DEMOCRATIC AND POPULAR ALGERIAN REPUBLIC
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

Abderrahmane Mira University – Béjaïa

Faculty of Exact Sciences
Department of Computer Science

Project Report

Report Title

- Mini Python Compiler -

Supervised by:
Mrs. N. TASSOULT
(Instructor)
Mr. Nabil DJEBARI
(Instructor)

Prepared by:
Yacine MADANI

Presented as part of the **Compilation Module Project**

Level: **Third Year Computer Science – Licence (LMD System)**

Specialization: **Computer Systems**

Academic Year: 2025–2026

Mini-Python-Compiler

Yacine Madani

November 2025

Introduction

As part of the Compiler Design module, I developed a mini Python compiler with the objective of applying and demonstrating the core principles studied throughout the course. The development of this project took place during the entire month of November 2025 and continued through the first week of December 2025, before being officially submitted on December 8, 2025. This timeline allowed for a structured and progressive implementation of the different components of the compiler.

The project was intentionally designed to remain simple and focused, in order to highlight the fundamental stages of the compilation process. It implements two essential phases: lexical analysis and syntax analysis. The lexical analyzer is responsible for breaking down the source code into tokens, while the syntax analyzer verifies the structure of the code according to predefined grammar rules. Although the compiler does not perform semantic analysis or code generation, it effectively demonstrates the core mechanics required for building more advanced compilation systems.

A key feature of this mini compiler is its built-in error-handling system, which detects, describes, and reports syntactic and lexical errors encountered during analysis. This contributes to a clearer understanding of how compilers manage incorrect input and ensures a more robust user experience.

To enhance usability, the project also includes a graphical interface that allows users to write, load, and test Python-like code more easily. This interface makes the tool accessible even to those who may not be familiar with command-line environments, and it provides a more intuitive way to observe the compiler's behavior.

This report presents the methodology followed throughout the project, the technical decisions made during development, and the final structure and functionalities of the compiler. It also reflects on the learning outcomes gained through the implementation of this simplified yet representative compilation system.

Compiler General Information

Name: Mini-py-Compiler

Programming Language Java

Supported Language Python

IDE Used Visual Studio Code (VS Code)

Java Version JDK 21

Operating System Ubuntu

Modules
- Lexical Analyzer
- Syntax Analyzer
- Error Handler
- Graphic Interface

Executable Yes (Jar file included)

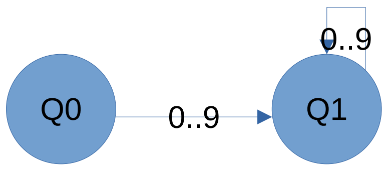
GitHub Yes (with committed changes and version tracking)

Important detail: Mini-py-Compiler can only verify syntax for some instructions only and mainly the **while loop**.
->it reads from a file located in the *executable* folder, and edits it.

Lexical Analyzer

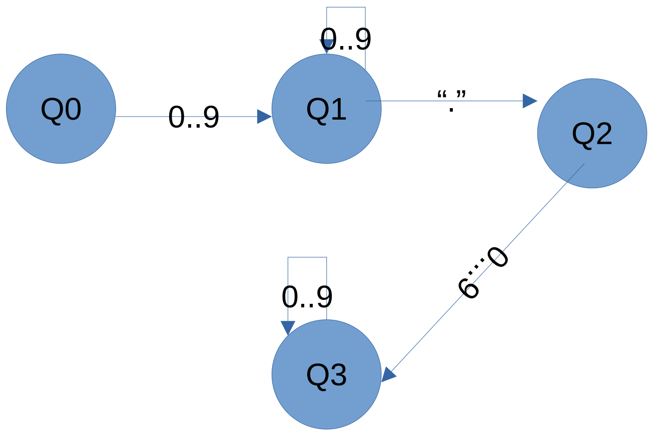
In this section, I will give the **automata way** (and the matrixes) of building my verifying methods, also *how this analyzer works briefly.* The rest of the **Keywords** and **Operators** even the **Separators** are recognized using the **matching method**.

INT TYPE



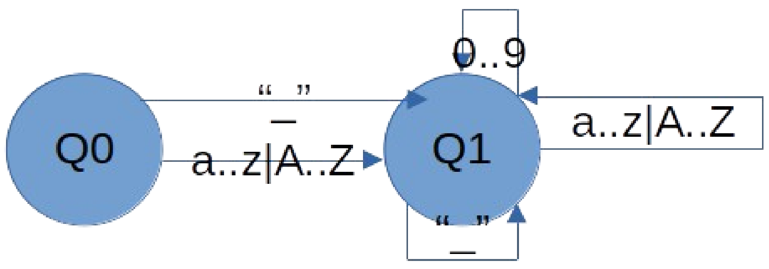
	0..9	other
Q0 (initial)	1	-1
Q1 (final)	1	-1

FLOAT TYPE



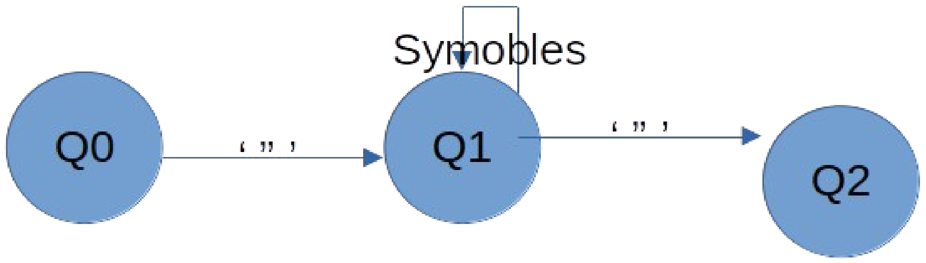
	0...9	"."	other
Q0 (initial)	1	-1	-1
Q1	1	2	-1
Q2	3	-1	-1
Q3 (final)	3	-1	-1

Identifier Unit



	a..z|A..Z	“_”	0..9	other
Q0 (initial)	1	1	-1	-1
Q1 (final)	1	1	1	-1

String TYPE



	‘ ” ’	other
Q0 (initial)	1	-1
Q1	2	1
Q2 (final)	-1	-1

Analyzer method:

To use the lexer anywhere in the code, we need to call the static method Lexical_Analyzer.Analyzer which triggers it to statrt analyzing the code situatued in the **code.py** file (in the **executable folder**).
- It skips spaces and tabulation .
- It count lines and columns.
- It adds tokens and there types, using method Tokenizer, to the Arraylist TOKENS.
- The TOKENS is what the syntax analyzer will use instead of going through the whole code again.

Please check my Github repository for clear visualization of the code.

Tokenizer method:

The lexer identifies the units it reads and their types, that's where the Tokenizer method comes in. We call it everytime the Analyzer has extracted an unit.
- It verifies the unit and gives it a token .
- It uses the automata methods for checking.
- If the token isn't valid, it states that it's and error and adds it to the TOKENS list.

Please check my Github repository for clear visualization of the code.

Syntax Analyzer

Here I used the **Recursive Descent** way to do my syntax analyzing, using the **grammar** presented below.

Grammar

Program → statementlist
Statementlist → statement statementlist | ~
Statement → assignement | whileloop | emptyline
Statement → declaration
Whileloop → “while” condition “:” newline Bloc
Bloc → INDENT bloc DEDENT
Assignement → id operator expression newline
Operator → “=”|”+=”|”-=”|”*=”|”/=”|”%=”
Expression → Term.Expprime
Expprime → “+”TermExpprime |“-”TermExpprime
Expprime → “*”factorTermExpprime
Expprime → “/”factorTermExpprime
Expprime → “%”factorTermExpprime
Term → id | number | “(”Expression”)” | “True”
Term → “False” | “None” | String
Factor → “(”Expression”)”| number | “True”
Factor → “False” | “-”factor
Declaration → def id(PARAM): newline Bloc
Declaration → class id: newline Bloc
PARAM → id.Prmprime | number.Prmprime
Prmprime → ,idprmprime| number.Prmprime | ~

Error Handler

The `ErrorHandler` class is responsible for managing and recording all errors detected during the compilation process. It plays a crucial role in ensuring that both lexical and syntax errors are properly captured and displayed in the graphical interface. The class provides three main functionalities:

1. **setLexicalErrorMessage:** This method is called whenever the lexical analyzer detects an error. It receives parameters such as `ERROR_CODE`, `token`, `line`, and `column`, and generates an error message to be displayed in the interface logs. Each call increments the total error count and stores the details in an `ArrayList<String[]>` named `ALL_ERRORS`, which is later used by the interface to present all detected errors.

2. **setSyntaxErrorMessage:** This method functions similarly to `setLexicalErrorMessage`, but is specifically used for syntax errors detected by the syntax analyzer. It also updates the error count and records the error information in the `ALL_ERRORS` list for interface display.

3. **resetCount / resetAllErrors:** These methods are called by the interface each time the user clicks the `Compile` or `Erase` button. They reset the total error count to zero and clear the `ALL_ERRORS` list, ensuring that the error display starts fresh for each new compilation attempt.

Overall, the `ErrorHandler` class centralizes error management, providing a reliable mechanism to track, store, and display errors while maintaining the interface logs in sync with the compilation process.

Folders and Files

→ Executable ---> main.jar | code.py

→ src
    → … (you have all the code here)

In order to execute the program get inside
The Executable folder directely

Conclusion

In this project, a mini Python compiler was successfully designed and implemented as part of the Compiler Design module. The development process, carried out between November 2025 and the first week of December 2025, allowed for a focused and structured implementation of the compiler's core components. By concentrating on lexical and syntax analysis, the project highlighted the fundamental stages of compilation while keeping the system simple and educational.

The compiler includes an effective error-handling mechanism, which demonstrates how lexical and syntactic errors can be detected and reported, providing a practical insight into compiler behavior. The addition of a graphical interface further improved the usability of the tool, making it easier to interact with the compiler and observe its operation in real time.

As this is the first version of the compiler, it may still contain some errors or limitations. These imperfections are expected in an initial implementation and represent opportunities for further refinement and improvement in future versions.

Overall, this project not only reinforced theoretical knowledge of compilation principles but also provided hands-on experience in software design, Java programming, and the practical challenges of implementing a compiler. It forms a solid foundation for potential future expansions, such as semantic analysis or code generation, which could transform this mini compiler into a more complete educational tool.