Universidad Distrital Francisco Jose de Caldas

Computer Science III

First Version of the Report

Juan Pablo Bustos Urueña - 20221020114

Maria Camila Restrepo Silva - 20221020044

Supervisor: Carlos Andrés Sierra

A report submitted for the first time of the project of compilers

July 10, 2025

Declaration

We, Juan Pablo Bustos, Maria Camila Restrepo Silva of the Faculty of Engineering, Universidad Distrital Francisco Jose de Caldas, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

We give consent to a copy of my report being shared with future students as an exemplar.

We give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Juan Pablo Bustos Urueña
Maria Camila Restrepo Silvas July 10, 2025

Abstract

This project presents the design and implementation of a basic educational compiler developed in Java, aimed at illustrating the fundamental phases of compilation. The compiler includes lexical analysis, syntax analysis using generative grammar, and the construction of derivation trees. Through a graphical interface built with JavaFX, users can input simple source code and visualize how it is processed step by step. The tool highlights token recognition, the application of grammar rules, and the generation of a syntax tree, offering a didactic approach to understanding how compilers operate internally.

Keywords: Educational compiler, Java, lexical analysis, generative grammar, derivation tree

Contents

Chapter 1

Introduction

1.1 Background

Many students find it difficult to understand how compilers work due to the abstract nature of formal grammars and parsing. This project proposes a basic educational compiler in Java that uses generative grammar, token analysis, and derivation trees to process simple code. Through a graphical interface, users can visualize each compilation step, making core concepts more accessible and easier to learn.

1.2 Problem statement

Many students struggle to understand how compilers work internally due to the abstract nature of concepts like lexical analysis, grammar parsing, and derivation trees. Traditional teaching methods often lack interactive tools that clearly demonstrate these processes. This project addresses that gap by developing a visual tool that shows how input code is tokenized and parsed using generative grammar, helping to clarify the compiler's core functions in a more accessible way.

1.3 Aims and objectives

Aims: The aim of this project is that the people in general that wants to learn about to programming and functionality of compilers could use this program to understand in a better much way

Objectives:

- General objective: To develop an educational compiler in Java that enables the visualization of lexical and syntactic analysis for a simple programming language, through the generation of derivation trees and the listing of applied grammar rules.
- Specifics Objectives:
  - To design a simplified programming language with a well-defined generative grammar capable of expressing basic structures such as assignments, arithmetic expressions, and conditional statements.
  - To implement a lexical analyzer that correctly identifies the language tokens, including identifiers, operators, literals, and punctuation symbols.

- To build a syntax analyzer capable of processing the sequence of tokens and generating the corresponding derivation tree by applying the defined grammar rules.
- To develop an interactive graphical user interface using JavaFX that allows users to input code and clearly visualize the resulting derivation tree

1.4 Solution expected

This project proposes the development of a simple, educational compiler implemented in Java. The solution provides a clear, visual representation of the compilation process by including a lexical analyzer, a syntax parser based on generative grammar, and the generation of derivation trees. Through a graphical interface, users can input basic code and observe how the compiler processes each line step by step. This approach bridges the gap between theory and practice, helping students grasp key concepts such as tokens, grammar rules, and parse tree construction.

Chapter 2

Literature Review

2.1 Review of the state of the art

Compiler design has been shaped by foundational concepts such as generative grammars, lexical analysis, and syntax parsing. Generative grammars, particularly context-free grammars, provide the structure for defining programming language syntax and are essential for building parsers and derivation trees. Lexical analysis breaks down input code into tokens, which are then parsed according to grammar rules to form structured representations of code.
Educational tools like grammar simulators, syntax analyzers, and automata visualizers have helped students grasp theoretical concepts in formal language and automata theory. However, many of these tools focus on static or isolated demonstrations of language recognition, often lacking full support for real input parsing, tree visualization, or semantic validation. Research has shown that graphical interfaces and dynamic visualizations can significantly enhance the understanding of complex abstract models by offering immediate feedback and interpretability of parser decisions and grammar applications.
Recent methods highlight the utilization of interactive visual settings to demonstrate code processing, striving to connect abstract theory with actual compiler functionality. Several initiatives have investigated Java-based educational compilers to enable students to follow the tokenization and parsing process via specific examples, encouraging active engagement in compiler theory instruction

2.2 Description of the project in the context of existing literature

The proposed project aims to fill the gap between theoretical learning tools and practical compiler implementation. Unlike existing applications that simulate grammars or automata in an abstract environment, this educational compiler allows users to input real code and observe how the system applies generative grammar to construct a derivation tree.
Furthermore, it introduces a semantic analysis phase and graphical representation of syntax trees, elements that are often absent or abstracted in other learning tools. While powerful compiler generators like ANTLR or yacc exist, they often hide the internal mechanisms that students need to understand. In contrast, this tool focuses on transparency and simplicity, providing clear insight into how tokens are generated, how grammar rules are applied, and how derivation trees are built and visualized in a real-time environment.

Chapter 3

Methodology

The development of this educational compiler followed a structured approach based on core principles of language theory and software engineering. The methodology focused on the systematic construction of lexical, syntactic, and semantic analysis components, all implemented in Java with modular design. All of this implemented with these methods

- Generative grammar: A simplified programming language was formally described using a manually defined context free grammar (CFG). Each production rule represents a structural pattern commonly found in beginner level code. This CFG enables the compiler to systematically validate user input by deriving a parse structure based on rule conformity. This was the context free used in the project.
  <S> -> "START" | <Instruction> | "END"
  <Instructions> -> <Instruction> <Instructions> | lambda
  <Instruction> -> "let" <Identifier> "=" <Expression> | "print" <Identifier> | "if" "("
  <Condition> ")" <Block> | "while" "(" <Condition> ")" <Block>
  <Block> -> "{" <Instructions> "}"
  <Expression> -> <Identifier> <Operator> <Expression> | <Identifier> | <Number>
  <Condition> -> <Expression>
  <Identifier> -> [a-zA-Z][a-zA-Z0-9]*
  <Numeber> -> [0-9]+
  <Operator> -> "+" | "-" | "*" | "/" | "=" | "<" | ">"

- Token Analysis: Lexical analysis segments the input into meaningful units called tokens (e.g., identifiers, operators, numbers). This method simplifies parsing by abstracting raw text into categorized elements, enabling rule based grammar application.

- Derivation tree (Syntax Analysis): A recursive method parser consumes the list of tokens produced by the lexer and constructs a derivation tree. This tree reflects how the input code conforms to the grammar rules, where each node corresponds to a grammar symbol and the branches represent applied production rules. Conditional structures such as if-statements and loops are properly nested, and each sub expression is parsed according to its grammatical hierarchy.

- Semantic Analysis: After syntax validation, a semantic analyzer checks the logical structure of the code by analyzing contextual elements. This step ensures that code meaning is coherent and that operations are semantically valid within the designed domain.

- Integration in java:The compiler is developed using Java's object-oriented principles. Each phase: lexical analysis, parsing, and semantic validation is encapsulated in its

own class. The graphical user interface (GUI) is implemented using JavaFX, allowing a visualization of derivation trees. Also the error messages are handled gracefully, guiding users to correct syntax or semantic mistakes.

Chapter 4

Results

- The educative compiler was developed successfully and completely functional in Java, designed specifically to a best understanding of the compiler's fundamentals. This compiler has the capacity to process a simple source code written by the student and generate the derivation tree in a visual form. This result shows the principal goal of the project.
- This project shows the way the successfully integration of the three analyzers: lexical, syntactic and semantic in only one application. Also, the project allows the integration and implementation of new grammar rules or even personalized tokens.
- The graphical component allows to the student watch the derivation tree from the source code. Each one of the nodes represents a terminal or non terminal symbol that change it content depending on the token joined
- The semantic of the compiler is very simple and the analyzer verifies it structure in terms of consistency. I.e. verifies that the instructions are in a block that begin with an START and end with an END and the expressions are correctly formed, helping to consolidate fundamental concepts of the semantic analyzer
- The compiler has the capacity to detect and report basic and common errors during the lexical and syntactic phase, such as the the parenthesis or bad closed blocks, through clear and understandable message in a information box. Transmitting the capacity of the identification of errors to the students.
- Finally, the project use the Java library, JavaFX, this library offers a friendly and useful experience to code and compile the tree and it characteristics without issues

Chapter 5

Conclusion

- The development of this educational compiler successfully demonstrates the practical application of lexical, syntactic, and semantic analysis within a single modular system. By translating a student written input into derivation trees, the project effectively bridges theoretical grammar rules with concrete compiler behavior.
- The visual representation of derivation trees offers a strong visual tool that improves understanding. It enables newcomers in computer science and compiler construction to understand abstract ideas such as parsing and grammar derivation via clear and prompt feedback.
- This project highlights the importance of visualization in technical education. The use of a simplified programming language, coupled with an error reporting and tree generation, makes the compiler an accessible and effective learning tool for understanding the core principles behind language processing and compiler construction.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson, 2006.

[2] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. Langendoen, *Modern Compiler Design*, 2nd ed. New York, NY, USA: Springer, 2012.

[3] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2001.

[4] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ, USA: Prentice Hall, 1976.

[5] C. N. Fischer and R. J. LeBlanc, *Crafting a Compiler*. Redwood City, CA, USA: Benjamin Cummings, 1988.

[6] M. Ben-Ari, "Constructivism in Computer Science Education," in Journal of Computers in Mathematics and Science Teaching, vol. 20, no. 1, pp. 45–73, 2001.

[7] A. DeGiovanni and A. O'Hara, "Visualizing Compiler Design Concepts Through Java-Based Educational Tools," in Journal of Computing Sciences in Colleges, vol. 20, no. 4, pp. 116–123, Apr. 2005.