento debe situarse en el cuarto
(ASIG, { VAR_TEMP, 2},  { VAR_TEMP, 3}, { VAR_LOCAL, 5} )
       //ERROR: el segundo argumento debe estar vacío: (ASIG, Arg1, , Result)
(ASIG, {CTE_ENT, 0},   , {VAR_LOCAL , 0})   (RETURN,  ,  ,  )
                                          //ERROR: dos cuartetos en la misma línea
{SUMA,(Par,8),(CTE_ENT,5),(VAR_LOCAL,4)}
                                    //ERROR: uso incorrecto de llaves y paréntesis
(INPUT_ENT,-,-, {VAR_NOLOCAL, 1})
```

```
                              //ERROR: falta un valor para la variable no local
(RETURN, , , { VAR_LOCAL, -2} )
                     //ERROR: el cuarteto debe tener los tres argumentos vacíos
(ASIG, {VAR_LOCAL, 0},  , {CTE_ENT , 0})
                                    //ERROR: no se puede asignar a una constante
(ETIQ, {ET , funcion1}, ,-)
               //ERROR: funcion1 debe ir entre comillas porque es una etiqueta
(ASIG, {VAR_LOCAL, 0}, _ , {CTE_ENT , 0})
                              //ERROR: el tercer campo debe ser vacío o guion (-)
(INPUT_CAD,  ,  , {VAR_NO_LOCAL , 1})   //ERROR: clase VAR_NO_LOCAL no válida
(ETIQ, {VAR_LOCAL,4},,-)
                //ERROR: el cuarteto ETIQ espera una etiqueta, no una variable
(PARAM,{ET,"func1"}, , )
             //ERROR: el argumento del cuarteto PARAM no puede ser de clase ET
(ASIG, {CTE_ENT, 1},  , {VAR_LOCAL, "cinco"})
               //ERROR: el valor de la clase VAR_LOCAL no puede ser una cadena
(CALL_FUN, {ET, "etiq0"}, ,)
                              //ERROR: el último argumento no puede estar vacío
(RETURN_ENT,,, {VAR_LOCAL,-})
                     //ERROR: no se puede dejar el valor de un argumento vacío
(PRINT_ENT,  ,  ,  {VAR_LOCAL,  1,  4}  )
       //ERROR: la tupla del argumento de clase VAR_LOCAL solo tiene un valor
(GOTO_May, {VAR_LOCAL , 2 }, {VAR_LOCAL, 3}, {ET, "etiq0"}, {VAR_LOCAL , 8})
                                    //ERROR: un cuarteto debe tener solo 4 campos
(PRINT_ENT,  ,  , {CTE_CAD, "hola"} )
       //ERROR: el cuarteto PRINT_ENT no puede tener un argumento de la clase
CTE_CAD
(PRINT,  ,    , {CTE_ENT, -0})           //ERROR: no existe el cuarteto PRINT
(ASIG_PTR_CAD,{CTE_CAD,"5"},,{par_ref,4})
                                //ERROR: el primer argumento no puede ser cadena
(Asig, {CTE_Ent, 33333},, {VAR_LOCAL , 5})
                               //ERROR: el valor está fuera del rango permitido
(PRINT_CAD,  ,    , {CTE_CAD, hola}  )
                                 //ERROR: la cadena hola debe ir entre comillas
(suma, {VAR_GLOBAL , -22} , { VAR_TEMP, +2},
                                    //ERROR: comentario en medio de un cuarteto
       {VAR_LOCAL , -1} )   //ERROR: un cuarteto no puede estar en dos líneas
(PARAM, {Cte_Cad, "proc1}, , )             //ERROR: la cadena no está cerrada
(ASIG,{PAR,-9},--,{Par,9})
               //ERROR: un campo vacío se deja en blanco o con un único guion
(PRINT-ENT,,,{VAR_LOCAL,1})
               //ERROR: el nombre de operador solo admite letras, dígitos y _
(PRINT_ENT,,,{VAR+LOCAL,1})
                 //ERROR: el nombre de clase solo admite letras, dígitos y _
                                    //ERROR: no está declarada la etiqueta “main”
```

8.2 Translation Template for Quadruples

This section presents the complete set of translation templates used by the code generator to convert each supported quadruple into assembly instructions.

It is important to clarify that the labels shown are merely illustrative examples. In the actual implementation, the quadruples compiler relies on a unique label generator, which ensures that no label is repeated, even when the same operation appears multiple times. Descriptive names have been used here solely for clarity.

Additionally, the error messages shown in some templates are not embedded directly within the generated assembly code. Instead, the object code generator

places these messages in a separate memory area, which is accessed through appropriate pseudo-instructions during program execution. This design avoids unintended execution of static data and ensures a clean separation between code and message definitions.

(PRINT_CAD, -, -, result)
```
WRSTR result
```

(PRINT_CAR, -, -, result)
```
WRCHAR result
```

(PRINT_ENT, -, -, result)
```
WRINT result
```

(INPUT_CAD, -, -, result)
```
INSTR result
```

(INPUT_ENT, -, -, result)
```
ININT result
```

(INPUT_CAR, -, -, result)
```
INCHAR result
```

(SUMA, arg1, arg2, result)
```
ADD arg1, arg2
MOVE .A, result
```

(RESTA, arg1, arg2, result)
```
SUB arg1, arg2
MOVE .A, result
```

(MUL, arg1, arg2, result)
```
MUL arg1, arg2
MOVE .A, result
```

(DIV, arg1, arg2, result)
```
DIV arg1, arg2
MOVE .A, result
```

(MOD, arg1, arg2, result)
```
MOD arg1, arg2
MOVE .A, result
```

(EXP, arg1, arg2, result)
```
; Required error messages:
errOverflow: DATA "ERROR: Desbordamiento al calcular la potencia\n"
errDivZero: DATA "ERROR: División por cero\n"
MOVE arg1, .R1
MOVE arg2, .R2
MOVE .R2, .R3
CMP .R3, #0
BP /label0
NEG .R3
label0:
```

```
MOVE #1, .R0
CMP .R3, #0
BZ /ResExp
LoopExp:
  MUL .R0, .R1
  BV /overflow
  MOVE .A, .R0
  DEC .R3
  CMP .R3, #0
  BNZ /LoopExp
ResExp:
CMP .R2, #0
BP /expPositive
CMP #0, .R0
BZ /divZero
DIV #1, .R0
MOVE .A, .R0
expPositive:
MOVE .R0, result
BR /finish
overflow:
WRSTR /errOverflow
HALT
divZero:
WRSTR /errDivZero
HALT
finish:
```

(AND, arg1, arg2, result)

```
AND arg1, arg2
MOVE .A, result
```

(OR, arg1, arg2, result)

```
OR arg1, arg2
MOVE .A, result
```

(CONCAT, arg1, arg2, result)

```
; Required error message:
errStringSize: DATA " La cadena resultante supera el tamaño máximo
permitido\n"
MOVE #maxSize, .R4
MOVE #0, .R3
MOVE arg1, .R1
MOVE arg2, .R2
MOVE result, .R0
DEC .R1
DEC .R2
DEC .R0
Loop1:
  INC .R1
  INC .R0
  CMP [.R1], #0
  BZ /decLabel
  INC .R3
```

```
  CMP .R3, .R4
  BZ /error
  MOVE [.R1], [.R0]
  BR /Loop1
decLabel:
DEC .R0
INC .R4
Loop2:
  INC .R2
  INC .R0
  INC .R3
  CMP .R3, .R4
  BZ /error
  MOVE [.R2], [.R0]
  CMP [.R2], #0
  BNZ /Loop2
BR /finish
error:
WRSTR /errStringSize
HALT
finish:
```

(MENOS, arg1, -, result)

```
MOVE arg1, result
NEG result
```

(NOT, arg1, -, result)

```
CMP arg1, #0
BZ $5
MOVE #0, result
BR $3
MOVE #1, result
```

(ASIG, arg1, -, result)

```
MOVE arg1, result
```

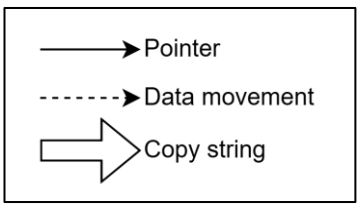

Figure 32 Legend of arrow types used in the quadruples operation diagrams

(ASIG_CAD, arg1, -, result)

```
MOVE arg1, .R0
MOVE result, .R1
DEC .R0
DEC .R1
Loop:
  INC .R0
  INC .R1
```

```
  MOVE [.R0], [.R1]
  CMP [.R0], #0
  BNZ /Loop
```

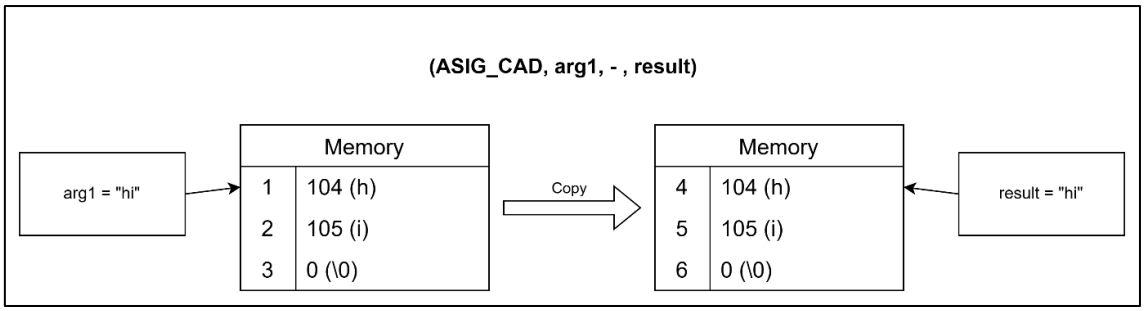

Figure 33 Representation of the ASIG_CAD quadruple operation

(PTR_ASIG, arg1, -, result)

```
MOVE result, .R0
MOVE arg1, [.R0]
```

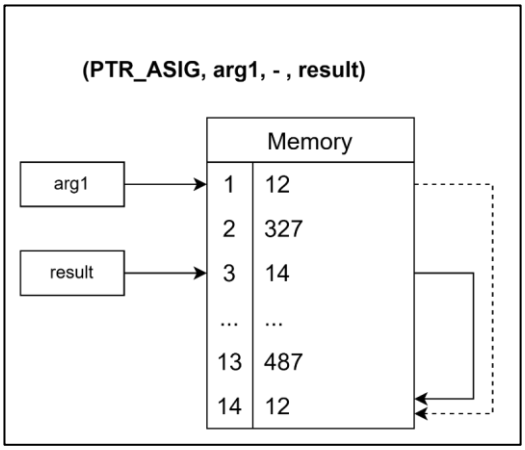

Figure 34 Representation of the PTR_ASIG quadruple operation

(PTR_ASIG_CAD, arg1, -, result)

```
MOVE arg1, .R0
MOVE result, .R1
DEC .R0
DEC .R1
Loop:
  INC .R0
  INC .R1
  MOVE [.R0], [.R1]
  CMP [.R0], #0
  BNZ /Loop
```

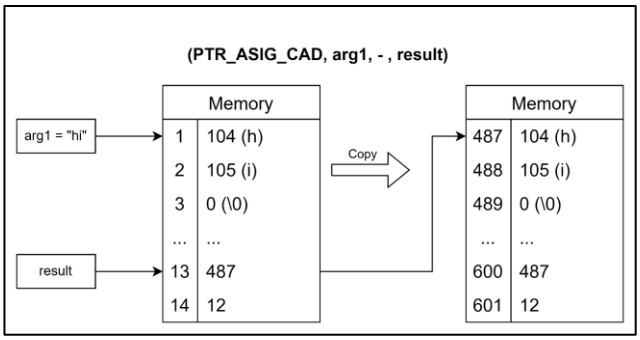

Figure 35 Representation of the PTR_ASIG_CAD quadruple operation

(ASIG_DIR, arg1, -, result)

```
MOVE arg1, result
```

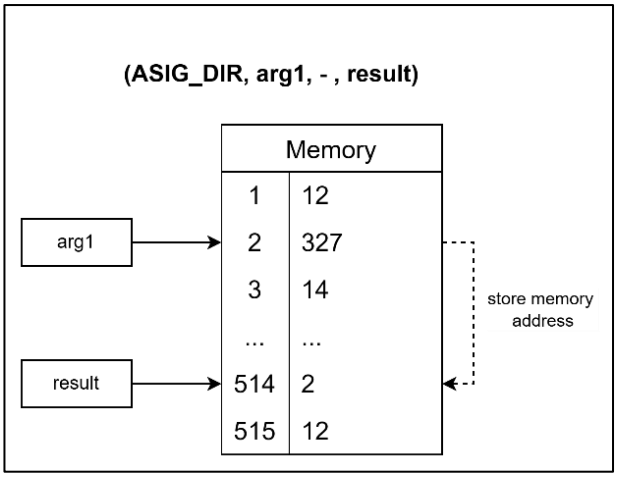

Figure 36 Representation of the ASIG_DIR quadruple operation

(ASIG_PTR, arg1, -, result)

```
MOVE arg1, .R0
MOVE [.R0], result
```

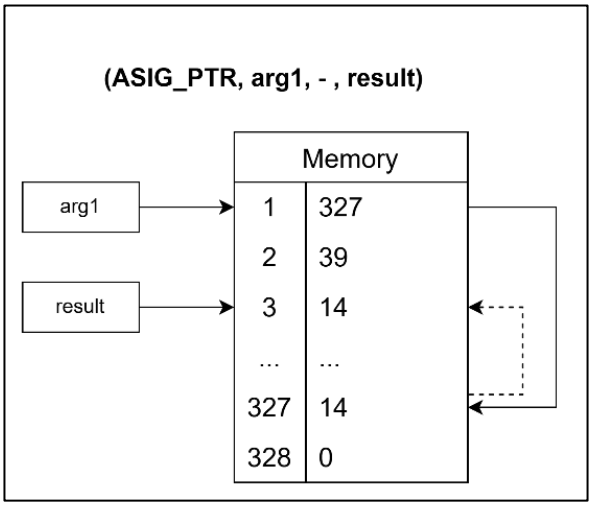

Figure 37 Representation of the ASIG_PTR quadruple operation

(ASIG_PTR_CAD, arg1, -, result)

```
MOVE arg1, .R0
MOVE result, .R1
DEC .R0
DEC .R1
Loop:
  INC .R0
  INC .R1
  MOVE [.R0], [.R1]
  CMP [.R0], #0
  BNZ /Loop
```

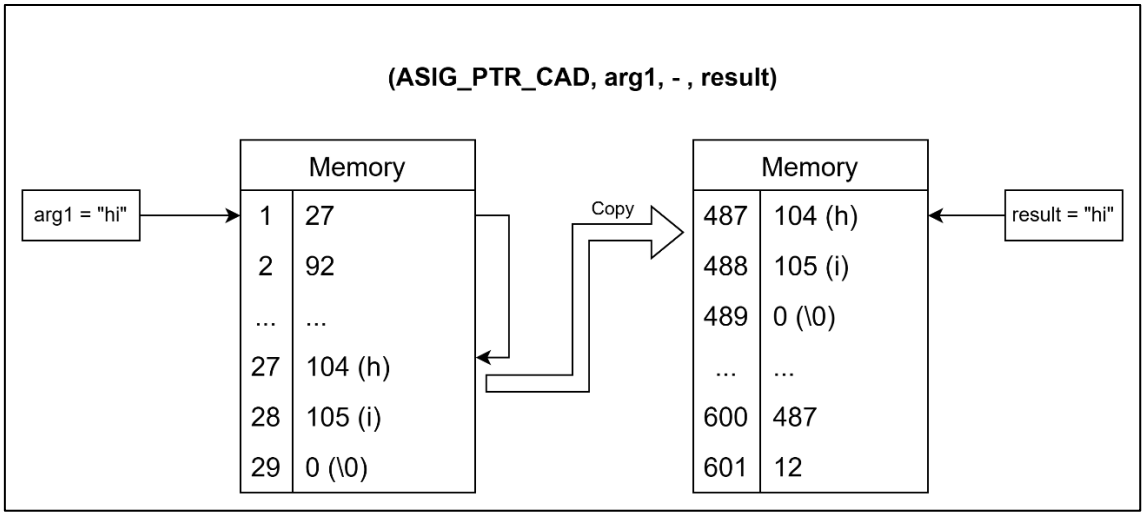

Figure 38 Representation of the ASIG_PTR_CAD quadruple operation

(GOTO, -, -, label)

```
BR /label
```

(GOTO_IG, arg1, arg2, label)

```
CMP arg1, arg2
BZ /label
```

(GOTO_DIST, arg1, arg2, label)

```
CMP arg1, arg2
BNZ /label
```

(GOTO_MAY, arg1, arg2, label)

```
CMP arg2, arg1
BN /label
```

(GOTO_MEN, arg1, arg2, label)

```
CMP arg1, arg2
BN /label
```

(GOTO_MAY_IG, arg1, arg2, label)

```
CMP arg1, arg2
BP /label
```

(GOTO_MEN_IG, arg1, arg2, label)

```
CMP arg2, arg1
BP /label
```