

Project

Compiler Construction

Section: G3

Group Members

Names	Roll Numbers
Abu Bakar Tahir	L1F21BSCS0503
Muhammad Hammad	L1F21BSCS0514
Hassan Latif	L1F21BSCS1235
Musa Zeeshan Khan	L1F21BSCS1264

Custom Language Table Driven Visual Tokenizer and Parser

1st Abu Bakar Tahir

Faculty of Information and Technology)
University of Central Punjab)
Lahore, Pakistan
L1F21BSCS0503@ucp.edu.pk

2nd Muhammad Hammad

Faculty of Information and Technology)
University of Central Punjab)
Lahore, Pakistan
L1F21BSCS0514@ucp.edu.pk

3rd Hassan Latif

Faculty of Information and Technology)
University of Central Punjab)
Lahore, Pakistan
L1F21BSCS1235@ucp.edu.pk

4th Musa Zeeshan Khan

Faculty of Information and Technology)
University of Central Punjab)
Lahore, Pakistan
L1F21BSCS1264@ucp.edu.pk

Abstract—This paper outlines the design and execution of an extensive, table-driven compiler system that combines lexical and syntactic analysis with real-time visualization. The compiler employs a deterministic finite automata (DFA) for tokenization procedures and an LL(1) parser for syntactic analysis with no backtracking, thus emphasizing on speed, ease of use as well as reliability. The tokenization procedure identifies and registers valid tokens such as operators, keywords, identifiers and numerals, while at the same time systematically recording the errors for debugging purposes. The LL(1) parser accepts a specific context-free grammar (CFG) and context such as left recursion elimination and left factoring to ensure that the syntactic structure is well checked. Panic-Mode has been implemented along with Parse tree generation to ensure error's are dealt with and a proper visual parsing process is shown.

An intuitive interaction with the parsing mechanism is ensured by having an representation of the DFA and parsing tree available to the user at any moment. This facilitates even further the understanding of the internal organization of a compiler and can be used for teaching purposes as well. The system provides two modes of displaying information: one via console and another in file format and it is particularly impressive how the system is able to record errors, tokenized data and other linguistic components and the parse operations in a manageable way. Systematic findings confirm the accuracy of the system as it was able to successfully detect 150 valid tokens and 43 invalid tokens. It also in each occasion provided confirmation of it's correctness and reliability. The use of a non-backtracking architecture guarantees an effective implementation of a design due to minimal computation effort and easy maintenance and extensibility.

In this research, we enhance compiler technology by presenting a new approach for creating more efficient and easy to use language analysis tools. This integrates developed both the practical and the educational aspects of compiler design by providing the vision oriented approaches and integrating them with the table driven ones, setting the level for simplicity, efficiency, and usability in the next generation of compilers design effort.

Index Terms—Compiler design, table-driven parsing, deterministic finite automata (DFA), tokenization, syntax analysis, LL(1) parser, finite state machine (FSM), context-free grammar (CFG), visualization, backtracking-free parsing

I. INTRODUCTION

The proposed work focuses on the development of a new visual interactive compiler [1]. The goal is to help users understand the synthesized process in a better way. This project provides an insight into the designer aspect of the compiler providing interfaces for the otherwise concealed operations such as tokenization and parsing. The principal aim is to focus on the user needs and develop a high quality tool that provides informative illustrations of working creation and manipulation of tokens of a custom programming language. The project implements a no backtracking paradigm which achieves a balance between ease of use implementation efficiency and educational benefits making the project useful for practitioners teachers and students [2], [3].

This endeavor features a strong table-driven tokenization method for lexical analysis. This stage tokenization, which comes first in the compilation pipeline and is also quite essential, restructures the input unprocessed source code into orderly tokens that will be used for further processing. In this project, we make use of deterministic finite automata (DFA) to facilitate effective and deterministic tokenization [4], [5]. The DFA-based architecture scans for language features such as operators, numbers, punctuation, and identifiers successively, thus cutting out backtracking mechanisms that are often sources of complexities as well as performance limitations in typical compilers. Transition tables describe states and define how transitions occur between states, thereby making the software development lifecycle systematic, cost-effective, easily predictable, and efficient as well, features which are quite important for both instructional and real life technologies [6], [7].

This is to point out that there are drawbacks when making the choice of not having backtracking, in that it has been found to make the system easier and more efficient. Backtracking improves parsing in case the grammar is vague or the language is complicated. Compilers are able to recover from

inconsistencies leading to false conclusions by backtracking to an earlier point in order to redefine the accepted conclusions whenever appropriate. However, in order to accommodate such flexibility there is an increase in the consumption of processing resources together with the complexity of implementation [4]. In the course of the project, we have carried out the tokenisation phase without using backtracking and in the process increased speed, ease of use and maintenance which are key requirements of any tool which will be used by people with varying comprehension of how the compiler works [8], [9].

Through the use of LL(1) parsing, there is an increased efficiency and ease that accompanies the parsing step because LL(1) parsing is a top-down syntactic analysis that is well-structured. LL(1) parsing relies on a single look ahead token to make decisions thus avoiding the backtracking and recursion that most methods rely on [4]. Token sequences are efficiently checked against grammar rules by preconstructed tables and on this basis, syntactic constructions that satisfy the requirements of the language can be generated [10], [11]. Nevertheless, some approaches, such as recursive descent parsing with backtracking, allow for more complicated grammars, but add more complication to code and usage of resources. LL(1) parsing is therefore a conscious design to achieve a balance in the level of complexity of implementation and accessibility of the system to its users while ensuring that the system is still operational [3], [12].

What is, perhaps, more interesting is the fact that the project's focus is centered around the advanced visualization feature as it changes the interaction and understanding that a user has during compilation of a program. It is clear from the system how the raw input gets through DFA states, gets tokenized, and thereafter gets to be formed into structured computer syntaxes [2], [13]. This step by step visualization enhances comprehension of theoretical aspects and how they can be implemented practically thus giving consumers a vision of what the compiler is meant to do [14], [15].

The initiative elucidates and engages people by rendering abstract processes visible and interactive, so fostering experimentation, debugging, and innovation with confidence [3]. The interactive quality of the visualization distinguishes it from traditional instructional aids. Users can actively investigate data input methods and navigate DFA states, viewing real-time decision-making and transitions [16], [9]. This involvement enhances understanding and facilitates practical experimentation, whether users are debugging code, trying novel language features, or investigating fundamental compiler concepts. The lack of backtracking improves this experience by facilitating a clear, linear data flow, hence maintaining an uncluttered and intuitive display [17], [18].

The compiler assists programmers and language designers in acquiring new constructs, comprehending grammatical modifications, and identifying errors. The interactive visualisation displays the immediate effects of changes. Facilitates understanding. Interactivity accelerates prototyping and iterative design, facilitating the testing and refinement of

creative concepts [8]. The compiler facilitates experimentation and debugging, rendering it indispensable for programming language design. Efficiency and innovation in the development of language features enhance [19], [20].

The decision not to include a backtracking feature puts emphatic focus on aim of the project- to keep it simple, efficient and involve the users' feedback. Backtracking can be advantageous in a number of cases, but it also adds complexity which could challenge the clarity and effectiveness which are important for the educational aspect of this project. In so doing, the compiler avoids these disadvantages while at the same time offering a robust, easy to maintain tool that will be used for learning purposes and as a working tool [4], [3].

Panic mode is known as an error recovery technique in compilers. It is very useful to cope with syntax errors in an efficient manner during the parsing stage. When a parser runs into an error, it tries to skip some part of the input and wait until it finds a particular character (syntax point), like a semicolon, closing brace, etc. These characters are structurally identifying marks. This method also helps in minimizing cascading errors: errors caused by a major syntax error leading to many more issues. While lingering errors and syntax errors are identified, panic mode ensures some order is preserved in the flow of parsing so that the compiler continues to run. To some extent, it balances the error recovery approach. For this project, panic mode has been applied to enhance the user experience by allowing easy to understand and more accurate localized error handling of the user's mistakes. This way users are able to debug and analyze their programs properly which also contributes to the goals of the educational and interactive aspects of the compiler.

A graphical compiler reaches a higher level of the architecture by considering complex issues and problems together with simplified ones in application. It brings a solution which is efficient, and easy to maintain and it does not only help one to learn but also has the purposes of being an operative tool [2], [13]. Such a compiler changes understanding and use of the compilation procedure by merging coherence with various technicalities. Inasmuch that it has become possible for a graphical compiler to synthesize graphical, interactive displays out of complicated algorithms, this in itself sets a new design standard for compilers. As a result, advances in the art of language processing can be researched and investigated [8], [21].

II. LITERATURE REVIEW/BACKGROUND STUDY

The design of the compiler comes as one of the key concepts in computer science since it connects natural spoken languages with commands that can be understood by machine. This innovative procedure entails the transforming of source code which is understood by humans into simple commands that can be run directly on the computer system. Designing a compiler makes programming easier and increases efficiency as one does not need to know the details of how the hardware is configured. The progress of this field has been steady, with the improvement of performance, understanding, and usability

addressed particularly to education. This section discusses new perspectives on DFA-based tokenization and LL(1) parsing methods together with interactive visualization toolsets and constituent modern libraries for compiler design, mainly from works published between 2020 and 2025 and some older works [21].

A. DFA-Based Tokenization

The first step in the process of compilation is tokenization, which consists of breaking the source code into several components called tokens. Deterministic finite automata (DFA) are the best known solution to this problem since they are able to carry out the lexical analysis in a deterministic manner, without necessitating any form of backtracking. Tilscher and Wimmer analyzed two lexical analyzers based on deterministic finite automata and stressed its ease of use as well as its significance in compiler construction [22] [23]. Qiu et al. [24] extended this idea by building an artificial intelligent driven malicious traffic detection system incorporating multi-faceted transformer based FVF tokenizer. Their findings corroborate the tokenizer's claim about its ability to efficiently and effectively contain large data volumes through scalable thresholding and data flow using dynamic state transition that intelligently distinguishes harmful and useful data in real time [25]. Naik et al have shown how manufacturing query language tokens were generated using DFA parsers and its flexibility and strength has been emphasized [26] [27]. Similarly, Friedman had also explored the tokenization process in computational frameworks [19]. Besides that, work done by Ginsburg and Greibach on deterministic context-free languages seems to have had an influence on the current implementations of DFA, particularly on error recovery and state transition strategies [28] [29] [30].

B. LL(1) Parsing

Parsing is the process of arranging tokens into semantic structures according to the grammar of a particular language, for the purpose of syntax analysis in the context of a compiler design. An LL(1) parser, being a top-down parser which requires only one lookahead token is user-friendly and is good at error detection. Tilscher and Wimmer dealt with the topic of LL(1) parser generators and their practical construction and usage [31]. Hiranishi et al. were able to demonstrate the effectiveness of LL(1) parsing in web visual compilers which proved to be quite effective in the education sector [32]. Kozlov and Svetlakov investigated the ways of more effective work with the simplification of LL(1) grammars observing the increase of the efficiency of parsing and the decrease of the amount of errors made [33]. Further, Abubakar et al. performed a detailed and systemic analysis on various techniques used in the design of compilers, stating that the role of LL(1) parsing is fundamental in both theoretical works and the practice [34]. Visual Interactive tools such as PAVT also further highlights the importance of visualization of LL(1) parsing in pedagogy [35]

C. Visualization in Compiler Education

It is essential to use interactive visualization tools to be able to comprehend compilers especially in an educational context. Ivanova and Kostadinov built an FPGA based system where finite state machines were used as a lexical analyzer which helped in not only teaching concepts but also their application [5], [36]. Stamenkovic and Jovanovic produced educational simulators that were crucial in enhancing students' grasp of compiler theory [31]. This system is further developed by Cummins et al. who devised a number of advanced visualization techniques that simplified the creation and modification of the parser tables. Such tools receive input from users and let them gradually parse through the tables and parsing error and resolution algorithms, hence demystifying complicated actions of the compiler [37]. The visualization strategies discussed by Rodger et al. helps use understand the role of visual aids and automated exercises in teaching of Compiler design through helping in improving teaching of formal languages [38] [39].

D. Role of Libraries in Compiler Design

The improvements are most likely linked to the emergence of modern programming libraries that have simplified, optimized, and increased the reliability of the process of building compilers. Various compiler development phases such as input/output, text processing, and data structure handling employ libraries such as `<iostream>`, `<iomanip>`, `fstream`, `<cctype>`, `<string>`, `<unordered_set>`, `<unordered_map>`, `<fstream>`, `<algorithm>` and `<sstream>`. For example, assist in tokenization of source files while libraries such as aid in parsing its multiple tokens. Integrated development environments (IDEs) for example Visual Studio 2022 ease the work by providing various debugging options and easy integration of libraries development [34]. Cummins et al. pointed out new trends on these libraries in nexus with enhancement of table driven lexical analysis techniques [37]. Tilscher and Wimmer on the other hand showed that recent methods are effective in the development of LL(1) parser generators noting wideranging and greater freedom of implementation [22] [40].

E. Foundations of Compiler Design

The design of the compiler is a very important aspect of computer systems as it allows for high level computer programming languages to be translated into instructions that the computer can understand. In Compiler Design [41], Chatopadhyay and Santanu analyze the critical aspects of the construction of compilers. Their second book extends the discussions on several problems encountered in the optimization of compilers, error detection and its recovery process. Their work serves as both a theoretical paradigm and a manual, tackling such important matters like lexical scanning, syntactic recognition, and the generation of intermediate forms. This large collection is essential for students and experts who intend to understand the basic concepts of how a compiler is constructed.

F. Overview of Compiler Construction Techniques

The work of compiler builders involves a number of techniques with the aim of crafting compilers that are both efficient and multifunctional. In their work ‘An Overview of Compiler Construction’ Abubakar et al. provide a detailed coverage of traditional and modern approaches towards the construction of compiler [25]. They emphasize the importance of using a modular structure which enhances the flexibility and long-term functionality of the compilers. Additionally, their examination explores the roles of formal languages, different methods of parsing, and optimization techniques that increase the performance of a compiler [42]. This paper outlines useful ideas for compiler practitioners that are then used in real-life situations.

This overview combines the achievements and history of compiler technologies by explaining how techniques and tools evolve towards defining the future of compilers. The emphasis on DFA-based tokenization, LL(1) parsing, interactive visualization, and library support indicates the cross disciplinary aspect of this new area of undertaking [43].

III. METHODOLOGY

A. Compiler Design Workflow

The diagram exemplifies the ordered steps involved in using a compiler, focusing on two of its main structural features: the Lexical Analyzer and the Syntax Analyzer (Parser). This structure emphasizes the modular design as well as data-oriented processing which improves efficacy as well as clarity to the operations of the compiler illustrated in Figure. 1.

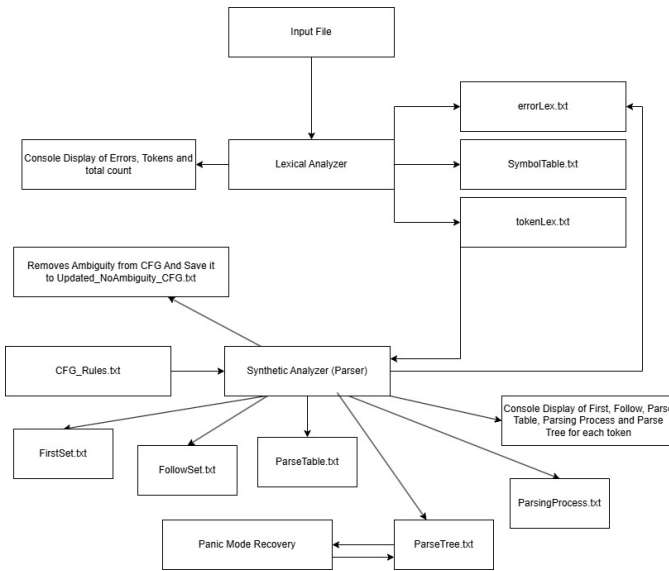


Fig. 1. Compiler Design Workflow

1) *Compiler Architecture and Design Principles:* While designing this compiler, a modular approach was taken in order to improve upon the compilation pipeline. As such, it strives for clarity, efficiency, and extensibility. The Lexical Analyzer and Syntax Analyzer are key parts of the architecture and

are meant to work as separate, but interconnected modules. Modularity aids in debugging because the errors can be confined to specific phases. Furthermore, this approach to design integrates Deterministic Finite Automata (DFA) and LL(1) parsing without backtracking, thus improving speed and reducing complexity as well. Also the panic mode implementation ensures that it can handle errors.

2) *Input and Lexical Analysis:* The process starts with the Input File that contains the source code that should be processed. The Lexical Analyzer takes care of input in order to identify tokens, errors or fill out the Symbol Table. It produces three main outputs which include:

- **error.txt:** It contains detailed description of the lexical mistakes that were spotted during the course of the tokenization process. The errors that occurs in the parser or synthetic analyser are also appended to this file.
- **SymbolTable.txt:** It contains both a logical unit and every identifier, keyword or other symbols together with their attributes.
- **tokenLex.txt:** It contains every token obtained from the input file including the thousands of tokens source file.

Apart from generating those files, the lexical analyzer also gives a brief report on the screen about the errors, tokens and their number for easier debugging.

3) *Syntactic Analysis:* The Syntax Analyzer (or Parser) carries out the processes of tokens using the CFG_Rules.txt (Context-Free Grammar rules) during the phase of syntactical analysis. As a result of this phase, we have several outputs:

- **FirstSet.txt:** The set of terminals that can occur at the beginning of any string derived from a certain non-terminal.
- **FollowSet.txt:** The set of terminals which may come immediately after a non-terminal while deriving it.
- **ParseTable.txt:** A tabular arrangement for the storage of predictive parsing in LL(1) grammars.
- **ParsingProcess.txt:** The record of all the four data structures in compilation and the parsing process of every token.

Thanks to these mechanisms, the parser also has essential tasks to create a visual display of the First and Follow sets, the parse table, and explain in more detail the parsing process, which eases the understanding as well as troubleshooting.

4) *Integration and Output:* The ability of the lexical and the syntactic analyzers to work as one unit makes it easier for one to move from lexical analysis to syntactic verification. Such modularization explains task distribution and aids in debugging by localizing errors in single modules.

B. Deterministic Finite Automata and Transition Tables

Within the process of designing a compiler, Deterministic Finite Automata (DFA) and transition tables are used in the design to assist in locating and classifying various aspects of the source code. These DFAs are aimed to recognize identifiers and keywords, operators and punctuation with the help of tabular rules which are the governing factors for the transitions.

The use of table driven techniques in this context eliminates the need to go back for state changes thereby increasing the speed and efficiency of tokenization. The later parts of this paper will focus on the building and using of these DFAs and transition tables for each category in a bid to ensure the source code is well analyzed and classified.

1) *Identifier*: As shown in Figure 2, this is a DFA that is used within a table driven lexical analyzer specifically for recognition of valid identifiers. The states of this DFA encompasses S0 (initial state), S1 (valid identifier start), S2 (valid identifier continuation), and S3 (valid identifier termination) with a state of DEAD (error state). Depending on the types of the input characters, there are four categories of transitions: L for letters, D for digits, O for others, and - for the end of the input. The DFA enters into state S1 from S0 when a letter

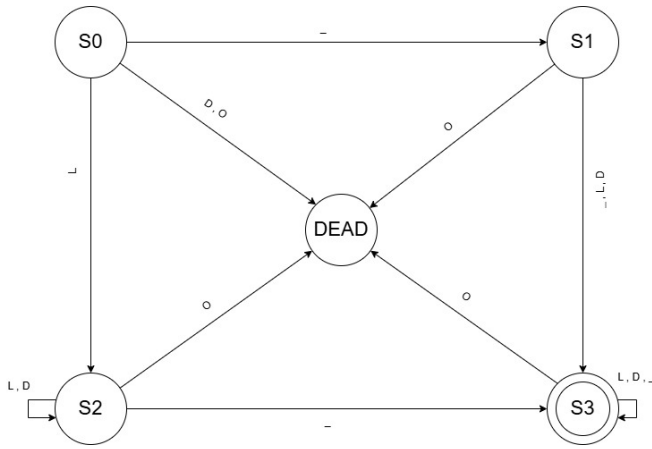


Fig. 2. Finite Automata for Identifier's

is entered, thus starting an identifier. However, S0 can only accept letters, and if it accepts a number or special characters it will move to DEAD state.

TABLE I
STATE TRANSITION TABLE FOR IDENTIFIER

States	L (Letter)	D (Digit)	O (Other)	_ (Underscore)
S0	S2	Dead	Dead	S1
S1	S3	S3	Dead	-
S2	S2	S2	Dead	S3
S3	S3	S3	Dead	S3
Dead	-	-	-	-

This Transition TableI explains how tokens made up of letters (L), digits (D), and characters (O) are accepted. Starting at state S0, S2 is entered when a letter is received and when S2 is active, more letters or digits allow the DFA to stay in a valid state. S1 transforms into S3 if some specific sets of letters and combination with numbers get through. Any input outside of these also causes the DFA to go into the Dead state effectively cancelling out the input. In the case when two or three letters follow in succession creating an accepted sequence of inputs then S2 or S3 is reached which are accepting states since the input is a valid token.

TABLE II
IN CODE STATE TRANSITION TABLE FOR IDENTIFIER

States	L (Letter)	D (Digit)	O (Other)	_ (Underscore)
S0	2	-1	-1	1
S1	3	3	-1	-1
S2	2	2	-1	3
S3	3	3	-1	3
Dead	-	-	-	-

In Table II, we show a DFA that has been encoded such that states are now represented as values for the ease of understanding. The first state S0, which is the starting point, is represented by one, the valid intermediate states S1, S2, S3 news denoted by 2, 3, 4 respectively and Dead state is denoted by -1. Transition appears as once the input category does letters (L), numbers (D) and underscore (_) or any other (O). Thus, as identified in the rules, the DFA moves between these states with identifiers, and when one goes invalid, in case an input is incorrect, one moves to the -1 state, indicating that no more processing shall continue.

2) *Numbers*: A DFA that accepts integer, decimal, and scientific notation of numbers inside a table driven lexical analyzer is illustrated in Figure 3. The input categories that each cause the DFA to transition between states (S0 to S7 and DEAD) are as follows: digits - D, signs - S, dots - ., exponents - E, other invalid characters - O.

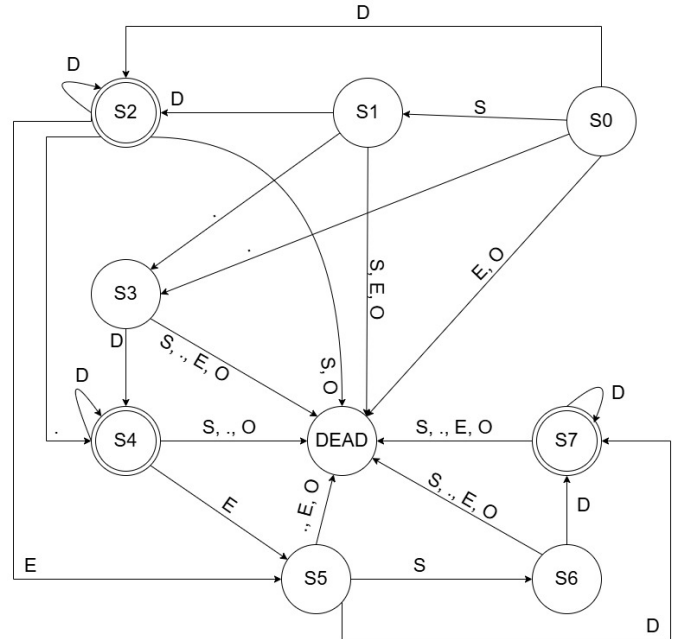


Fig. 3. Finite Automata for Numbers's

We consider as start state the S0 in which the valid numbers start enabling through specific pathways: integers to S1, decimals to S2 then forwarding to S3 or S4, and scientific notation from S5 to S6 then reaching S7. All invalid inputs during a state transition are homologous to DEAD state which leads to a token not being accepted. Even though tokens may

be invalid, compilers still are able to do lexically analysis of the tokens due to the fact that this DFA allows for the clear cut classification of numerical tokens.

This transition table III handles validation of numeric inputs

TABLE III
STATE TRANSITION TABLE FOR NUMBER DFA

States	S (Sign)	D (Digit)	E (Exponential)	. (Dot)	O (Other)
S0	S1	S2	Dead	S3	Dead
S1	Dead	S2	Dead	S3	Dead
S2	Dead	S2	S5	S4	Dead
S3	Dead	S4	Dead	Dead	Dead
S4	Dead	S4	S5	Dead	Dead
S5	S6	S7	Dead	Dead	Dead
S6	Dead	S7	Dead	Dead	Dead
S7	Dead	S7	Dead	Dead	Dead
Dead	-	-	-	-	-

composed of signs (S), digits (D), exponents (E), and the point (.). Form state S0 conditions allow for transitions as determined by the numerical configuration. A digit leads to S2 while a decimal leads to S3. The DFA also incorporates other states like S4 and S7 to control the appearance of the exponents or extra digits within the number So long as I deal with characters which are not defined, the DFA will go to the Dead state. Accepting states like S4 assures that the input conforms to certain well defined numeric standards enough to allow the formatting to be acceptable.

TABLE IV
IN CODE STATE TRANSITION TABLE FOR NUMBER DFA

States	S (Sign)	D (Digit)	E (Exponential)	. (Dot)	O (Other)
S0	1	2	-1	3	-1
S1	-1	2	-1	3	-1
S2	-1	2	5	4	-1
S3	-1	4	-1	-1	-1
S4	-1	4	5	-1	-1
S5	6	7	-1	-1	-1
S6	-1	7	-1	-1	-1
S7	-1	7	-1	-1	-1

This in-code Table IV controls the working of the DFA for integers and floating-point numbers. State S0 goes to S1 to allow a sign, S2 for some digits, and to S3 for a decimal point. Upon reading some digits or an exponent (E), the DFA goes into a number of approved states. The exponent part is taken care of by states S5 and S6 while S7 is the accepting state for numbers. There are also transitions where numeric components could be supplied, for example, alphanumeric ones which leads to a -1 rejection.

3) *Punctuation*: This DFA was designed to validate punctuation letters in Figure 4. This figure consists of three states: S0, which is the start state, S1, and DEAD. The DFA shifts from S0 to S1 only in the event of receiving valid punctuation characters [, , ;, :, , or]. All other inputs including O or the wrong character force the change to a DEAD state where every other input is rendered unimportant.

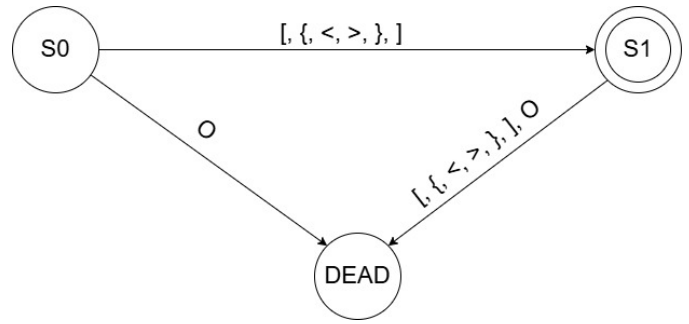


Fig. 4. Finite Automata for Punctuations

The goal of this DFA is to determine if a given input string is made solely of the specified acceptable punctuation letters. As soon as invalid characters are scanned by it, the DFA changes into the DEAD state, and hence, it ensures the rejection of that string. This system provides assistance for the straightforward syntax checking in situations where strong character checking is needed.

TABLE V
STATE TRANSITION TABLE FOR PUNCTUATION

States	[]	{	}	;	:	O
S0	S1	S1	S1	S1	S1	S1	Dead
S1	Dead	Dead	Dead	Dead	Dead	Dead	Dead
Dead	-	-	-	-	-	-	-

This Transition Table V describes how characters including brackets ([,]), braces (,), angle brackets (;, :), and other unspecified characters are handled. It begins at S0 state, as soon as a character [is entered an input recognition success results and a transition occurs to state S1. All characters that are not acceptable result to a transition to DEAD state thus no further action is taken as the input is wrong. From S1, inputs that are accepted lead the DFA in given states, all disallowed characters however lead the DFA to DEAD state.

TABLE VI
IN CODE STATE TRANSITION TABLE FOR PUNCTUATION

States	[]	{	}	;	:	O
S0	1	1	1	1	1	1	-1
S1	-1	-1	-1	-1	-1	-1	-1

This Table VI provides details of a DFA in which the states have been numbered for the purpose of explanation. The first state of the machine S0 is represented by 1. Punctuation characters such as [, , ;, :, , and] result in a transition to state S1, else any other input leads to the Dead state (-1) terminating any further computation. S1 is used as a check of punctuation - that is an accepted state of punctuation, however, if Punctuation is misplaced resulting in a transition to -2 results, no further computation is done.

4) *Operators*: This Figure 5 aims to recognize operator tokens which may be mathematical or logical in context. The figure indicates a Deterministic Finite Automaton (DFA)

which commences its operation at S0 and thereafter spreads towards several other states, which are S1 to S13, each of which represents an acceptable combination of operators such as '!', 'i', '=', '+', '(', '*', '/', '&', '—', '%', etc. The moment an operator is followed by any non-acceptable character, the automaton conditionally transfers to a DEAD state and henceforth remains indifferent to any further input.

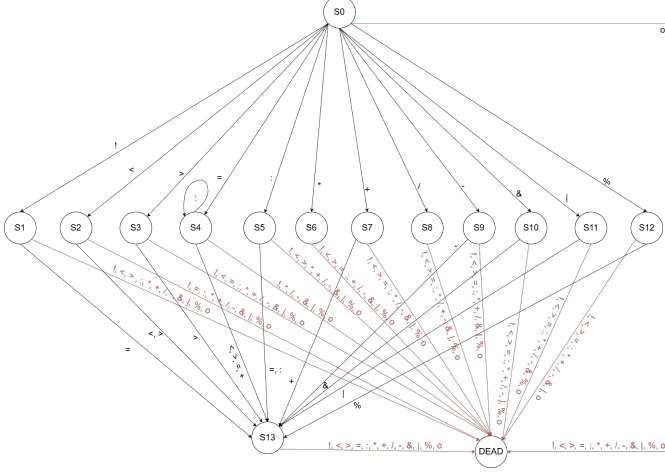


Fig. 5. Finite Automata for Operators

Each state (e.g., S1 to S13) represents either partial or complete validation of an operator. The DFA guarantees acceptance just of prescribed operator sequences while rejecting incorrect combinations. This framework is advantageous for authenticating operator tokens during lexical analysis in compilers or parsers.

TABLE VII
STATE TRANSITION TABLE FOR OPERATORS

States	!	i	i	=	:	*	+	/	-	&	—	%	O
S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Dead
S1	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S2	Dead	S13	S13	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S3	Dead	Dead	S13	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S4	Dead	S13	S13	S13	S4	Dead	S13	Dead	Dead	Dead	Dead	Dead	Dead
S5	Dead	Dead	Dead	S13	S13	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S6	Dead	Dead	Dead	Dead	Dead	S13	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S7	Dead	Dead	Dead	Dead	Dead	S13	S13	Dead	Dead	Dead	Dead	Dead	Dead
S8	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead
S9	Dead	Dead	Dead	Dead	Dead	Dead	Dead	S13	Dead	Dead	Dead	Dead	Dead
S10	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	S13	Dead	Dead	Dead	Dead
S11	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	S13	Dead	Dead	Dead
S12	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	S13	Dead	Dead
S13	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	Dead	S13	Dead
Dead	-	-	-	-	-	-	-	-	-	-	-	-	-

This Transition Table VII verifies for the presence of operators such as !, i, i, =, :, and also for the compound forms, for example !=, i=, i=. Starting from S0, certain operators proceed to their specific states (e.g. ! opens up S1). Compound states handle the compound operators, for example, S13 represents the not equal to sign. The illegal transitions cause the DFA to move into the Dead state and further processing is suspended. The structure ensures that only appropriate operators or their combinations are recognized while the final states are used to check the correctness.

In this section Table VIII deals with DFA Transitions especially for the operators !, i, i, = and so forth. The transitions of the initial state S0 into the other states which are

TABLE VIII
IN CODE STATE TRANSITION TABLE FOR OPERATORS

States	!	i	i	=	:	*	+	/	-	&	—	%	O
S0	1	2	3	4	5	6	7	8	9	10	11	12	-1
S1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S2	-1	13	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S3	-1	-1	13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S4	-1	13	13	13	4	-1	13	-1	-1	-1	-1	-1	-1
S5	-1	-1	-1	13	13	-1	-1	-1	-1	-1	-1	-1	-1
S6	-1	-1	-1	-1	-1	13	-1	-1	-1	-1	-1	-1	-1
S7	-1	-1	-1	-1	-1	-1	13	-1	-1	-1	-1	-1	-1
S8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
S9	-1	-1	-1	-1	-1	-1	-1	-1	13	-1	-1	-1	-1
S10	-1	-1	-1	-1	-1	-1	-1	-1	-1	13	-1	-1	-1
S11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	13	-1	-1
S12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	13	-1
S13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Dead	-	-	-	-	-	-	-	-	-	-	-	-	-

the transition states of the operator go to the special designated operator states. For example, the state S1 is assigned the != operator whereas the states S3 and S5 are in charge of other combinations like i i or ==. In the event of invalid input being encountered, the impossible or the dead state of the DFA is inactive because it has changed to the Dead state -1. In this table, the transitions of the DFA for the operators including but not limited to the three mentioned: !, i, i are also given. The transitions of the initial state S0 into the other states which are the transition states of the operator go to the special designated operator states. For example, the state S1 is assigned the != operator whereas the states S3 and S5 are in charge of other combinations like i i or ==. In the event of invalid input being encountered, the impossible or the dead state of the DFA is inactive because it has changed to the Dead state -1.

C. Other Token Type

In such ways, other token types, for example keywords and errors, can be organized in a straightforward and efficient manner which will result in better performance and maintainability of the code.

1) *Keywords*: The unordered_set is employed for keyword searching. Such a method allows the organization, retrieval, and comparison of keywords with no requirement of FA, or even worse, table lookups which can be more expensive in terms of processing time and efficiency. The use of this improved data structures of unordered_set significantly improves the speed and expansion level of keyword spotting, making it applicable for conditions with large numbers of keywords or a lot of searches liable to be carried out.

2) *Errors*: There is an error if a character, a string or a symbol cannot be classified properly under any particular token type. This feature promotes tokenization since it ensures that all characters, strings or symbols which are unknown or invalid are properly encoded for the purpose of error control and debugging.

D. Syntax Analysis (Synthetic Section)

This section focuses on the syntactic analysis using a specially designed context-free grammar for the targeted task. The CFG undergoes certain, changing procedures to ensure their effectiveness and correctness. Such procedures include

elimination and left recursion together with left factoring. After confirming the correction of the CFG components, the First and Follow sets for every non-terminal symbol are constructed. A tabular structure, which is called a parser table, is constructed from the sets in order to allow for effective parsing. Then the processes continues with the stack or subroutine oriented processing of the input tokens. This also aims at checking the syntax of the tokenized input against that of the grammatical rules so that the expected form and the appropriate CFG are observed.

1) *Implementation of Panic Mode:* To manage the syntax errors that may occur at the stage of syntactic analysis, panic mode has been integrated into the parser. The parser, after an error, skips the insignificant part of the input until it reaches a synchronization point such as a semicolon or closing brace. This allows the parsing process to be completed without the concern of subsequent errors in the other steps. The incorporation of the panic mode helps achieve the dual objectives of stability of the parsing scheme and advanced user interface design through simple, effective error marking. This function is important for the didactic purpose of the project as it enables users of differing knowledge and skills to accurately diagnose an error and correct it.

Advantages: The project stands to gain with the aid of panic mode error recovery as it allows compilation with a high degree of flexibility, functionality and readability. One of the evident pros would be its capability of resolving syntax errors without cascading failures, which is when the former misleads the remaining code and gives a plethora of error messages. Allowing the parser to operate and furthering the rest of the code to be usable by jumping to already set synchronization points such as colon or closing braces is the unique mode of panic. This reduces the focus required while the capsule is in use and allows the user to concentrate on finding the real errors instead of being bombarded with counterproductive messages. Further to this, panic mode as a whole allows for greater mixing with the pedagogic and interactive aims of the project as it provides sufficient localized feedback for effective debugging and teaching. With the mode enforced, a user gets to see how the compiler processes and solves issues, helping them better understand how error recovery and parsing works. Further, the panic mode assists in the general performance of the system while decreasing the amount of work the system has to do because other methods of error recovery like the backtracking is much more complicated. This allows the tool to be simple enough for a novice or a professional user to use it as a teaching and debugging tool.

2) *Visualization Integration:* The main goal of the visualization feature is to give an engaging and rich overview of the entire compiling process using step by step instructions to guide users on how to tokenize, manage the DFA (Deterministic Finite Automaton) state transitions, and finally syntax parse using LL(1) parsing tables. One of its primary functions is to help users comprehend advanced computing theories like multi-level reinforced parsing by portraying difficult abstract concepts through simple visuals. Users witness the transfor-

mation of raw source from tokens into syntactically accurate parse trees which gives invaluable insight into a compiler's inner workings. Also, the visualization can be beneficial for feedback intake and offer solutions to error handling problems. The tutorial on modulation shows how parsing errors can be found, dealt with, and fixed as the system goes into the "panic mode" which facilitates error recovery learning. This tool allows for more in-depth analysis because users do not only build their theoretical foundation but also broaden their problem solving skills by considering the whole compilation and interpretation of source code.

E. Output

The analysis of every vocabulary and grammar rule has a record which is in a form of pictures and files for easy access in the future. The subsequent files have been generated:

- TokenLex.txt: Contains the enumeration of tokens discerned during the lexical analysis stage.
- error.txt: Keeps account of all unknown tokens or errors encountered in the tokenization phase. Also contains errors caused in the synthetic analysis phase, which are appended after the errors in lexical analysis phase
- SymbolTable.txt: Keeps a record of different symbols and what they depict.
- ParseProcess.txt: Provides a description of how the input tokens were processed in the chronological order.
- ParseTable.txt: Displays the produced parse table as a result of context free grammar.
- FirstSet.txt:
- FollowSet.txt: Lists all the non-terminal Symbols and their corresponding Follow sets that were computed.

The text files are otherwise known in the industry as the narratives gives most of the information necessary to understand how the processes of tokenization, error and parsing works which in return will be able to help in the analysis, debugging, and finally the verification of the operation of the system.

IV. RESULTS

Our experiment records testify the efficacy and correctness of our compiler approach which is based on table driven techniques for implementing scanning and LL(1) techniques for parsing. The process begins from examining the input file that includes a number of program's components such as operators, numbers, identifiers, and punctuation signs. Such a system ensures the fast and unique resolution of tokens by the use of predetermined state diagrams of DFAs and no backtracking is needed. Tokens are distributed into a number of classes including operators, keywords, identifiers, and integers while non-words are retained for later checks. The result of the tokenization process along with its incorrectness is uploaded onto the console almost instantaneously enabling the user to provide feedback in real time without waiting too long.

The general strategy is quite simple, since it relies on the systematic application of the DFA state tables for each input symbol. It does not use back tracking and therefore simplifies

the issues related to the design of the tokenization process. Once the tokens have been found, the next stage is the choice of an LL(1) parser for the synthetic analysis in which the forms of the language obtained syntactically correspond to a grammar of some language which has been implemented. The table-driven approach to parsing also overcomes restrictions which recursive descent parsers are subjected to and this improves speed and performance efficiency.

Despite the efficiency of the system, there are some errors that occur during tokenization. Malformed tokens such as incorrectly constructed operators or outlines of identifiers have been detected, and they, along with their positions in the source code, have been registered in an error file. Focusing spreadsheets on valid analysed tokens, a score of 150 has been determined, while 43 rounded off invalid tokens indicating the accuracy and range of the lexical exceptional procedure.

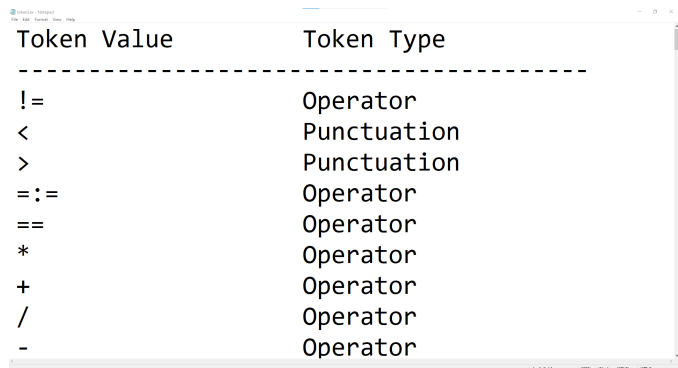
Also, there are some inputs that should have successfully parse according to Context Free Grammar (CFG) but doesn't pass due to how they are handled in the code. Such input tokens are then handled by Panic Mode recovery so that no error is caused by them. The Inputs that don't parse successfully through the parse table are recorded in the error.txt file. The error.txt file contains the errors that were discovered in lexical analysis and errors that are appended in this synthetic analysis phase. Despite all this, most of the inputs given would successfully parse. The inputs that are having problem in parsing are variable names along with digits of size 3 and more.

To facilitate users and make the ledgers user friendly, real time visualisation tools are decked into interfaces which allow users to see the transitions of the states of DFAs and also to see the parsing process live. The graphical representation in this way gives an easy and simple definition of classification of tokens and the functions of the parser and as such it becomes very useful in matters of learning and debugging. This section will include images and data outputs so that tokenization and parsing results will be well explained. The use of table driven techniques for tokenization coupled with LL(1) parsing increases the speed of execution and eases the design and maintenance of the compiler which makes it a good and powerful tool in compiler construction.

A. Output Figures

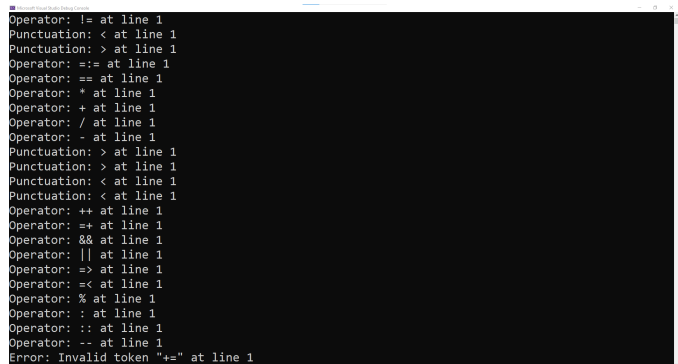
Figures 6- 17 provide a comprehensive view of the outcomes of our project. These figures provide visual confirmation of the outputs reported on the console as well as many other files that were created during the work of the compiler. The active window feedback provides information on the tokens and errors that have been processed and located. The reports, on the other hand, provide detailed and permanent evidence of the information suitable for future considerations [44].

Figures 6 and Figure 7 show the contents of TokenLex.txt and what has been produced by the console which is also errors thereby showing all the tokens that were detected during the process of analyzing the source code in detail.



Token Value	Token Type
!=	Operator
<	Punctuation
>	Punctuation
==	Operator
==	Operator
*	Operator
+	Operator
/	Operator
-	Operator

Fig. 6. Text File Output for Tokens



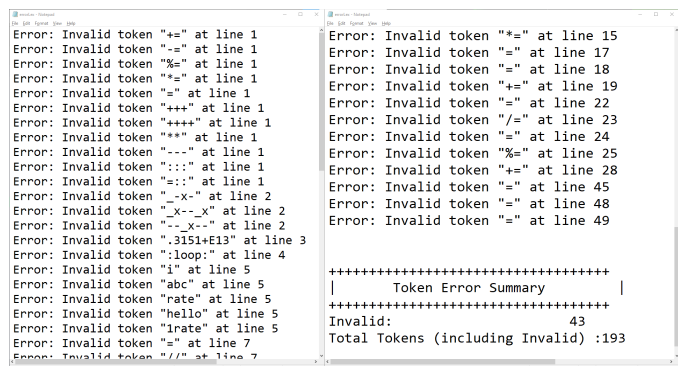
```

Operator: != at line 1
Punctuation: < at line 1
Punctuation: > at line 1
Operator: == at line 1
Operator: == at line 1
Operator: * at line 1
Operator: + at line 1
Operator: / at line 1
Operator: - at line 1
Punctuation: > at line 1
Punctuation: > at line 1
Punctuation: < at line 1
Punctuation: < at line 1
Operator: ++ at line 1
Operator: += at line 1
Operator: && at line 1
Operator: || at line 1
Operator: => at line 1
Operator: =< at line 1
Operator: % at line 1
Operator: : at line 1
Operator: :: at line 1
Operator: -- at line 1
Error: Invalid token "+=" at line 1
  
```

Fig. 7. Console Output for Tokens and Errors

These tokens include standard operands, keywords, symbols, numbers and punctuation marks where each of the tokens created is distinguished and defined by its attributes such as type and place. Figure 8 shows error.txt in which tokens that were not recognized or errors that occurred at the stage of recognition of symbols and formats of ibid such as identifiers and operators, together with the corresponding number of lines in the source text. This kind of classification of mistake aids the user to locate and correct the mistakes.

This parsing proceeds with Figure 9 and Figure 10, which is the SymbolTable.txt and in which symbols such as the



```

Error: Invalid token "+=" at line 1
Error: Invalid token "-=" at line 1
Error: Invalid token "%=" at line 1
Error: Invalid token "*=" at line 1
Error: Invalid token "=" at line 1
Error: Invalid token "++" at line 1
Error: Invalid token "+++" at line 1
Error: Invalid token "+++" at line 1
Error: Invalid token "***" at line 1
Error: Invalid token "---" at line 1
Error: Invalid token ":::" at line 1
Error: Invalid token "==" at line 1
Error: Invalid token "-X-" at line 2
Error: Invalid token "-X-" at line 2
Error: Invalid token "-X-" at line 2
Error: Invalid token "3151+E13" at line 3
Error: Invalid token ":loop:" at line 4
Error: Invalid token "i" at line 5
Error: Invalid token "abc" at line 5
Error: Invalid token "rate" at line 5
Error: Invalid token "hello" at line 5
Error: Invalid token "irate" at line 5
Error: Invalid token "=" at line 7
Error: Invalid token "/" at line 7

Error: Invalid token "+=" at line 15
Error: Invalid token "=" at line 17
Error: Invalid token "=" at line 18
Error: Invalid token "+=" at line 19
Error: Invalid token "=" at line 22
Error: Invalid token "/=" at line 23
Error: Invalid token "=" at line 24
Error: Invalid token "%=" at line 25
Error: Invalid token "+=" at line 28
Error: Invalid token "+" at line 45
Error: Invalid token "=" at line 48
Error: Invalid token "=" at line 49

+++++
|      Token Error Summary      |
+++++
Invalid: 43
Total Tokens (including Invalid): 193
  
```

Fig. 8. Text File Output for Errors

Token Value	Token Type	Line No	Token No
!=	Operator	1	0
<	Punctuation	1	1
>	Punctuation	1	2
==	Operator	1	3
==	Operator	1	4
*	Operator	1	5
+	Operator	1	6
/	Operator	1	7
-	Operator	1	8
>	Punctuation	1	9
>	Punctuation	1	10
<	Punctuation	1	11
<	Punctuation	1	12

Fig. 9. Text File Output for Symbol Table (Part 1)

==	Operator	50	147
5	Number	50	148
}	Punctuation	51	149

Token Count Summary	
Keywords:	21
Identifiers:	29
Numbers:	32
Punctuations:	35
Operators:	33
Invalid:	43
Total Tokens (Valid):	150

Fig. 10. Text File Output for Symbol Table (Part 2)

name of the variables, data types, and relevant features are stored for reference during parsing. The Figures 11 and 15 depict the step by step parsing technique explanation found in the ParseProcess.txt and the created parser table from the ParseTable.txt, respectively. These outputs indicate the LL(1) parsing procedures and describe the table-driven implementations of the parser assuring the right set of grammar's rules are followed for the tokenized input stream.

Parsing line 3: !=		
Stack	Input	Action
<program> \$!= \$	Expand: <program> -> <operator>
<operator> \$!= \$	Expand: <operator> -> !=
!= \$!= \$	Match: !=
\$	\$	Match: \$
Input successfully parsed.		
Parsing line 4: <		
Stack	Input	Action
<program> \$	< \$	Expand: <program> -> <punctuation>
<punctuation> \$	< \$	Expand: <punctuation> -> <
< \$	< \$	Match: <
\$	\$	Match: \$
Input successfully parsed.		
Parsing line 5: >		
Stack	Input	Action
<program> \$	> \$	Expand: <program> -> <punctuation>

Fig. 11. Text File Output for Input's Parsing Process in Stack

```

FIRST(<expression>) = { [a-zA-Z] _ [+]? [ ] }
FIRST(<declaration>) = { default int float true signed short protected bool char double long export void
return class magari loop static goto agar while typename try typeid false break struct namespace public
asm register else sizeof operator this new switch auto enum throw explicit union const catch private
case using extern continue typedef virtual inline do for friend volatile if unsigned delete mutable }
FIRST(<program>) = { float true [ ] [a-zA-Z] [+]? / default int unsigned if _ [ loop signed short
protected bool char double export long class return void magari < static agar goto try while typename
break false typeid register public asm namespace struct else operator sizeof this switch new auto enum
throw explicit const union private catch case using extern typedef continue virtual do inline for &&
volatile friend delete mutable = - != * : > + % [ > ] } }
FIRST(<start_identifier>) = { [a-zA-Z] _ }
FIRST(<loop>) = { loop }
FIRST(<term>) = { [a-zA-Z] [+]? _ [ ] }
FIRST(<arguments>) = { [a-zA-Z] [+]? _ [ ] }
FIRST(<greaterOp>) = { > < & }
FIRST(<identifier>) = { [a-zA-Z] _ }
FIRST(<letter>) = { [a-zA-Z] }
FIRST(<number>) = { [+]? }
FIRST(<statements>) = { unsigned if [+]? default int loop signed short true float [a-zA-Z] protected
bool char double long export void return class magari static goto agar while typename try typeid false
break struct namespace public asm register else sizeof operator this new switch auto enum throw explicit
union const catch private case using extern continue typedef virtual inline do for friend volatile
delete mutable [ ] }
FIRST(<PlusOp>) = { + & }
FIRST(<statements>) = { [+]? default int unsigned if loop signed short [a-zA-Z] float true char bool
protected double export long class return void magari static agar goto try typename while break false

```

Fig. 12. Text File Output for First Set

```

FOLLOW(<arguments>) = { $ [ ] }
FOLLOW(<expression>) = { $ <rest_arguments> [ ] : }
FOLLOW(<rest_expression>) = { $ [ ] <rest_arguments> : }
FOLLOW(<identifier>) = { [a-zA-Z] _ [+]? [ ] }
FOLLOW(<greaterOp>) = { $ [a-zA-Z] _ [+]? [ ] }
FOLLOW(<start_identifier>) = { _ [a-zA-Z] 2 1 3 4 5 6 7 8 9 0 & }
FOLLOW(<identifier_tail>) = { $ < [a-zA-Z] _ [+]? [ ] }
FOLLOW(<rest_identifier_tail>) = { [a-zA-Z] _ [+]? [ ] }
FOLLOW(<letter>) = { 2 [a-zA-Z] 1 3 4 5 6 7 8 9 0 & }
FOLLOW(<number>) = { [a-zA-Z] _ [+]? [ ] }
FOLLOW(<digit>) = { 2 [a-zA-Z] 1 3 4 5 6 7 8 9 0 & }
FOLLOW(<MinusOp>) = { $ [a-zA-Z] _ [+]? [ ] }
FOLLOW(<colonOp>) = { $ [a-zA-Z] _ [+]? [ ] }
FOLLOW(<punctuation>) = { $ }
FOLLOW(<types>) = { $ export long loop true float [a-zA-Z] int [+]? default _ if unsigned signed short
protected bool char double void return class magari static goto agar while typename try typeid false
break struct namespace public asm register else sizeof operator this new switch auto enum throw explicit
union const catch private case using extern continue typedef virtual inline do for friend volatile
delete mutable [ ] }
FOLLOW(<keywords>) = { loop export long true float [a-zA-Z] default [+]? int _ unsigned if signed
short protected bool char double class return void magari static agar goto try while typename break false
typeid register public asm namespace struct else operator sizeof this switch new auto enum throw
explicit const union private catch case using extern typedef continue virtual do inline for volatile
friend delete mutable [ ] }

```

Fig. 13. Text File Output for Follow Set

```

Grammar Contents:
<program> -> <statements> | <function_call> | <statements> | <loop> | <conditional> | <expression> | <declaration> | <assignment>
| <return_statement> | <term> | <identifier> | <number> | <operators> | <punctuation> | <type>
<declaration> -> <type> <identifier>
<conditional> -> if [ <expression> ] { <statements> } [ else { <statements> } ]
<term> -> <identifier> | <number> | [ <expression> ]
<loop> -> loop { <expression> } { <statements> }
<statements> -> <statement> <statements>
<assignment> -> <identifier> = <expression>
<return_statement> -> return <expression>
<function_call> -> <identifier> [ <arguments> ]
<arguments> -> <expression> <rest_arguments>
<rest_arguments> -> [ <expression> <operator> <expression> ]
<start_identifier> -> <letter> | <letter> <rest_identifier_tail> | <digit> <rest_identifier_tail>
<letter> -> [a-zA-Z]
<digit> -> [0-9]
<rest_identifier_tail> -> <identifier_tail> | <letter> <rest_identifier_tail> | <digit> <rest_identifier_tail>
<operator> -> + - * / % <greaterOp> > < < PlusOp > & % % : colonOp ||
<punctuation> -> [ ] { } ; , < > &
<type> -> int | bool | float | short | char | double | long | void | <keyword>
<keyword> -> magari | static | loop | try | agar | double | register | break | asm | false | else | operator | this | switch | new
| auto | enum | void | class | throw | bool | explicit | const | private | true | export | protected | case | using | extern |
public | typedef | virtual | catch | do | for | typeid | volatile | char | signed | float | goto | typename | return | namespace
| inline | continue | union | friend | short | unsigned | if | sizeof | default | delete | while | int | long | struct | mutable
No left factoring detected.
No left recursion detected.

```

Fig. 14. Console Output for Synthetic Analysis (Part 1)

ParseTable - Notepad												
File Edit Format View Help												
Non-Terminal	!=	\$	%	&&	*	+	-	/	0	1	2	
mutable	namespace	new	operator	private	protected	public	register	return	short	signed	sizeof	
<program>	<program>	sync	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>
<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>	<program>
<declaration>	-	sync	-	-	-	-	-	-	-	-	-	-
ifier> :	<type>	<identifier>	:	<type>	<identifier>	:	<type>	<identifier>	:	<type>	<identifier>	:
<MinusOp>	-	sync	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<statements>	-	sync	-	-	-	-	-	-	-	-	-	-
ements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>
atements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	<statement>	<statements>	-	sync
<term>	sync	sync	sync	sync	sync	sync	sync	sync	sync	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<loop>	-	sync	-	-	-	-	-	-	-	-	-	-
ements> }	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync
<keyword>	-	sync	-	-	-	-	-	-	-	-	-	-
mutable	namespace	new	operator	private	protected	public	register	return	short	signed	sizeof	
<conditional>	-	sync	-	-	-	-	-	-	-	-	-	-
sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync
<statement>	-	sync	-	-	-	-	-	-	-	-	-	-
<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>	<statement>
<PlusOp>	-	sync	-	-	-	<PlusOp>	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<program>	<operator>	sync	<operator>	<operator>	<operator>	<operator>	<operator>	<operator>	<operator>	<digit>	<digit>	<digit>
<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>	<type>
<assignment>	-	sync	-	-	-	-	-	-	-	-	-	-
sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync
<return_statement>	-	sync	-	-	-	-	-	-	-	-	-	-
sync	sync	sync	sync	sync	sync	sync	sync	sync	return <expression> :	sync	sync	!
<function_call>	-	sync	-	-	-	-	-	-	-	-	-	-
sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync	sync
<equalOpRest>	-	sync	-	-	-	<equalOpRest>	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<arguments>	-	sync	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<expression>	-	sync	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-
<rest_expression>	<rest_expression>	sync	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	<rest_expression>	-

Fig. 15. Text File Output for Parse Table

[illegible]

```

Parsing line 150: ==
Stack      Input      Action
<program> $      == $      Error: No rule for '<program>' with token '=='. Entering Panic Mode.
$          $      Match: $

Parsing failed.

Parse Tree:
|====> <program>

Parsing line 151: 5
Stack      Input      Action
<program> $      5 $      Expand: <program> -> <program'*
<program>' $      5 $      Expand: <program'* -> <digit>
<digit> $      5 $      Expand: <digit> -> <digit'*
<digit>' $      5 $      Expand: <digit>' -> 5
$ $      5 $      Match: 5
$ $      $      Match: $

Input successfully parsed.

Parse Tree:
|====> <program>
|====> <program'*
|====> <digit>
|====> <digit'*
|====> 5

```

The Figure12 and Figure13 corresponds to the documents FirstSet.txt and FollowSet.txt of the First and Follow sets computation for the non-terminals. The most relevant aspect regarding these sets is their role in the LL(1) parser table construction and hence in achieving correctness of the parsing of language syntax. Finally, console outputs of the parser function are depicted in Figures 14, Figure 16, and Figure 17, which are the outputs displayed by the console where the information and outputs include the grammar of the input, left recursion elimination, and left factoring. It also shows how to calculate the First and Follow sets of the grammar, as well as the parsing process itself, along with parse tree. In the parsing process, Panic Mode implementation can be seen.

V. CONCLUSION

One of the advantages of the compiler is that it is user-friendly, efficient, and maintainable; it provides a straightforward and quick approach to lexical and syntactic analysis making all of the traditional backtracking problems vanish. This architecture is a major contribution to the evolution of compiler construction systems by providing an alternative view of solving the problem of the accuracy of educational and practical embedding of the systems and their simultaneous solutions.

the visualization coupled with the table driven techniques to build a compiler that is technologically sophisticated and easy to work with. The deployment of full sentential or text phrase structure grammar analysis framework with the aid of LL(1) parsing based compiler technology makes all the process more organized, effective and non-duplicative hence realizing the optimal architecture of the compiler. The system assists to more comprehend language analytical processes and the processes of transforming language into code because it emphasizes on issues of clarity of the context, how the content is performed, and how the content is taught. It acts as a powerful instrument for educational and practical applications thereby paving the way for the future advancements in compiler development. This paper deals with the issues which have arisen from the historical perspective and focuses on the seeking out the problems of constructions of a compiler which are simple and effective.

REFERENCES

- [1] R. del Vado Vírveda, "An interactive tutoring system for learning language processing and compiler design," in Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, 2020.
- [2] R. del Vado Vírveda, "Visualizing Compiler Design Theory from Implementation Through an Interactive Tutoring Tool: Experiences and Results," in Proceedings of the 15th International Conference on Computer Supported Education (CSEDU 2023), pp. 333-340, 2023. [Online]. Available: <https://www.scitepress.org/Papers/2023/117098/117098.pdf>. [Accessed: Jan. 2, 2025].
- [3] Srećko Stamenković, Nenad M Jovanovic, Pinaki Chakraborty, "Evaluation of simulation systems suitable for teaching compiler construction courses," Researchgate.net, Mar-2020. [Online]. Available: https://www.researchgate.net/publication/340346717_Evaluation_of_simulation_systems_suitable_for_teaching_compiler_construction_courses. [Accessed: 01-Jan-2025].
- [4] N. Xie, "flap: A Deterministic Parser with Fused Lexing," Proceedings of the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2023), pp. 123-135, 2023. [Online]. Available: <https://xnning.github.io/papers/pldi23flap.pdf>. [Accessed: Jan. 2, 2025].
- [5] A. Ivanova and N. Kostadinov, "An approach to introduce the concept of lexical analysis through FPGA based finite state machines," in Proceedings of the 24th International Conference on Computer Systems and Technologies, 2023, pp. 79-84.
- [6] V. Lindmark, "Analyzing the effect of DFA compression in token DFAs," Diva-portal.org. [Online]. Available: <https://umu.diva-portal.org/smash/get/diva2:1878423/FULLTEXT01.pdf>. [Accessed: 01-Jan-2025].
- [7] K. Van Nguyen, K. V. Tran, S. T. Luu, A. G.-T. Nguyen, and N. L.-T. Nguyen, "Enhancing lexical-based approach with external knowledge for Vietnamese multiple-choice machine reading comprehension," IEEE Access, vol. 8, pp. 201404-201417, 2020.
- [8] M. Jovanović and M. Stamenković, "ComVIS—Interactive Simulation Environment for Compiler Learning," Computer Applications in Engineering Education, vol. 29, no. 6, pp. 1635-1650, 2021. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/cae.22456>. [Accessed: Jan. 2, 2025].
- [9] R. del Vado Vírveda, "ITT: An interactive tutoring tool to improve the learning and visualization of compiler design theory from implementation," in Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2, 2022.
- [10] Y. Xu and Y. Zhu, "A survey on pretrained language models for Neural Code Intelligence," arXiv [cs.SE], 2022.
- [11] R. del Vado Vírveda, "Learning compiler design: From the implementation to theory," in Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2, 2021.
- [12] V. Subramanian, K. Karthikeyan, and P. Venkataram, "A concept map based teaching of Compiler Design for undergraduate students," ICST Trans. e-Educ. e-Learn., vol. 8, no. 1, p. e4, 2022.
- [13] M. Mernik and V. Zumer, "An educational tool for teaching compiler construction," IEEE Trans. Educ., vol. 46, no. 1, pp. 61-68, 2003.
- [14] I. G. S. Yasa, U. Lampung, H. Yufrizal, N. Nurdiana, U. Lampung, and U. Lampung, "Improving students' reading comprehension through visualization strategy," U-Jet Unila Journal of English Language Teaching, vol. 11, no. 2, 2022.
- [15] J. J. Castro-Schez, C. Glez-Morcillo, J. Albusac, and D. Vallejo, "An intelligent tutoring system for supporting active learning: A case study on predictive parsing learning," Inf. Sci. (Ny), vol. 544, pp. 446-468, 2021.
- [16] H. Shaziya and R. Zaheer, "Strategies to effectively integrate visualization with active learning in computer science class," in Proceedings of International Conference on Computational Intelligence and Data Engineering, Singapore: Springer Singapore, 2021, pp. 69-81.
- [17] S. Stamenković and N. Jovanović, "A web-based educational system for teaching compilers," IEEE Trans. Learn. Technol., vol. 17, pp. 143-156, 2024.
- [18] S. Stamenkovic and N. Jovanovic, "Improving participation and learning of compiler theory using educational simulators," in 2021 25th International Conference on Information Technology (IT), 2021.
- [19] R. Friedman, "Tokenization in the theory of knowledge," Encyclopedia (Basel, 2021), vol. 3, no. 1, pp. 380-386, 2023.
- [20] G. P. Arya, N. Sohail, P. Ranjan, P. Kumari, and S. Khatoon, "Design and implementation of a customized compiler," Ijcsit.com. [Online]. Available: <https://www.ijcsit.com/docs/Volume%208/vol8issue3/ijcsit2017080308.pdf>. [Accessed: 01-Jan-2025].
- [21] G. Costagliola, M. De Rosa, and M. Minas, "Visual parsing and parser visualization," in 2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2019.
- [22] S. Tilscher and S. Wimmer, "LL(1) Parser Generator," Arch. Formal Proofs, vol. 2024, 2024.
- [23] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. Langendoen, Modern compiler design. New York, NY: Springer New York, 2012.
- [24] K. Qiu et al., "Flexible deterministic finite automata (dfa) tokenizer for ai-based malicious traffic detection," 20220279013:A1, 01-Sep-2022.
- [25] B. S. Abubakar, A. Ahmad, M. M. Aliyu, M. M. Ahmad, and H. U. Uba, "An Overview of Compiler Construction," Cloud-front.net, 2008. [Online]. Available: https://d1wqtxs1xzle7.cloudfront.net/68895919/IRJET_V8I398-libre.pdf. [Accessed: 25-Dec-2024].
- [26] Mr. Girish R. Naik*, Dr. V. A. Raikar, Dr. Poornima G. Naik, "Implementation of DFA parser for manufacturing query language tokens," International Journal of Engineering Sciences & Research Technology (IJESRT), vol. 4, no. 1, pp. 370-375, Jan. 2015.
- [27] P. Gupta, L. S. Y. Kumar, J. V. V. M. S. D. Santosh, D. Y. Kumar, C. Dinesh, and M. M. Venkata Chalapathi, "Design of efficient Programming Language with Lexer using '\$'-prefixed identifier," ICST Trans. Scalable Inf. Syst., 2023.
- [28] S. Ginsburg and S. Greibach, "Deterministic context free languages," in 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965), 1965, pp. 203-220.
- [29] S. M. Algaibeh, "Techniques for Enhancing Compiler Error Messages," Nmt.edu, Dec-2021. [Online]. Available: https://www.cs.nmt.edu/~jeffery/courses/585/Sanaa_Proposal2.pdf. [Accessed: 01-Jan-2025].
- [30] S. M. Algaibeh, "Techniques for Enhancing Compiler Error Messages," in Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 2, 2022.
- [31] S. Stamenkovic and N. Jovanovic, "Improving participation and learning of compiler theory using educational simulators," in 2021 25th International Conference on Information Technology (IT), 2021, pp. 1-4.
- [32] H. Hiranishi, Y. Imai, P. Podržaj, A. Ohno, and T. Hattori, "Application of web-based visual compiler to computer education," IEEE Trans. Electron. Inf. Syst., vol. 142, no. 3, pp. 389-394, 2022.
- [33] S. V. Kozlov and A. V. Svetlakov, "About LL(1)-grammars, algorithms on them and methods of their analysis in programming," Int. J. Open Inf. Technol., vol. 10, no. 3, pp. 30-38, 2022.
- [34] T. Æ. Mogensen, Introduction to compiler design. Cham, Switzerland: Springer Nature, 2024.
- [35] S. Sangal, S. Kataria, T. Tyagi, N. Gupta, Y. Kirtani, S. Agrawal, and P. Chakraborty, "PAVT: A Tool to Visualize and Teach Parsing Algorithms," Education and Information Technologies, vol. 23, pp. 2737-2764, 2018. [Online]. Available:

- https://www.researchgate.net/publication/325345755_PAVT_a_tool_to_visualize_and_teach_parsing_algorithms. [Accessed: Jan. 2, 2025].
- [36] M. Kennedy, "Parsing the Practice of Teaching," Researchgate.net, Nov-2015. [Online]. Available: https://www.researchgate.net/publication/283800050_Parsing_the_Practice_of_Teaching. [Accessed: 01-Jan-2025].
 - [37] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather, "ProGraML: A graph-based program representation for data flow analysis and compiler optimizations," ICML, vol. 139, pp. 2244–2253, 18–24 Jul 2021.
 - [38] S. H. Rodger, T. W. Finley, and C. R. Raposa, "Teaching Formal Languages with Visualizations and Auto-Graded Exercises," in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE 2021), pp. 1234–1240, 2021. [Online]. Available: <https://users.cs.duke.edu/~rodger/papers/openflapSigcse21.pdf>. [Accessed: Jan. 2, 2025].
 - [39] L. D'Antoni, M. Helfrich, J. Kretinsky, E. Ramneantu, and M. Weininger, "Automata Tutor v3," arXiv [cs.FL], 2020.
 - [40] Y. G. S. B. Q. H. Y. Le Traon, "CodeLens: An Interactive Tool for Visualizing Code Representations," Researchgate.net, Jul-2023. [Online]. Available: https://www.researchgate.net/publication/372684550_CodeLens_An_Interactive_Tool_for_Visualizing_Code_Representations. [Accessed: 01-Jan-2025].
 - [41] CHATTOPADHYAY and SANTANU, Compiler Design, second edition. Delhi, India: PHI Learning, 2022.
 - [42] L. D'Antoni, M. Helfrich, J. Kretinsky, E. Ramneantu, and M. Weininger, "Automata Tutor v3," in Computer Aided Verification, Cham: Springer International Publishing, 2020, pp. 3–14.
 - [43] Salisu Abubakar, Bashir and Ahmad, Abdulkadir and Aliyu, Muktar and Ahmad, Muhammad and Usman, Hafizu, "An Overview of Compiler Construction," International Journal of Research in Engineering and Technology, vol. 8, pp. 587–590, Mar. 2021.
 - [44] P. Akhilesh, K. Amal Krishna, S. K. Bharadwaj, D. Subham, and M. Belwal, "A visual approach to understand parsing algorithms through python and manim," in 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), 2024, pp. 1–7.