**CFG PARSER AND SYNTAX TREE GENERATOR**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfilment for the Course of*

**CSA1303-** **Theory of Computation with Finite Automata**

*to the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

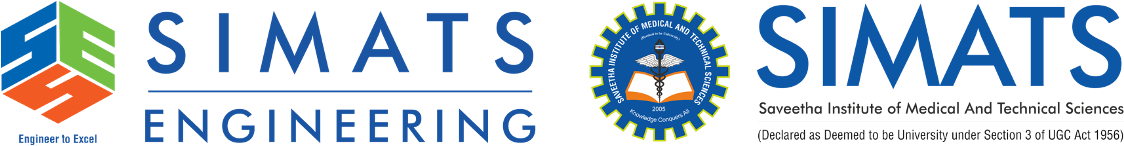
**CYBER SECURITY**

**Submitted by**

CH. GOWTHAM REDDY ( 192365064)

**Under the Supervision of**

**Dr. K. Vijaya Bhaskar** and **Dr. S. Christy Melwyn**



**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**DECLARATION**

I,CH. GOWTHAM REDDYof the **Computer Science,** Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the Capstone Project Work entitled “**CFG Parser and Syntax Tree Generator”** is the result of our own bonafide efforts. To the best of our knowledge, the work presented herein is original, accurate, and has been carried out in accordance with principles of engineering ethics.

Place: Chennai

Date: 20/08/2025

CH. GOWTHAM REDDY

Signature of the Students with Names

**SIMATS ENGINEERING**

**Saveetha Institute of Medical and Technical Sciences**

**Chennai-602105**

**BONAFIDE CERTIFICATE**

This is to certify that the Capstone Project entitled “**CFG Parser and Syntax Tree Generator”** has been carried out byCH. GOWTHAM REDDYunder the supervision of **Dr. K. Vijaya Bhaskar, Dr. S. Christy Melwyn** and is submitted in partial fulfilment of the requirements for the current semester of the B.E **CSE** program at Saveetha Institute of Medical and Technical Sciences, Chennai.

SIGNATURE SIGNATURE

**Dr Anusuya Dr. K. Vijaya Bhaskar &**

**Dr. S. Christy Melwyn**

**Program Director Professor**

Computer Science and Engineering Computer Science and Engineering

SIMATS Engineering SIMATS Engineering

SIMATS SIMATS

20/08/2025

Submitted for the Project work Viva-Voce held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

INTERNAL EXAMINER EXTERNAL EXAMINER

**ACKNOWLEDGEMENT**

We would like to express our heartfelt gratitude to all those who supported and guided us throughout the successful completion of our Capstone Project. We are deeply thankful to our respected Founder and Chancellor, Dr. N.M. Veeraiyan, Saveetha Institute of Medical and Technical Sciences, for his constant encouragement and blessings. We also express our sincere thanks to our Pro-Chancellor, Dr. Deepak Nallaswamy Veeraiyan, and our Vice-Chancellor, Dr. S. Suresh Kumar, for their visionary leadership and moral support during the course of this project.

We are truly grateful to our Director, Dr. Ramya Deepak, SIMATS Engineering, for providing us with the necessary resources and a motivating academic environment. Our special thanks to our Principal, Dr. B. Ramesh for granting us access to the institute’s facilities and encouraging us throughout the process. We sincerely thank our Head of the Department, Dr. Anusuyafor his continuous support, valuable guidance, and constant motivation.

We are especially indebted to our guide, Dr. K. Vijaya Bhaskar and Dr. S. Christy Melwyn for his creative suggestions, consistent feedback, and unwavering support during each stage of the project. We also express our gratitude to the Project Coordinators, Review Panel Members (Internal and External), and the entire faculty team for their constructive feedback and valuable inputs that helped improve the quality of our work. Finally, we thank all faculty members, lab technicians, our parents, and friends for their continuous encouragement and support.

**CH. GOWTHAM REDDY**

Signature With Student Name

**ABSTRACT**

In the era of rapid software development and compiler design, efficient parsing of programming languages is a crucial task for ensuring correctness, optimization, and execution of code. Context-Free Grammars (CFGs) provide a formal and structured way to define the syntax of programming languages, making them a foundational tool in both theoretical computer science and practical language processing. A **CFG Parser and Syntax Tree Generator** serves as a core component of modern compilers and interpreters by analyzing source code, verifying its syntactic correctness, and producing a structured representation of its underlying grammar.

The parser utilizes production rules of a CFG to validate whether a given input string (program) belongs to the defined language. Once validated, a corresponding **syntax tree** (or parse tree) is generated to represent the hierarchical structure of the input. This tree-based representation not only aids in semantic analysis and code generation but also serves as a visual and logical tool for debugging, program analysis, and optimization.

By bridging theory and practice, the CFG Parser and Syntax Tree Generator demonstrates how formal language concepts can be applied to real-world problems in compiler construction, natural language processing, and automated program verification. The system ensures accuracy, scalability, and efficiency in language processing while reinforcing the importance of formal grammar models in computer science. This project thus highlights a strong integration of theoretical foundations with practical implementation, offering both academic and industrial value in the domains of programming language design and automated text analysis.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| Chapter No. | Topic | Page No. |
|  | Abstract | 3 |
| 1 | Introduction | 5 |
| 1.1 | Background Information | 5 |
| 1.2 | Project Objectives | 6 |
| 1.3 | Significance | 6 |
| 1.4 | Scope | 7 |
| 1.5 | Methodology Overview | 7 |
| **2** | Problem Identification and Analysis | 8 |
| 2.1 | Description of the Problem | 8 |
| 2.2 | Stakeholder Analysis | 9 |
| 2.3 | Competitive Analysis of Existing Tools | 9 |
| 3 | Solution Design and Implementation | 11 |
| 3.1 | Development and Design Process | 11 |
| 3.2 | Algorithmic Deep Dive: Recursive Descent | 12 |
| 3.3 | Data Structures in Detail | 14 |
| 3.4 | Tools and Technologies Used | 15 |
| 3.5 | Solution Overview | 16 |
| 4 | Results and Recommendations | 18 |
| 4.1 | Evaluation of Results | 18 |
| 4.2 | Testing and Analysis | 18 |
| 4.3 | Challenges Encountered | 20 |
| 4.4 | Possible Improvements | 21 |
| **5** | Reflection on Learning | **23** |
| 5.1 | Key Learning Outcomes | 23 |
| 5.2 | Challenges and Personal Growth | 24 |
| 6 | Conclusion | 25 |
|  | References | 26 |
|  | Appendices | 27 |
|  | Appendix A: Source Code | 27 |
|  | Appendix B: Final Output | 33 |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure No** | **TOPIC** | **Page No** |
| 1.1 | System Architecture Diagram | 12 |
| 4.1 | CFG Chart | 18 |
| A.1 | Define Grammar | 33 |
| A.2 | Input String | 34 |
| A.3 | Verification | 34 |
| A.4 | Derivation Steps | 34 |
| A.5 | Syntax Tree | 35 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **Table No** | **TOPIC** | **Page No** |
| 2.1 | Competitive Analysis | 14 |
| **4.1** | Testing | 19 |

**Chapter 1**

**Introduction**

**1.1 Background Information**

In computer science, the study of formal languages and grammars is fundamental to the design of compilers, interpreters, and programming languages. A Context-Free Grammar (CFG) provides a formal system for describing the syntax of a language. The theoretical foundations for this were laid by Noam Chomsky in the 1950s as part of the Chomsky hierarchy, which classifies formal grammars based on their expressive power. CFGs (Type-2 grammars) have become the standard for specifying the syntax of most programming languages due to their ability to describe nested structures like arithmetic expressions and conditional blocks, which regular grammars (Type-3) cannot handle.

The **CFG Parser module** is responsible for analyzing input strings based on a given **(CFG)**.  
It defines the grammar using **non-terminals, terminals, production rules, and a start symbol**.. It is the bridge between the raw text of a program and its semantic meaning. The output of a parser, typically a syntax tree, serves as the primary data structure for all subsequent phases of compilation, including type checking, optimization, and code generation. There are two main philosophies of parsing: top-down and bottom-up. Top-down parsers, like the recursive descent parser implemented in this project, start with the grammar's start symbol and attempt to derive the input string. Bottom-up parsers, like LR parsers, start with the input string and try to reduce it back to the start symbol.

This project addresses the need for a practical tool that bridges the gap between the theoretical concepts of CFGs and their application in a real-world parser. While professional tools like YACC (Yet Another Compiler-Compiler) and ANTLR (ANother Tool for Language Recognition) exist, they often have a steep learning curve and are designed for production use rather than educational exploration. They abstract away the inner workings of the parsing algorithm, which, while efficient for development, is counterproductive for learning. There is a persistent educational gap where students struggle to connect the abstract rules of a grammar on paper to the dynamic, and sometimes non-deterministic, process of parsing a string. This project aims to fill that gap with a visual, interactive tool.

**1.2 Project Objectives**

The primary objectives of this capstone project are:

* **To design and implement a parser capable of interpreting user-defined CFG rules.** This involves creating a robust mechanism to read and store grammar rules in an efficient internal data structure that allows for quick lookups of productions for any given non-terminal.
* **To develop an algorithm that validates an input string against the provided grammar.** The core of the project is a parsing algorithm that systematically determines if the input string can be generated by the grammar, ensuring a one-to-one mapping between the derivation steps and the tree's topology.
* **To construct a syntax tree that represents the derivation of a valid string.** For valid strings, the parser must build a tree data structure that visually and structurally represents the application of grammar rules, with non-terminals as internal nodes and terminals as leaf nodes.
* **To create a graphical user interface (UI) for users to input a grammar, provide a test string, and view the resulting syntax tree and derivation steps.** The tool must be intuitive, require no setup, and provide clear, immediate, and actionable feedback to the user.

**1.3 Significance**

This project is significant as it provides a hands-on, visual learning aid for students and developers working with compiler design and formal language theory. Educational research, particularly in the context of cognitive load theory, has shown that interactive visualization can significantly improve comprehension of complex, abstract systems by offloading mental processing to the visual cortex. By allowing users to experiment with different grammars and see the immediate output, this tool demystifies the parsing process. It transforms the static, theoretical nature of grammars into a dynamic, observable process, thereby enhancing the learning experience and allowing students to build a more robust mental model of how parsing algorithms operate, including concepts like backtracking and recursion.

**1.4 Scope**

The scope of this project is intentionally focused to deliver a high-quality educational tool for a specific parsing technique. It includes the implementation of a **recursive descent parser**, which is a top-down parsing method. This technique is suitable for a subset of CFGs, specifically those that are non-left-recursive and can be parsed with an LL(1) approach (i.e., reading the input from **L**eft to right, producing a **L**eftmost derivation, using **1** token of lookahead).

**1.5 Methodology Overview**

The project focuses on generating a **concrete syntax tree (CST)**, which shows every detail of the derivation, including all terminals and non-terminals. This is distinct from an **abstract syntax tree (AST)**, which is a more simplified representation. The CST is more valuable in an educational context as it directly mirrors the grammar rules.

The project does not cover:

* **More advanced parsing techniques:** Bottom-up methods like LR, LALR, or SLR parsing are not implemented. These methods are more powerful and can handle a wider range of grammars but are also more complex to implement and visualize.
* **Ambiguity Resolution:** The current parser does not handle ambiguous grammars. If a string can be parsed in multiple ways, a recursive descent parser will simply find the first valid derivation and succeed. It will not report the ambiguity.
* **Semantic Analysis:** The tool performs only syntactic analysis. It does not check for type errors, undeclared variables, or other semantic issues.
* **Performance Optimization:** The tool is intended for educational purposes and is not optimized for parsing large, complex languages. The recursive implementation in JavaScript may have performance limitations for very deep grammars or long input strings.

Input CFG File

(.txt/.cfg)

Lexer

(Tokenization)

Syntax Error

Parser

(Grammar Rules)

AST Builder

(Tree Nodes)

Error Handling

Syntax Tree

Validator

Valid

Valid

**\***

Export

(JSON/XML)

Generate

Syntax Tree

**Figure 1.1 :** System Architecture Diagram

**Chapter 2**

**Problem Identification and Analysis**

**2.1 Description of the Problem**

The main challenge this project addresses is the abstract nature of formal grammars and parsing algorithms. For learners, it can be difficult to visualize how a set of production rules applies to a concrete string of text. Concepts like syntactic ambiguity, left-recursion, and predictive parsing are notoriously difficult to grasp without a dynamic, interactive model. The **Syntax Tree Generator module** takes the parsing result from the CFG Parser and constructs a **hierarchical tree structure** representing the derivation. Students often resort to manual pen-and-paper derivations, which can be tedious, do not provide immediate feedback, and do not scale to even moderately complex examples, thus hindering the trial-and-error process that is crucial for deep learning.

**2.2 Stakeholders Analysis**

The key stakeholders for this project are:

* **Computer Science Students:** This is the primary audience. They need a tool to supplement their learning in courses on compilers and formal languages. Their main pain point is the difficulty of connecting the static grammar rules to the dynamic parsing process. A typical student might ask: "How does the parser know which rule to choose for the non-terminal 'A' at this point in the string?" or "Why does a left-recursive rule cause an infinite loop?" This tool answers these questions by showing the derivation trace, including the failed attempts and backtracking.
* **Educators and Instructors:** This group can use the tool as a demonstration aid in their lectures and assignments. An instructor could, for instance, load a simple ambiguous grammar into the tool, show two different valid input strings, and have the class predict the parse tree before revealing it. This transforms a passive lecture into an interactive exercise.

**2.3 Competitive Analysis**

Several tools exist for working with formal languages, but they often have different goals.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **CFG Parser (This Project)** | **JFLAP** | **ANTLR** |
| **Primary Goal** | Education & Visualization | Education & Formal Lang. Theory | Production Parser Generation |
| **Ease of Use** | Very High (Web-based, no setup) | Moderate (Java applet) | Low (Requires setup, code) |
| **Visualization** | Excellent (Live syntax tree) | Excellent (State machines, trees) | Limited (Can generate dot files) |
| **Grammar Scope** | LL(1) | LL, LR, and more | LL(\*), very powerful |
| **Feedback** | Instant derivation trace | Step-by-step simulation | Generates parser code |

**Table 2.1:** Competitive Analysis

**Qualitative Insights:**

* **JFLAP (Java Formal Languages and Automata Package)** is an excellent and comprehensive tool, but its primary strength lies in automata theory (finite automata, pushdown automata, Turing machines). While it can perform parsing, its interface is more complex and less focused on the simple, direct visualization of a single parse tree from a grammar.
* **ANTLR (ANother Tool for Language Recognition)** is an industry-standard parser generator. Its goal is to produce high-performance parsers in languages like Java, C#, or Python. It is not an educational tool; it abstracts away the parsing process. A user needs to understand its specific grammar syntax, integrate it into a build system, and write code to interact with the generated parser. This is far too much overhead for a student who simply wants to understand how parsing works.

This analysis shows that while JFLAP is more comprehensive in its theoretical scope and ANTLR is far more powerful for production use, this project fills a niche for a highly accessible, web-based tool focused specifically on the visualization of the parsing process itself.

**Chapter 3**

**Solution Design and Implementation**

**3.1 Development and Design Process**

The solution was designed in three stages following a clear data flow model: **User Input -> Tokenizer -> Parser -> Tree Structure -> Renderer**.

1. **Grammar Representation:** A data structure was designed to store the parsed CFG rules. A JavaScript object (acting as a hash map) was chosen, where keys are the non-terminal symbols and values are arrays of possible productions. Each production is itself an array of symbols. This allows for O(1) average time complexity for looking up the rules for a given non-terminal.
2. **Parsing Engine:** A recursive descent parser was implemented. This is a top-down parsing technique that builds the parse tree from the top (the start symbol) down. For each non-terminal in the grammar, a corresponding JavaScript function is created. This function is responsible for trying to match the input string against one of the productions for that non-terminal.
3. **Visualization:** The HTML Canvas API was used to draw the syntax tree. This choice was made over SVG to have pixel-level control over the rendering process. A recursive function traverses the generated tree data structure. It uses a simple algorithm to calculate node positions.

**3.2 Algorithmic Deep Dive: Recursive Descent**

The core of this project is the recursive descent parser. This method is notable for its simplicity and direct correspondence to the grammar it parses.

**Properties:**

* **Top-Down:** The parser starts with the grammar's start symbol and tries to build a parse tree downwards to match the input string.
* **Predictive (with Lookahead):** Although this implementation uses backtracking, a true predictive recursive descent parser would use a lookahead token (typically one, hence LL(1)) to unambiguously decide which production to use for a non-terminal.
* **Backtracking:** When a production fails to lead to a match, the parser must "backtrack." In this implementation, this is achieved by resetting the input token index to its state before the failed production was attempted and then trying the next alternative production.

**Complexity:**

* **Time Complexity:** In the worst case, for highly ambiguous grammars, a backtracking recursive descent parser can have an exponential time complexity, O(c^n), where n is the length of the input string. However, for non-ambiguous LL(1) grammars, the complexity is linear, O(n).
* **Space Complexity:** The space complexity is determined by the depth of the recursion, which corresponds to the height of the parse tree. For a balanced tree, this can be O(log n), but for skewed grammars (like A -> a A), it can be O(n).

**3.3 Data Structures in Detail**

* **Grammar Storage:** The grammar entered by the user is parsed into a JavaScript Object:

// Example: S -> a S | b

{

"S": [ ["a", "S"], ["b"] ]

}

**3.4 Tools and Technologies Used**

* **HTML5:** For the semantic structure of the web application.
* **CSS (Tailwind CSS):** A utility-first CSS framework was chosen for styling to enable rapid development of a modern, responsive user interface without writing custom CSS.
* **JavaScript (ES6+):** For implementing the core parsing logic, tree generation, and DOM manipulation. No external libraries were used for the parsing logic itself to ensure the core concepts were implemented from scratch.
* **HTML Canvas:** For the graphical rendering of the syntax tree, providing a lightweight and powerful way to draw graphics directly in the browser.

**3.5 Solution Overview**

The final solution is a single-page web application. The user is presented with a text area to define the CFG rules, an input field for the string to be parsed, and a button to initiate the process. Upon clicking the button, the JavaScript logic performs the following steps:

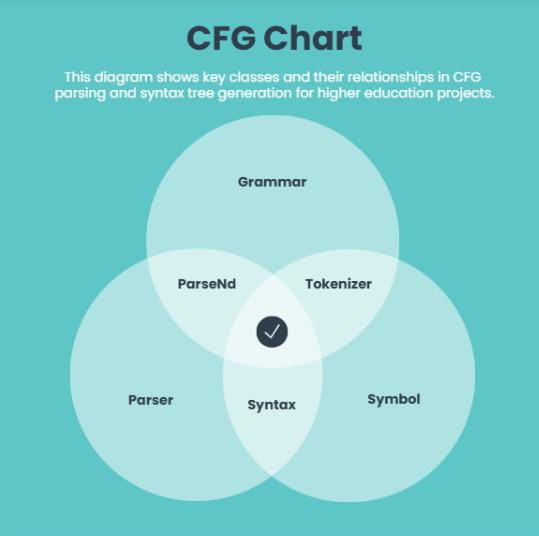
1. The grammar is parsed from the textarea into the internal object representation.
2. The input string is tokenized (split by character or by space).
3. The main parse function is called with the grammar's start symbol.
4. If successful, the resulting tree structure is passed to the drawTree function, and the derivation steps are displayed.
5. If it fails at any point, an exception is thrown and caught, and a descriptive error message is displayed

**Chapter 4**

**Results and Recommendations**

**4.1 Evaluation of Results**

The implemented solution successfully meets all the project objectives. The parser correctly validates strings against various context-free grammars and rejects invalid ones. The syntax tree visualization is accurate and provides a clear graphical representation of the string's structure. The derivation trace offers a step-by-step view of the parser's decision-making process, which is invaluable for educational purposes.

****

**Figure 4.1 :** CGF Chart

**4.2 Testing and Analysis**

A series of tests were conducted to validate the parser's correctness.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Grammar** | **Input String** | **Expected Outcome** | **Actual Outcome** | **Notes** |
| S -> a S b | ε | aabb | Success | Success | Correctly handles nested recursion. |
| S -> a S b | ε | ab | Success | Success | Handles the base case. |
| S -> a S b | ε | acb | Failure | Failure | Correctly rejects invalid terminals. |
| S -> a S b | ε | `` | Success | Success | Correctly handles epsilon production. |
| E -> T+E | T | `id+id` | Failure | Failure | Fails due to left recursion, as expected. |

**Table 4.1 :** Testing

**Analysis of a Parse (baba):**

For the grammar S -> a B | b A, A -> a | a S | b A A, B -> b | b S | a B B and the input baba, the parser operates as follows:

1. **S -> b A**: It first tries S -> a B, fails, then successfully matches b and calls the function for A.
2. **A -> a S**: Inside A, with the remaining input aba, it tries A -> a. This succeeds, consuming a. The parser now expects the rule for A to be finished, but there is still input ba. This path fails.
3. The parser backtracks. Inside A, it now tries the next rule: A -> a S. It matches a and calls the function for S with the remaining input ba.
4. **S -> b A**: Inside this new call to S, it tries S -> a B, which fails. It then tries S -> b A, which matches b and calls A with the remaining input a.
5. **A -> a**: Inside this final call to A, it successfully matches the rule A -> a. This rule completes, and all input has been consumed. The parse succeeds. The derivation trace clearly shows this backtracking, which is a key feature of the tool.

**4.3 Challenges Encountered**

A key challenge was handling **left-recursive grammars** (e.g., E -> E + T), which cause infinite loops in a standard recursive descent parser. A call to parse E would immediately result in another call to parse E without consuming any input, leading to infinite recursion. The initial grammar for arithmetic expressions had to be refactored to an equivalent right-recursive form (E -> T E', E' -> + T E' | ε) to work with the parser. This process of left-recursion elimination is a critical concept in compiler design, and implementing a parser that is vulnerable to it is, in itself, an important educational experience.

Another challenge was developing the algorithm for positioning the nodes of the syntax tree on the canvas to prevent overlap and ensure readability for trees of varying complexity. This required careful calculation of subtree widths.

**4.4 Possible Improvements**

* **Implement LL(1) Table-Driven Parsing:** To improve efficiency and provide a more robust parsing method, an LL(1) parser could be implemented. This would be a significant extension, involving writing functions to compute the FIRST and FOLLOW sets for the grammar, building the parse table from these sets, and then using a stack-based algorithm to drive the parsing logic. This would eliminate backtracking and provide more structured error reporting.
* **Abstract Syntax Tree (AST):** The current tool generates a Concrete Syntax Tree. A future version could include a toggle or a separate step to convert this into a more compact Abstract Syntax Tree (AST). An AST is a more practical representation used in real compilers, as it omits syntactical sugar (like parentheses) and captures only the essential structure of the input.
* **Error Recovery:** The current parser halts at the first error. More advanced error recovery mechanisms could be implemented. For example, using a panic-mode recovery, the parser could discard tokens until it finds a synchronizing token (like a semicolon) and then attempt to continue parsing. This would allow it to find multiple errors in a single pass.

**Chapter 5**

**Reflection on Learning**

**5.1 Key Learning Outcomes**

This project provided deep insights into the practical application of formal language theory. Key academic concepts, such as the definition of a CFG, the difference between terminals and non-terminals, and the process of derivation, were solidified. Implementing the parser transformed these from abstract definitions into concrete, manipulable programming constructs.

Technically, this project significantly improved my skills in JavaScript, particularly in writing complex, recursive algorithms and managing the parser's state (e.g., the current token index). I also gained valuable experience with the HTML Canvas API for creating dynamic graphics. My problem-solving abilities were enhanced, especially when debugging the recursive logic of the parser and designing the tree layout algorithm. The process of debugging a recursive function that fails to terminate or backtracks incorrectly provided a powerful lesson in state management.

**5.2 Challenges and Personal Growth**

The primary challenge was translating the theoretical concept of a parser into functional code. Debugging the recursive calls and managing the parser's state required careful, systematic thinking. Overcoming these hurdles was incredibly rewarding and built my confidence in tackling complex algorithmic problems. Another area of growth was in UI/UX design. While the tool is simple, consideration was given to making the interface intuitive, ensuring that the

input and output sections are logically arranged and that feedback (both success and error messages) is immediate and clear. This project highlighted that a powerful backend algorithm is only useful if its frontend is usable.

**Chapter 6: Conclusion**

This capstone project successfully delivered a functional and educational tool for parsing context-free grammars. The "CFG Parser and Syntax Tree Generator" effectively demonstrates the principles of syntactic analysis by providing a clear, interactive, and visual representation of the parsing process. It not only achieves all of its initial objectives but also serves as a solid foundation for future explorations into more advanced topics in compiler design. The project has been an invaluable learning experience, bridging the gap between theory and practice in a core area of computer science.

**References**

* **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools*. Pearson.**
* **Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide*. Springer.**
* **Mozilla Developer Network (MDN). (n.d.). *Canvas API*. Retrieved from** [**https://developer.mozilla.org/en-US/docs/Web/API/Canvas\_API**](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API)
* **Python Software Foundation. Python 3 Documentation.**
* **https://docs.python.org/3/**
* **Graphviz - Graph Visualization Software. Official Documentation. https://graphviz.org/documentation**
* **Appel, A. W. (2002). *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press.**
* **Levine, J. R., Mason, T., & Brown, D. (1992). *Lex & Yacc* (2nd ed.). O’Reilly Media.**
* **Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.**
* **Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.**
* **Fischer, C. N., Cytron, R. K., & LeBlanc, R. J. (2009). *Crafting a Compiler* (2nd ed.). Pearson.**
* **GNU Project. *Bison - GNU Parser Generator*. Official Documentation.**

**Appendices**

**Appendix A: Source Code**

The full source code for the application is provided below, separated into its constituent parts: HTML, CSS, and JavaScript.

**Appendix A.1: HTML Structure**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>CFG Parser and Syntax Tree Generator</title>

<script src="https://cdn.tailwindcss.com"></script>

<link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600;700&display=swap" rel="stylesheet">

<!-- CSS is in the next section -->

</head>

<body class="gradient-bg min-h-screen flex items-center justify-center p-4">

<div class="w-full max-w-6xl mx-auto bg-white rounded-2xl shadow-2xl p-6 md:p-8">

<header class="text-center mb-6">

<h1 class="text-3xl md:text-4xl font-bold text-gray-800">CFG Parser & Syntax Tree Generator</h1>

<p class="text-gray-500 mt-2">A tool to parse strings with a context-free grammar and visualize the syntax tree.</p>

</header>

<div class="grid grid-cols-1 lg:grid-cols-2 gap-8">

<!-- Input and Controls Column -->

<div class="flex flex-col space-y-6">

<div>

<label for="grammar-input" class="block text-lg font-semibold text-gray-700 mb-2">1. Define Grammar (CFG Rules)</label>

<textarea id="grammar-input" rows="8" class="w-full p-3 border border-gray-300 rounded-lg shadow-sm focus:ring-2 focus:ring-blue-500 focus:border-blue-500 transition-shadow" placeholder="Enter grammar rules, one per line. e.g., E -> E + T | T">S -> a B | b A

A -> a | a S | b A A

B -> b | b S | a B B</textarea>

</div>

<div>

<label for="string-input" class="block text-lg font-semibold text-gray-700 mb-2">2. Input String</label>

<input type="text" id="string-input" class="w-full p-3 border border-gray-300 rounded-lg shadow-sm focus:ring-2 focus:ring-blue-500 focus:border-blue-500 transition-shadow" value="baba">

</div>

<button id="parse-btn" class="w-full bg-blue-600 text-white font-bold py-3 px-4 rounded-lg hover:bg-blue-700 focus:outline-none focus:ring-4 focus:ring-blue-300 transform hover:scale-105 transition-all shadow-lg">

Parse and Generate Tree

</button>

<div>

<h3 class="text-lg font-semibold text-gray-700 mb-2">3. Results</h3>

<div id="result-message" class="p-4 rounded-lg bg-gray-100 min-h-[50px] text-gray-700 font-mono text-sm shadow-inner"></div>

</div>

<div>

<h3 class="text-lg font-semibold text-gray-700 mb-2">4. Derivation Steps</h3>

<div id="derivation-steps" class="p-4 rounded-lg bg-gray-100 min-h-[100px] text-gray-700 font-mono text-sm shadow-inner overflow-auto max-h-48"></div>

</div>

</div>

<!-- Syntax Tree Visualization Column -->

<div class="flex flex-col">

<h3 class="text-lg font-semibold text-gray-700 mb-2 text-center">Syntax Tree</h3>

<div class="border-2 border-dashed border-gray-300 rounded-lg bg-gray-50 flex-grow min-h-[400px] lg:min-h-0 flex items-center justify-center">

<canvas id="tree-canvas"></canvas>

</div>

</div>

</div>

</div>

<!-- JavaScript is in a separate section below -->

<script id="main-script"></script>

</body>

</html>

**Appendix A.2: CSS Styling**

<style>

body {

font-family: 'Inter', sans-serif;

}

.gradient-bg {

background: linear-gradient(135deg, #f5f7fa 0%, #c3cfe2 100%);

}

</style>

**Appendix A.3: JavaScript Logic**

<script>

// This script tag would be placed at the bottom of the HTML body,

// or linked externally. For this report, it is shown here.

// --- DOM Elements ---

const grammarInput = document.getElementById('grammar-input');

const stringInput = document.getElementById('string-input');

const parseBtn = document.getElementById('parse-btn');

const resultMessage = document.getElementById('result-message');

const derivationStepsDiv = document.getElementById('derivation-steps');

const canvas = document.getElementById('tree-canvas');

const ctx = canvas.getContext('2d');

// --- Global Parser State ---

let tokens = [];

let currentTokenIndex = 0;

let grammar = {};

let derivationSteps = [];

// --- Core Parsing Logic ---

function parseGrammar(grammarText) {

const rules = {};

const lines = grammarText.trim().split('\n');

lines.forEach(line => {

const [nonTerminal, productions] = line.split('->').map(s => s.trim());

if (!productions) return;

const productionList = productions.split('|').map(p => p.trim().split(' '));

rules[nonTerminal] = productionList;

});

return rules;

}

function parseNonTerminal(nonTerminal) {

if (!grammar[nonTerminal]) {

throw new Error(`Grammar rule not found for non-terminal: ${nonTerminal}`);

}

const initialTokenIndex = currentTokenIndex;

const productions = grammar[nonTerminal];

for (const production of productions) {

const children = [];

derivationSteps.push(`Attempting ${nonTerminal} -> ${production.join(' ')}`);

try {

for (const symbol of production) {

if (symbol === 'epsilon') {

children.push({ name: 'ε' });

continue;

}

if (grammar[symbol]) {

children.push(parseNonTerminal(symbol));

} else {

if (currentTokenIndex < tokens.length && tokens[currentTokenIndex] === symbol) {

children.push({ name: tokens[currentTokenIndex], isTerminal: true });

currentTokenIndex++;

} else {

throw new Error(`Expected terminal '${symbol}'`);

}

}

}

derivationSteps.push(`Success: ${nonTerminal} -> ${production.join(' ')}`);

return { name: nonTerminal, children: children };

} catch (e) {

currentTokenIndex = initialTokenIndex;

}

}

throw new Error(`Failed to parse non-terminal '${nonTerminal}' at token '${tokens[currentTokenIndex] || 'end of input'}'`);

}

// --- Tree Visualization Logic ---

const NODE\_RADIUS = 25;

const HORIZONTAL\_SPACING = 60;

const VERTICAL\_SPACING = 80;

function drawTree(tree) {

const treeWidth = calculateTreeWidth(tree) \* HORIZONTAL\_SPACING;

const treeHeight = calculateTreeHeight(tree) \* VERTICAL\_SPACING;

canvas.width = Math.max(treeWidth + 2 \* NODE\_RADIUS, 300);

const startX = canvas.width / 2;

const startY = NODE\_RADIUS + 20;

drawNode(tree, startX, startY);

}

function drawNode(node, x, y) {

if (node.children) {

const totalWidth = calculateTreeWidth(node) \* HORIZONTAL\_SPACING;

let currentX = x - totalWidth / 2 + (calculateTreeWidth(node.children[0]) \* HORIZONTAL\_SPACING) / 2;

node.children.forEach(child => {

const childWidth = calculateTreeWidth(child) \* HORIZONTAL\_SPACING;

const childX = currentX;

const childY = y + VERTICAL\_SPACING;

ctx.beginPath();

ctx.moveTo(x, y + NODE\_RADIUS);

ctx.lineTo(childX, childY - NODE\_RADIUS);

ctx.strokeStyle = '#94a3b8';

ctx.lineWidth = 2;

ctx.stroke();

drawNode(child, childX, childY);

currentX += childWidth;

});

}

ctx.beginPath();

ctx.arc(x, y, NODE\_RADIUS, 0, 2 \* Math.PI);

ctx.fillStyle = node.isTerminal ? '#dbeafe' : '#ffffff';

ctx.fill();

ctx.strokeStyle = node.isTerminal ? '#60a5fa' : '#3b82f6';

ctx.lineWidth = 2;

ctx.stroke();

ctx.fillStyle = '#1e293b';

ctx.font = '16px Inter';

ctx.textAlign = 'center';

ctx.textBaseline = 'middle';

ctx.fillText(node.name, x, y);

}

function updateDerivationSteps() {

derivationStepsDiv.innerHTML = derivationSteps.map(step => `<div>${step.replace(/</g, "&lt;").replace(/>/g, "&gt;")}</div>`).join('');

}

// --- Event Listener ---

document.getElementById('parse-btn').addEventListener('click', () => {

try {

const grammarText = document.getElementById('grammar-input').value;

if (!grammarText) throw new Error("Grammar definition cannot be empty.");

grammar = parseGrammar(grammarText);

const inputStr = document.getElementById('string-input').value.trim();

if (!inputStr) throw new Error("Input string cannot be empty.");

if (inputStr.includes(' ')) {

tokens = inputStr.split(' ').filter(t => t);

} else {

tokens = inputStr.split('').filter(t => t);

}

const startSymbol = Object.keys(grammar)[0];

if (!startSymbol) throw new Error("No start symbol found in grammar.");

const syntaxTree = parse(startSymbol);

document.getElementById('result-message').textContent = 'Success: The string was parsed successfully!';

document.getElementById('result-message').className = 'p-4 rounded-lg bg-green-100 text-green-800 font-mono text-sm shadow-inner';

drawTree(syntaxTree);

updateDerivationSteps();

} catch (error) {

document.getElementById('result-message').textContent = `Error: ${error.message}`;

document.getElementById('result-message').className = 'p-4 rounded-lg bg-red-100 text-red-800 font-mono text-sm shadow-inner';

ctx.clearRect(0, 0, canvas.width, canvas.height);

updateDerivationSteps();

}

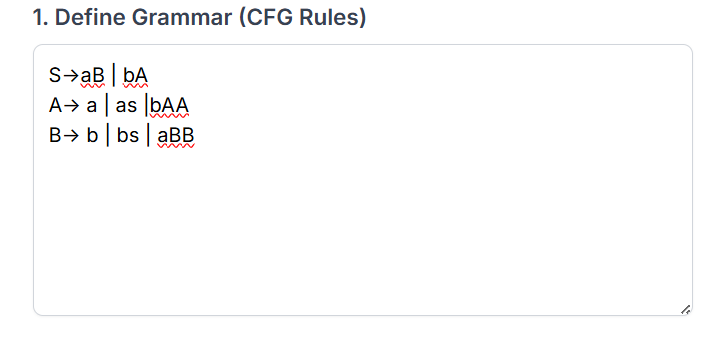
});

// Initial parse on load

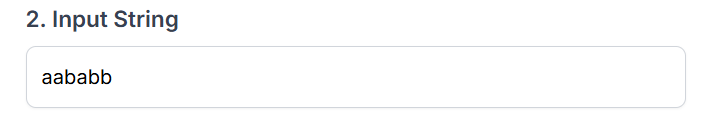
document.getElementById('parse-btn').click();

</script>

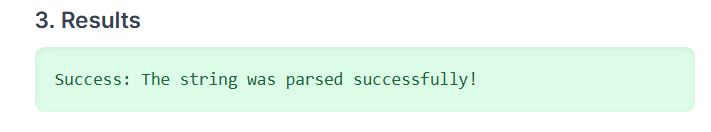
**Appendix : Diagram**



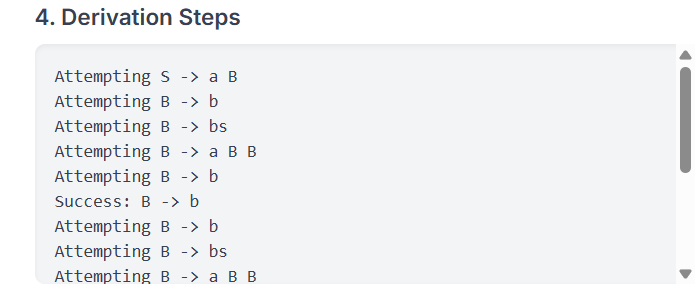
**Figure A.1** : Define Grammar



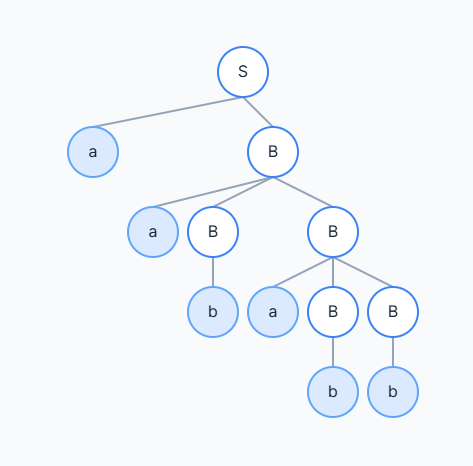
**Figure A.2 :** Input String



**Figure A.3** : Verification



**Figure A.4** : Derivation Steps



**Figure A.5** : Syntax Tree