Universidad Nacional Autónoma de México

Ingeniería en Computación

Compilers

Compiler Report

Students:

320021312

320112179

320184624

320565661

423093137

Group: 5 Semester: 2026-I

Mexico City, Mexico. December 2025

Contents

1 Introduction **2**

2 Theoretical Framework **3**

2.1 Syntax Analysis 3

2.2 Intermediate Code Generation 4

2.3 Three-Address Code (TAC) and Quadruples 4

2.4 Syntax-Directed Translation (SDT) 5

2.5 PLY (Python Lex-Yacc) 5

3 Development **5**

3.1 Grammar Definition and Precedence 6

3.2 Syntax-Directed Translation (SDT) for Code Gen 6

3.2.1 Temporary Variable Management 7

3.2.2 Semantic Rules and Quadruple Emission 7

3.3 Architecture and Components 8

3.4 Error Handling and Validation 9

4 Results **10**

4.1 Project Structure and Source File 10

4.2 Type Checking of Declarations 11

4.3 Division-by-Zero Detection 12

4.4 Type Checking in Assignments 12

4.5 Use of an Undeclared Variable 13

4.6 Invalid Declaration Syntax with a Numeric Token 13

4.7 Missing Semicolon at the End of a Declaration 14

4.8 Successful Execution and Output 14

4.9 Correct Execution After Commenting Out a Statement 15

5 Conclusions **16**

References **18**

1 Introduction

The compilation process is fundamentally a transformation pipeline. As established in compiler theory, this pipeline is divided into two primary phases: the *front-end* (Analysis), which validates the syntactic and semantic integrity of the source code, and the *back-end* (Synthesis), which is responsible for constructing a target representation suitable for execution [1]. In previous stages of this project, the front-end was completed with a working lexer, parser, and Syntax-Directed Translation (SDT) scheme capable of checking types and basic semantic constraints. However, despite having a correct analysis phase, the system still lacked a concrete representation of the program that could be executed, optimized, or translated to a specific architecture.

This limitation reveals a clear practical need: once the source code has been validated, the compiler must produce a structured and machine-independent representation of the program that captures its behavior in a sequential form. Parse trees are inherently hierarchical and tied to the grammar, which makes them unsuitable as a direct execution model. Real machines, virtual machines, and optimizers operate over linear sequences of instructions. Therefore, the central challenge addressed in this phase is the systematic transformation of a nested syntactic structure into a linear sequence of operations that preserves operator precedence, associativity, and type correctness.

The primary objective of this final phase of the project is to implement the *Code Generation* module for the compiler hosted in the group repository. Building upon the previously developed SDT scheme and using the Python Lex-Yacc (PLY) library, the compiler is extended so that semantic actions do not merely validate types, but actively emit Intermediate Code in the form of Three-Address Code (TAC) represented as Quadruples [2]. In this design, each instruction is decomposed into an operator and up to two operands, producing a result that may be stored in a variable or a temporary name.

Choosing to generate Intermediate Code instead of machine-specific assembly addresses two complementary needs. First, it provides a machine-independent abstraction layer that allows the same front-end to be reused for different target architectures by replacing only the final translation stage. Second, it creates a clean and explicit structure on top of which future optimization phases can be implemented, such as common subexpression elimination or constant folding [3], [4]. In this sense, the module produced in this report acts as a bridge between the abstract syntax level and the eventual low-level implementation.

This report details how that need is covered. Section 2 reviews the theoretical foundations of Intermediate Code, Three-Address Code, SDT, and the PLY framework. Section 3 describes the Grammar adaptation for code emission, the SDT attributes and actions used to generate Quadruples, and the modular architecture of the implementation. Section 4 presents representative execution results, showing how declarations, assignments, and arithmetic expressions are translated, checked, and executed by the compiler. Finally, Section 5 summarizes the main conclusions and outlines possible extensions such as control-flow constructs and later optimization stages.

2 Theoretical Framework

The development of a compiler is a multi-phase process that typically includes lexical analysis, syntax analysis, semantic analysis, Intermediate Code Generation, optimization, and final code generation. While the earlier deliverables of this project focused on analysis (breaking down and validating the source code), the current phase belongs to the synthesis part of the pipeline, where a concrete program representation is produced from the validated input.

2.1 Syntax Analysis

As established in previous reports, syntax analysis verifies that the token stream generated by the lexer conforms to a Context-Free Grammar (CFG). It constructs a hierarchical parse tree that represents the syntactic structure of the code and enforces the language's grammatical rules. Parser construction methods can be broadly divided into top-down and bottom-up strategies; in this project, a bottom-up approach (LALR(1)) is implicitly used through PLY's parsing engine, which internally builds parse tables from the production rules.

The parse tree is an essential intermediate structure for reasoning about the correctness of the program, but it is not the final representation used for execution. Instead, it serves as the backbone over which semantic rules are attached, allowing the compiler to propagate attributes (such as types or storage locations) from the leaves to the root.

2.2 Intermediate Code Generation

Intermediate code is a representation of the source program that bridges the gap between the high-level language and the low-level machine code. According to compiler theory, intermediate code is machine-independent, which facilitates retargeting the compiler to different architectures without rewriting the entire front-end [2]. It also allows optimization techniques to be applied more easily before the final machine code is generated, since transformations can be expressed over an abstract instruction set instead of concrete opcodes.

A key requirement for an effective intermediate representation is that it should be both close enough to machine code to expose control flow and data dependencies, and high-level enough to remain independent of specific registers or calling conventions. Three-Address Code (TAC) satisfies this property by decomposing complex expressions into small, regular steps.

2.3 Three-Address Code (TAC) and Quadruples

Three-Address Code (TAC) is a sequence of instructions where each instruction has at most three operands and usually conforms to the pattern

$$\text{result} = \text{operand}_1 \; op \; \text{operand}_2,$$

or to a related form such as a unary operation or a simple assignment [3]. A common data structure to implement TAC is the *Quadruple*, which consists of four fields:

- **Operator (Op):** The operation to be performed (e.g., $+$, $-$, $*$, $/$, $=$).
- **Argument 1 (Arg1):** The first operand (variable or constant).
- **Argument 2 (Arg2):** The second operand (if applicable).
- **Result (Res):** The location where the result is stored. In complex expressions, this is usually a temporary variable generated by the compiler [4].

By decomposing expressions into a list of Quadruples, the compiler obtains a linear representation that preserves the semantics of the original program while making data flow explicit. This explicitness simplifies later phases such as liveness analysis or register allocation.

2.4 Syntax-Directed Translation (SDT)

Syntax-Directed Translation is a method where semantic actions are embedded within the grammar productions. These actions are executed during the parsing process to generate the intermediate code. For example, when an arithmetic expression is reduced, an SDT action can create a new temporary variable to store the result and emit the corresponding Quadruple [5]. Depending on the position of the actions, SDT schemes are often classified as S-attributed (only synthesized attributes) or L-attributed (with inherited attributes).

In this project, an S-attributed style is adopted: each nonterminal is associated with synthesized attributes like `.addr` and `.type`, which are computed from its children. As a consequence, once the parser finishes reducing a production, all the information needed to generate the corresponding Quadruples is already available at that point in the parse.

2.5 PLY (Python Lex-Yacc)

The implementation utilizes PLY, a pure Python implementation of the classic lex and yacc tools. Unlike other tools that require separate input files, PLY relies on Python reflection to build the lexer and parser directly from the Python code, using naming conventions on functions and docstrings that contain the grammar rules [6]. This allows for seamless integration of the SDT logic within Python functions.

PLY also automatically builds the LALR(1) parse tables and can generate diagnostic files (`parsetab.py`, `parser.out`) that help inspect conflicts and the structure of the grammar. By combining token definitions, production rules, and semantic actions in a single language (Python), the project achieves a tightly integrated front-end and intermediate code generator.

3 Development

This section details the internal construction of the Code Generator, focusing on the adaptation of the grammar for linear code emission, the implementation of the SDT scheme, and the architecture of the PLY-based system.

3.1 Grammar Definition and Precedence

The grammar implemented in the `compiler` directory is designed not only to validate syntax but to enforce operator precedence and associativity, which is critical for generating correct arithmetic instructions. The core fragment in charge of expressions and declarations can be summarized as:

$$S \rightarrow D \mid P$$

$$D \rightarrow T\ id = E; \quad \text{(Declaration \& Assignment)}$$

$$E \rightarrow E + M \mid E - M \mid M$$

$$M \rightarrow M * F \mid M/F \mid F$$

$$F \rightarrow \text{number} \mid \text{decimal} \mid \text{id} \mid (E)$$

Hierarchical Analysis. The grammar is stratified into three levels to handle precedence automatically during the bottom-up parsing:

- **Level F (Factors):** Handles high-priority units like parentheses, identifiers, and literal numbers.
- **Level M (Terms):** Handles multiplication and division, ensuring they bind tighter than addition and subtraction.
- **Level E (Expressions):** Handles the lowest-priority operations (addition and subtraction).

This structure ensures that an expression like 5+2∗3 is parsed as 5+(2∗3), allowing the code generator to emit the multiplication instruction before the addition without complex reordering logic. The precedence is therefore enforced by the grammar itself and not by ad-hoc post-processing.

3.2 Syntax-Directed Translation (SDT) for Code Gen

The core of the implementation is the Syntax-Directed Translation scheme. The design uses *Synthesized Attributes*, where the semantic value of a parent node is derived from its children. For code generation, a specific attribute `.addr` (address) is introduced. This attribute holds the location of the value computed by a subexpression, which can be:

1. A memory address of a variable (e.g., `id.entry`).
2. A raw constant value (e.g., `5` or `3.14`).
3. A temporary variable generated by the compiler (e.g., `t_1, t_2`).

3.2.1 Temporary Variable Management

To flatten the hierarchical tree into linear code, the system generates temporary variables. A global counter ensures each new intermediate result gets a unique name (`t_1, t_2, ...`). The function `new_temp()` is called whenever an arithmetic operation is reduced. This systematic generation of temporary names guarantees that every intermediate result has a stable identifier that can be reused by later instructions and analysis phases.

3.2.2 Semantic Rules and Quadruple Emission

The following pseudocode illustrates the logic embedded in the Python functions. The function `emit()` appends a new tuple $(Op, Arg1, Arg2, Res)$ to the global list of Quadruples.

Rule for Addition ($E \rightarrow E_1 + M$):

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    # 1. Generate a new temporary variable holder
    t_new = new_temp()

    # 2. Extract addresses from child nodes
    arg1 = p[1].addr  # Address of E1
    arg2 = p[3].addr  # Address of M

    # 3. Emit the Quadruple
    emit('+', arg1, arg2, t_new)

    # 4. Synthesize the attribute up to the parent
    p[0].addr = t_new
```

Rule for Assignment ($D \rightarrow T\ id = E;$):

```
def p_declaration_assign(p):
    'declaration : type ID EQUALS expression SEMICOLON'
    # 1. Semantic Check: Type compatibility
    if not check_type(p[1], p[4].type):
        raise SemanticError("Type Mismatch")

    # 2. Emit Assignment Quadruple
    # Note: Arg2 is None for simple assignments
    emit('=', p[4].addr, None, p[2])

    # 3. Update Symbol Table
    symbol_table.add(p[2], p[1], p[4].addr)
```

In this way, every successful reduction of a production not only propagates attributes but also contributes one or more Quadruples to the intermediate representation. If a semantic error is detected (for instance, an invalid type combination), the reduction is interrupted, and no Quadruple is emitted.

3.3 Architecture and Components

The system relies on the `PLY` (Python Lex-Yacc) library [6]. The architecture is modular, separating tokenization from parsing and code generation logic:

- `lexer.py` **(Lexical Analyzer):** Scans the input stream and produces tokens. It handles line tracking and error reporting for illegal characters, and feeds the token stream into the parser.
- `parser.py` **(Syntax Analyzer & Code Generator):** This is the central component. It performs two simultaneous tasks:
    1. **Validation:** Verifying the token stream against the grammar rules using PLY's parsing engine.
    2. **Translation:** Executing the Python code associated with each production to generate the list of Quadruples, while consulting and updating the Symbol Table.
- `main.py` **(Driver):** Initializes the pipeline. It reads source code from the standard input or a file, invokes the lexer and parser, and finally formats and prints

the generated Quadruples or directly executes them, producing the observable output.

To provide a high-level view of how these components interact, Figure 1 shows the data flow between the front-end and the Code Generation module.

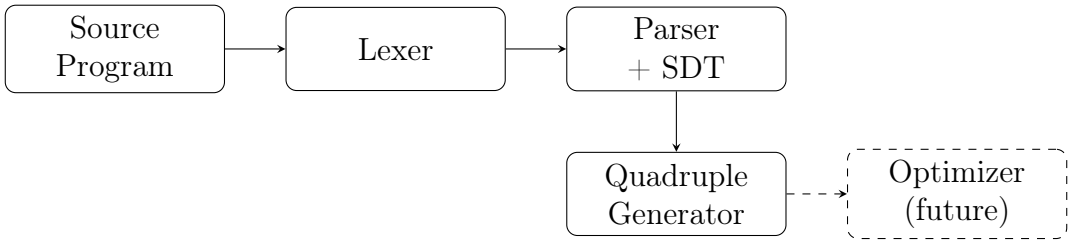

Figure 1: Compilation pipeline with integrated Syntax-Directed Translation and code generation.

3.4 Error Handling and Validation

Robust error handling is essential for a reliable compiler. The system implements a fail-fast strategy for semantic errors during code generation, while relying on PLY's default mechanisms for syntax error reporting.

- **Syntactic Errors:** Handled by PLY's `p_error` function. If the input violates the grammar, the parser reports the offending token and either attempts local recovery or aborts, depending on the configuration.
- **Semantic Errors (Type Mismatch and Division by Zero):** Before generating an assignment Quadruple, the SDT logic checks whether the expression type matches the variable declaration (for instance, preventing the assignment of a `float` expression to an `int` variable without conversion). Additionally, during the construction of Quadruples for division, the compiler detects divisions by zero at compile time whenever the denominator is a constant or an already evaluated value.
- **Result:** If a semantic error is detected, the `emit()` function is not invoked for that production, and the error is reported immediately with an informative message that includes the line number. This prevents the generation of invalid or inconsistent intermediate code and enforces semantic safety at compile time.

4 Results

This section illustrates, step by step, how the implemented compiler processes a small program stored in the file `ejemplo.src`. The figures show the source code inside the editor, the project structure, the invocation of the compiler from the terminal, and the different kinds of feedback produced: successful execution, type errors, and division-by-zero detection.

4.1 Project Structure and Source File

Figure 2 shows the organization of the files inside the `compiler` package. The modules `lexer.py`, `parser.py`, and `ast_nodes.py` implement the front-end of the compiler, whereas `compiler.py` acts as the driver that orchestrates the invocation of the lexer, the parser, and the Code Generation phase. The file `ejemplo.src` is a plain-text source program written in the small language defined by the grammar.

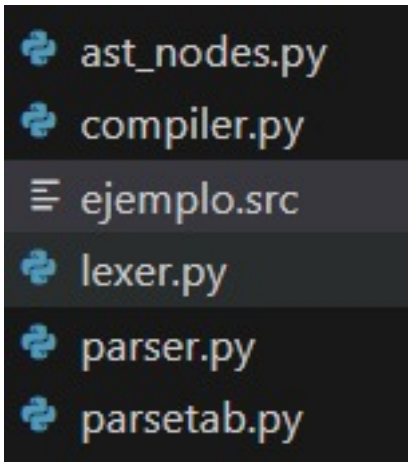

Figure 2: Project layout inside the `compiler` directory, highlighting the source file `ejemplo.src` and the Python modules that implement the lexer, parser, and compiler driver.

Figure 3 presents an initial version of the program `ejemplo.src` and the command used to invoke the compiler. The code declares an integer variable `a`, a floating-point variable `b` that depends on `a`, and then updates `a` with an arithmetic expression. In the terminal, the program is compiled by executing `python -m compiler ejemplo.src`, which launches the driver module and triggers the complete compilation pipeline for the specified input file.

Figure 3: Editing the source file `ejemplo.src` in the IDE and invoking the compiler from the terminal with the command `python -m compiler ejemplo.src`.

4.2 Type Checking of Declarations

One of the main responsibilities of the compiler is to enforce type safety. Figure 4 shows a program where the declaration of `a` uses a floating-point literal (`3.3`) even though the variable is declared as `int`. After reading the file and performing lexical and syntactic analysis, the SDT rules consult the Symbol Table and detect that the literal is incompatible with the declared type.

The error message printed in the terminal clearly indicates the problem: `error in line 1: cannot assign float to int variable 'a'`. At this point, the Code Generation phase is aborted for that statement, and no Quadruples are emitted for the invalid assignment, ensuring that the intermediate representation remains consistent with the language rules.

Figure 4: Detection of a type error in the declaration: the compiler identifies that the value `3.3` is a `float` and cannot be assigned to the integer variable `a`, reporting the error on line 1.

4.3 Division-by-Zero Detection

Figure 5 illustrates another kind of semantic validation: division by zero. In this version of the program, the variable `b` is defined as `float b = a / 0;`. During the SDT evaluation of the expression, the compiler recognizes that the denominator is a constant equal to zero and flags the operation as undefined.

The terminal output reports `error in line 2: division by zero`. As before, the compiler stops the Code Generation process for the offending statement and prevents the execution of any intermediate code that would perform an invalid arithmetic operation.

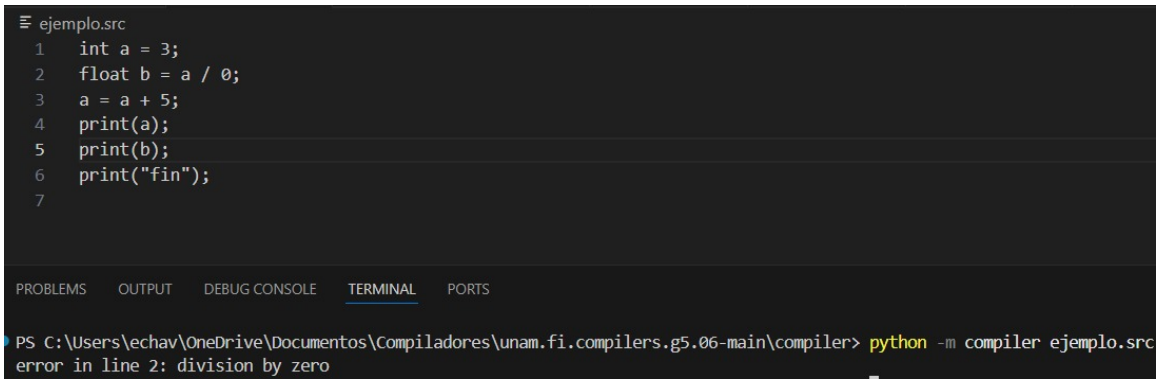

Figure 5: Semantic validation of arithmetic expressions: the compiler detects a division by zero in the initialization of `b` and reports an error on line 2 before generating intermediate code.

4.4 Type Checking in Assignments

Type safety is also enforced in subsequent assignments. Figure 6 shows a scenario where the declaration of `a` is correct, but the later update `a = a + 5.2;` produces a `float` result that is again incompatible with the `int` type of `a`. The compiler reuses the same SDT infrastructure used in declarations: the expression is evaluated, its resulting type is inferred, and then compared against the variable's declared type in the Symbol Table.

The mismatch is reported as `error in line 3: cannot assign float to int variable 'a'`. This example confirms that the semantic checks are not limited to initial declarations but apply to all assignments that could break the type discipline of the language.

Figure 6: Run-time of the compiler when a later assignment produces a floating-point value that is incompatible with the integer variable `a`; the compiler reports a type mismatch on line 3 and stops the generation of invalid code.

4.5 Use of an Undeclared Variable

In this test, the source file `ejemplo.src` contains the statement `float b = a / 2.0;` on line 1, where the variable `a` has not been declared beforehand. During semantic analysis, the compiler consults the symbol table and fails to find an entry for `a`, which triggers a semantic error. As a result, the compilation process stops and the message `error in line 1: variable 'a' not declared` is printed in the terminal. This example demonstrates the compiler's ability to detect the use of undeclared identifiers and halt the generation of intermediate code when the program violates the language's semantic rules.

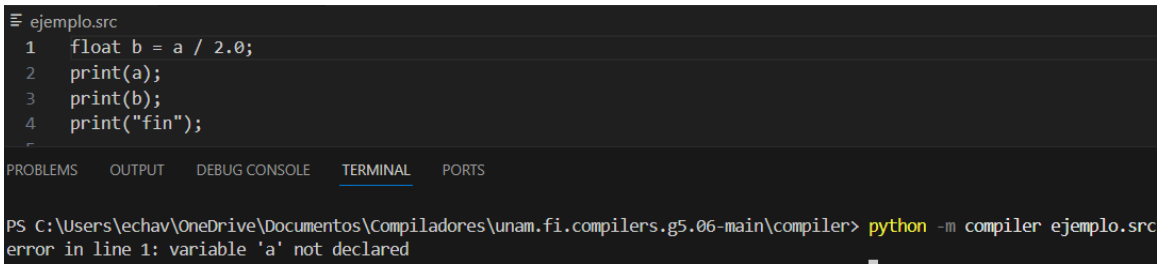

Figure 7: Detection of an undeclared variable: the compiler reports that `a` has not been declared before its use in the initialization of `b`.

4.6 Invalid Declaration Syntax with a Numeric Token

In this test, the program includes an invalid declaration of the form `int 122;` in line 1 of `ejemplo.src`. According to the grammar, an identifier must appear after the `int` keyword, but the lexer produces a `NUMBER` token instead. When the parser attempts to reduce the declaration, it encounters an unexpected token and invokes the generic syntax error handler. The compiler then reports the message `error`

`in line 1: syntax error at token 'NUMBER' with value 122`, clearly indicating that the declaration is malformed. This outcome shows that the parser effectively detects syntactic violations and identifies the precise token that caused the failure.

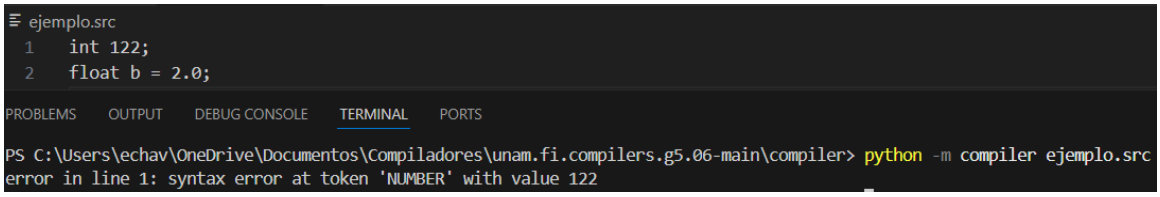

Figure 8: Invalid declaration using a numeric literal after the type keyword; the parser reports a syntax error at token `NUMBER` with value `122`.

4.7 Missing Semicolon at the End of a Declaration

Another test evaluates the compiler's ability to detect missing punctuation. In this case, the program contains the statement `int a = 3` on line 1 without the required terminating semicolon. The following line starts with a valid declaration, which makes the parser aware of a mismatch between the expected grammar structure and the actual token stream. The error is interpreted as a missing semicolon at the end of the previous declaration, and the compiler reports a syntax error on line 1 with the message `syntax error (missing ';')`. This example illustrates how the error-handling logic provides user-friendly messages for common syntactic mistakes.

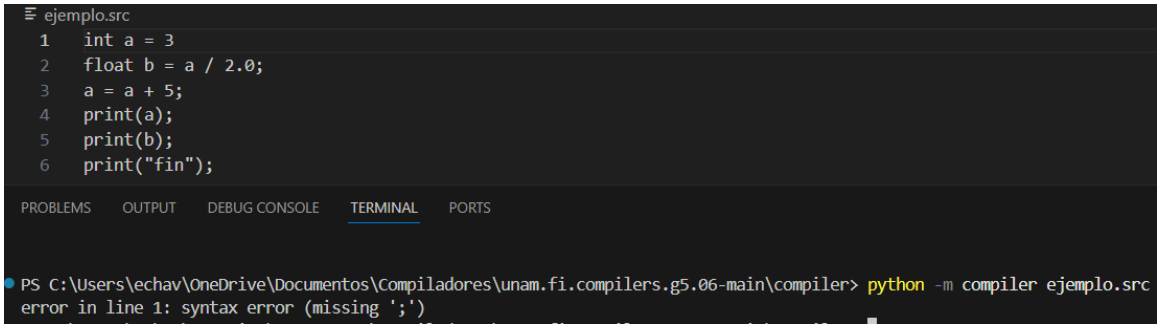

Figure 9: Detection of a missing semicolon: the declaration `int a = 3` lacks the terminating `';'` and the compiler reports a syntax error on line 1.

4.8 Successful Execution and Output

Figure 10 shows a consistent version of the program where all declarations and assignments respect the typing rules and no arithmetic errors occur. The variable `a` is initialized with the integer value `3`, the variable `b` is defined as `a / 2.0` (a valid `float` expression), and the update `a = a + 5;` preserves the integer type.

In this case, the compiler completes the entire pipeline: it performs lexical and syntactic analysis, checks the semantics, generates the corresponding Quadruples, and then executes them. The observable effect is the printing of the final values: `8` for `a`, `1.5` for `b`, and the string `"fin"`. The output confirms that the intermediate code correctly represents the intended computation and that the Code Generation phase integrates seamlessly with the interpreter embedded in the driver.

Figure 10: Successful compilation and execution of a well-typed program: the compiler generates and runs the intermediate code, printing the final values of `a` and `b` and the message `"fin"`.

4.9 Correct Execution After Commenting Out a Statement

In this version of `ejemplo.src`, the statement `print(a);` is turned into a comment using the prefix `//`, which removes it from the token stream during lexical analysis. The remaining declarations and assignments are syntactically and semantically correct, so the compiler successfully generates intermediate code, executes it, and prints the final values of `b` and the string `"fin"`. The terminal output confirms that the comment mechanism works as expected and that the compilation pipeline behaves correctly once the source program is consistent.

Figure 11: Successful compilation and execution after commenting out a statement: the line `//print(a);` is ignored by the lexer, and the program prints the value of `b` and the message `"fin"`.

Overall, the scenarios demonstrate that the compiler is capable of reading source files from disk, validating their syntax and semantics, generating intermediate code for correct programs, and rejecting programs that violate type rules or contain unsafe arithmetic operations.

5 Conclusions

The work presented in this report addresses a concrete need in the compiler previously developed by the team: the absence of a structured, machine-independent representation of the validated source code. By extending the existing Syntax-Directed Translation scheme and integrating it into the PLY-based parser, the project successfully implements an Intermediate Code Generation phase that transforms declarations and arithmetic expressions into a linear sequence of Quadruples (Three-Address Code). Through the use of synthesized attributes, an explicit management of temporary variables, and a grammar that encodes operator precedence, the compiler is able to bridge the gap between a hierarchical parse tree and a sequential execution model while preserving semantic correctness and type safety.

At the same time, the adoption of TAC and Quadruples lays the foundation for future work in optimization and final code emission. The representation produced in this phase is independent of any specific machine architecture, which makes it suitable as an input for later stages such as control-flow analysis, register allocation, or translation to assembly for platforms like x86 or MIPS. The modular architecture that separates the lexer, parser, and Code Generation components also facilitates maintenance, testing, and extension of the compiler. In this sense, the current implementation not only validates the theoretical concepts seen in class but also provides

a functional and extensible tool that can be expanded with new language constructs and more advanced optimization techniques.

References

[1] GeeksforGeeks, *Synthesis phase in compiler design*, Accessed: 2025-11-28, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/synthesis-phase-in-compiler-design/.

[2] GeeksforGeeks, *Need for intermediate code and code optimization*, Accessed: 2025-11-28, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/need-for-intermediate-code-and-code-optimization/.

[3] GeeksforGeeks, *Types of three-address codes*, Accessed: 2025-11-28, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/types-of-three-address-codes/.

[4] GeeksforGeeks, *Issues in the design of a code generator*, Accessed: 2025-11-28, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/issues-in-the-design-of-a-code-generator/.

[5] GeeksforGeeks, *Syntax-directed translation schemes*, Accessed: 2025-11-28, 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/syntax-directed-translation-schemes/.

[6] D. Beazley, *Ply (python lex-yacc) documentation*, version 3.11, 2011. [Online]. Available: http://www.dabeaz.com/ply/ply.html.