A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design

Vaikunta Pai T.[1], A. Jayanthila Devi[2] & P. S. Aithal[3]
[1]Associate Professor, College of Computer Science & Information Science, Srinivas University, Mangalore-575001, India.
ORCID: 0000-0001-6100-9023; Email: vaikunthpai@gmail.com
[2]Professor, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-6023-3899; Email: jayanthilamca@gmail.com
[3]Professor, College of Management & Commerce, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-4691-8736; Email: psaithal@gmail.com

Subject Area: Computer Science.
Type of the Paper: Review based Research.
Type of Review: Peer Reviewed as per |C|O|P|E| guidance.
Indexed In: OpenAIRE.
DOI: http://doi.org/10.5281/zenodo.4454632
Google Scholar Citation: IJAEML.

How to Cite this Paper:
Vaikunta Pai T., Jayanthila Devi A., & Aithal P. S., (2020). A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design. *International Journal of Applied Engineering and Management Letters (IJAEML)*, *4*(2), 285-301. DOI: http://doi.org/10.5281/zenodo.4454632.

A Systematic Literature Review of Lexical Analyzer Implementation Techniques in Compiler Design

Vaikunta Pai T.[1], A. Jayanthila Devi[2] & P. S. Aithal[3],

[1]Associate Professor, College of Computer Science & Information Science, Srinivas University, Mangalore-575001, India.
ORCID: 0000-0001-6100-9023; Email: vaikunthpai@gmail.com

[2]Professor, College of Computer Science & Information Science, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-6023-3899; Email: jayanthilamca@gmail.com

[3]Professor, College of Management & Commerce, Srinivas University, Mangalore – 575001, India.
ORCID: 0000-0002-4691-8736; Email: psaithal@gmail.com

ABSTRACT

The term "lexical" in lexical analysis process of the compilation is derived from the word "lexeme", which is the basic conceptual unit of the linguistic morphological study. In computer science, lexical analysis, also referred to as lexing, scanning or tokenization, is the process of transforming the string of characters in source program to a stream of tokens, where the token is a string with a designated and identified meaning. It is the first phase of a two-step compilation processing model known as the analysis stage of compilation process used by compiler to understand the input source program. The objective is to convert character streams into words and recognize its token type. The generated stream of tokens is then used by the parser to determine the syntax of the source program. A program in compilation phase that performs a lexical analysis process is termed as lexical analyzer, lexer, scanner or tokenizer. Lexical analyzer is used in various computer science applications, such as word processing, information retrieval systems, pattern recognition systems and language-processing systems. However, the scope of our review study is related to language processing. Various tools are used for automatic generation of tokens and are more suitable for sequential execution of the process. Recent advances in multi-core architecture systems have led to the need to re-engineer the compilation process to integrate the multi-core architecture. By parallelization in the recognition of tokens in multiple cores, multi cores can be used optimally, thus reducing compilation time. To attain parallelism in tokenizationon multi-core machines, the lexical analyzer phase of compilation needs to be restructured to accommodate the multi-core architecture and by exploiting the language constructs which can run parallel and the concept of processor affinity. This paper provides a systematic analysis of literature to discuss emerging approaches and issues related to lexical analyzer implementation and the adoption of improved methodologies. This has been achieved by reviewing 30 published articles on the implementation of lexical analyzers. The results of this review indicate various techniques, latest developments, and current approaches for implementing auto generated scanners and hand-crafted scanners. Based on the findings, we draw on the efficacy of lexical analyzer implementation techniques from the results discussed in the selected review studies and the paper provides future research challenges and needs to explore the previously under-researched areas for scanner implementation processes.

Keywords: Lexical Analysis, Scanner, Lexical Analyzer, Finite Automata, Regular Expression, Compiler, Tokens, Parallel Tokenization, Multi-core Machines

1. INTRODUCTION:

Phase one of the compiler construction is termed as scanning or lexical analysis. A lexical analyzer, also known as the lexer, is a pattern recognizer engine simulated by mathematical computational model known as Finite-State Machine (FSM) or Finite-State Automaton (FSA) that reads a string of individual characters as its input in the source program and clusters read characters into meaningful sequences called lexemes by matching with the token pattern and produces stream of tokens. A token is a sequence of character having a collective meaning and they are basic units of the programming language that is it describes the class or category of input string such as keywords, identifiers, literal strings, constants, operators and punctuation symbols. A pattern is a rule, which describes a token and to specify patterns in character strings regular expressions (RE) are used, which is an algebraic notation for describing sets of strings and used to construct recognizer for a language.

For every recognized lexeme, a token is represented and generated by a pair, <token-type and token-value>, is an attribute for token [1]. Here, the token-type refers to an abstract symbol or the category of token to be used in the syntax analysis process of compilation and the token-value is a pointer variable to the symbol table entry, in which the token information is stored. Fig.1 illustrates the working of tokenizer.

Apart from token recognition, lexical analyzer also performs following tasks**:**

(a) Removes white spaces (blanks, tabs and new lines) and comments added by the user from the program.

(b) Provides the stream of tokens generated as input to the next phase of the compilation – syntax analyzer or parser.

(c) Generates symbol table, which stores the information about identifiers, constants encountered in the source program, which is useful for successive phases of compilation.

(d) It keeps track of line numbers to facilitate the parser in reporting errors detected in the source program.

(e) It reports the error encountered while generating the tokens.

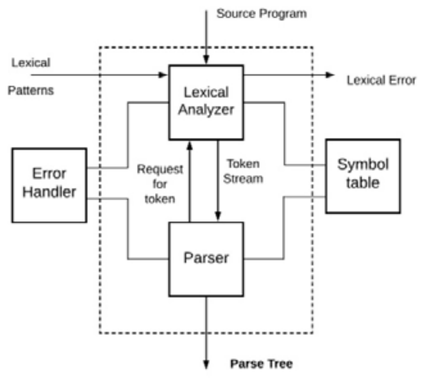

Fig. 1: Working of Tokenizer

The foremost attempt is made to generate lexical analyzers automatically for sequential programming with Lex**,** scanner generator tool that produces tables that drive the skeleton scanner. Hand-crafted lexical analyzer for languages is measured to be tedious process and time consuming factor. Therefore, it needs to be automatically generated. Lex tool [2] takes a set of formal description of tokens in the form of regular expression and produces a C program lex.yy.c which we call lexical analyzer or lexer that can identify these tokens. The process of identifying tokens is called as lexical analysis or lexing. The set of rules or descriptions given to lex is called a lex specification which contains two parts: (1) patterns and (2) corresponding actions. Lex tool automatically converts the lex specification into c statements into file containing a c subroutine called yylex().

Until the first decade of the twenty-first century, the method of compilation process was sequential in its execution. The recent advent of commercial multiprocessors has inspired researcher to take a closer look at parallel programming and various investigations were carried out for parallel tokenization process by splitting the source program on some criteria. Initially, Micknus and Shell [3] explored areas in which parallel processing is inherent to the compiling process and proposed concepts for further splitting the lexical analysis phase into two sub-phases: scanning and screening. The author in [3] recommends that text scanning can be done by multitask in parallel by breaking the source program into number of segments and for each segment pre-scanner accumulate the initial state and final states at each segment break.

2. OBJECTIVES:

The objective of this literature study is to summarize and synthesize the research findings currently available from the various lexical analyzer implementation techniques. To accomplish this objective, we carried out a comprehensive review of literature, which is "a way of reviewing and examining all available studies relevant to a specific research issue or specific subject or topic of interest" [4]. The primary objectives of this systematic analysis of literature are as follows:

- To identify and understand the different techniques of implementing lexical analyzer.
- To summarize the available research solutions for lexical analyzer implementation techniques.
- To synthesize the results from current lexical analyzer implementation techniques.
- To address the issues and needs of research studies in the context of lexical analyzer implementation techniques.

The remainder of this article is as follows: The process of systematic literature review is discussed in section 3, followed by a summary of the selected studies in section 4. Section 5 provides a generalized description of the different forms of lexical analyzer implementation techniques described in the selected studies. The current approaches to lexical analyzer implementation are outlined in section 6. Section 7 provides a collective discussion of all reviewed articles and outlines overall trends in the research findings. Conclusion of the study is provided in section 8 and section 9 sets out recommendations for potential work in the field of lexical analyzer implementation on problems and needs.

3. OVERVIEW OF SYSTEMATIC LITERATURE REVIEW METHODOLOGY:

Literature review is a critical mechanism that provides a solid basis for the advancement of knowledge. It facilitates the exploration of areas where research is required [5]. The purpose of this paper is to review the literature systematically in order to present the current research solutions for the implementation of lexical analyzer. We used the systematic literature review guidelines outlined by Kitchenham [4] to build a literature review framework. The literature review protocol to address the research objectives outlined in Section 2 is discussed in the following subsections. The literature review framework sets out the questions for research study, the methodology for the search for relevant studies, the selection of studies to be included in the literature review, the analysis of reviewed articles and the synthesis of the research findings will be discussed in the following subsections.

3.1. Research Queries for Study (RQS);

Research queries were drawn from the goals of the literature review and interested in responding to the following research issues:

RQS1: What are the various lexical analyzer implementation techniques?
RQS2: What are the current strategies for implementing lexical analyzer?
RQS3: What conclusions can we draw from the findings presented in the selected studies on the efficacy of lexical analyzer implementation techniques?
RQS4: What are the challenges and concerns of research in the domain of lexical analyzer implementation?

Before we explore the working of tokenization, we first want to understand the core idea in implementing lexical analyzer. Next, we want to understand the underlying ideas in current approaches for implementing lexical analyzer. The third question of research seeks to synthesize the outcomes from current research solution regarding the implementation of lexical analyzer and determine how well techniques work and its efficacy. The final research query concerns the exploration of open problems in the field of implementation of lexical analyzer.

3.2. Strategy for Search:

This section discusses the process of constructing search keywords, the search technique, the databases scanned, and documentation of search.

3.2.1. Search keywords and approach

Keywords used for our search were identified through prior experience with the field of study. The key database search string is "lexical analysis in compiler design" to focus on various approaches that implement lexical analyzers. The other search words are classified into two groups: lexical analyzer construction methods and implementation strategies. These terms are summarized in Table 1.

Table 1: Keywords describing lexical analyzer construction methods and implementation techniques

Construction Methods	Implementation Strategies
NFA to DFA	Table-driven
RE to DFA	Direct-coded
	Hand-coded

The search strings of the database combined the keyword "lexical analysis in compiler design" with one key term from the construction method column and one key term from the implementation strategies column from Table 1. For each database, there were six search strings using each combination of construction methods and implementation strategies.

3.2.2. Browsed databases

We compiled a list from the Google search engine of potential databases suggested for computer science research. We searched the following listed databases:

- ACM Digital Library
- IEEE Xplore
- ScienceDirect
- Google Scholar
- Web of Science

We excluded non-refereed papers as database search option allow for an advanced search and, in addition, we could limit papers by subject to Computer Science. The search was carried out between the January 2000 and December 2019.

3.3. Selection of Study:

The method and detailed documentation used to select studies for the systematic literature review of lexical analyzer implementation techniques are listed in this section.

3.3.1. Process of selection of study

The selection process of studies to be included in the systematic literature review is a done in three phases. (1) Selection of the initial study based on the title; (2) next selection process of studies based on reviewing the abstract concept; and (3) further selection process based on reading the full article. The table 2 shows the number of papers being evaluated at each stage of the selection process. The number of papers being reviewed at each point of the selection process is shown in Table 2.

Table 2: Number of research studies assessed at each point of the screening process

Stages	Selection Process	Total Papers
Phase 1	Based on the title	570
Phase 2	By reviewing the abstract concept	102
Phase 3	By reading the full article	50
Final phase	Studies selected	30

In phase 1, we began with 570 s papers from the database search excluding conference proceedings and picked 102 papers that passed to the next phase of screening of paper. At phase 3, 50 papers had relevant concepts and required full reading of the articles, and 30 papers were final selected for study.

The criteria for inclusion and exclusion focused on the identification of papers that report techniques of lexical analyzer implementation that have been used in practice. In particular, we are interested in algorithms for the implementation of lexical analyzer that improve the efficiency of token recognizers. The inclusion and omission parameters help to streamline our concept of lexical analyzer implementation techniques to meet our research objectives.

3.3.2. Documentation of the selection of studies

Before study selection, redundant papers found by different database keyword searches were eliminated. At each phase of selection process, research studies assessed at each point of the screening process were documented in separate worksheets in Excel spreadsheet application. After the final phase of screening process, the selected studies moved to my library in Mendeley.

3.4. Analysis of studies:

After all phases of the selection of the systematic literature review study were completed, the next stage assessed the quality of the studies chosen for analysis and extracted the relevant data from the selected studies for the literature review. Eighteen of the 30 studies selected did not report shortcomings in their assessment methodology. Nine of the relevant studies did not provide any performance benchmarks for the comparative evaluation or did not report any constraints or limitations on their implementation techniques. The following data were extracted from selected studies:

- Reference type (journal article/conference article)
- Objective of research
- Various Lexical analyzer implementation techniques
- Limitations of study
- Methodology for evaluation (experiment based, case study, etc.)
- Metrics of assessment
- Results of evaluation
- Implementation tools used
- Limitations of evaluation

3.5. Synthesis of data extracted from selected studies:

For each research query set out in Section 3.1, the related data obtained from each of the selected studies were summarized. Section 5 offers a high-level overview of various implementation techniques, which answer research query for study 1 (RQS1). Section 6 provides an overview of the current approaches in lexical analyzer implementation techniques, methodologies of research, and metrics of evaluation utilized in the selected studies, and presents the summary of the research results for RQS2. The efficacy of results outlined in the selected studies is summarized in section 7 and answers RQS3. Section 8 concludes and future work is discussed in Section 9 and addresses RQS4.

4. OVERVIEW OF RELATED WORK:

This section gives an extensive review about the lexical analyzer and methods that are used to improve efficiency by reducing its complexity.
Sabine Glesner et al, (2005) modeled architecture for validating the outcomes of front-end based computations more specifically in lexical analysis. The author shows the specific task of scanning tokens in HOL/Isabella [6]. Amit Barve et al., (2014) shows the limitation of past work is preprocessing time required for detection of pivot locations in programs [7]. A number of programs were written manually which require variable times as typing speeds of programmers vary. The advantage of using smart editor is that as soon as typing finishes, the preprocessing also finishes, thereby saving substantial preprocessing time. Xiaoyan Lai., (2014) initiates a novel implementation process for lexical analysis [8], interpretive execution and syntactic analysis. The experimental analysis provides integrity and reliability of compilation system. Amit Barve et al., (2012) presented a new approach of implementing lexical analyzer to run in parallel which is based on an open source automatic lexer generator Flex and

exploiting the concept of processor affinity. It is measured to be a simple and faster process by partitioning code written in C/C++ programming language based on for-loop looping structures [9]. Work reasonably illustrates the benefit of multi-core architecture machines in accelerating the process of lexical analysis tasks. Thomas Reps et al., (1998) described the compilation domain, where tokenization process can always be carried out in time linear in the input size [10], while most of the standard tokenization algorithm explains that, in the worst case, the scanner can exhibit quadratic behavior for some sets of token definitions. Amit Barve et al., (2013) examined the lack of availability of benchmark programs for different investigators to validate the analysis and performance of algorithm [11]. His proposed tool is extremely essential for academic and research purposed. The present version of tool generates only decision making and loop construction with appropriate assignment and printing statements. Ami et al., (2004) describes Unified Parallel C (UPC) and Open Multi-Processing using C (OpenMP-C) as two programming languages currently being developed for parallel programming built on ANSI C as a source-language. OpenMP offers a simple programming framework with an annotation of serial code with compiler directives and UPC is a parallel processing extension to C programming language based on the single program, multiple data (SPMD) programming model technique to achieve parallelism and due to the separation between logical parallelism and the physical execution environment, it provides high-level abstraction and portability of the machine [12]. Y. Omori et al., (1997) suggested a parallel compiler design approach using the virtual class principle to describe parallel tasks using Object Modeling Techniques [13]. Ivica M. Marković, (2018) explained JFlex tool, which is a scanner generator for Java programming language. It visually represents Deterministic Finite Automaton (DFA) as directed graph generated for given regular expression and shows how input is processed [14]. Amit Barve, (2016) provided an enhanced version for parallel lexical analysis process. It is more obvious that the process performs well compared to the previous approaches like round robin for parallel lexical analysis [15]. The maximal speed attained is 4.14. The speed is considered to be higher; moreover if number of CPU increases. As an outcome, this model can further enhances the compilation time. Amit Barve et al., (2015) explained an enhanced version for parallel lexical analysis algorithm. The author claims that the outcome of memory block based algorithm outperforms the result of his previous work round robin CPU scheduling technique based parallel processing of lexical analysis and the highest speed achieved is 6.84 [16]. The speed will be increased when number of CPU rises and also improves the overall compilation time. Daniele Paolo Scarpazza et al., (2007) investigated the importance of the efficiency of the cell processor system when it is used for the implementation of Deterministic Finite Automata based string matching process algorithms [17]. The results of their experiment indicate that the Cell is the perfect candidate for managing security requirements. Two of the eight processing elements available on one cell processor have ample computing capacity to process a network connection with a bit rate of more than 10 Gbps. Using the Aho-Corasick string searching algorithm, Daniele Paolo Scarpazza et al., (2008) developed optimized string matching solutions for the Cell processor and the result showed a throughput of 40 Gbps per processor when the dictionaries are small enough to fit into the local memory of the processing cores and the throughput for larger dictionaries is between 1.6 and 2.2 Gbps per processor [18]. Daniele Paolo Scarpazza et al., (2009) optimized Flex's original kernel to run on each of the eight IBM cell processor Synergetic Processing Elements [19] and proposed an algorithm to match regular expressions against a minimal set of rules that meet the needs of tokenizers for search engines and suit multi-core architectures. This approach will substitute the flex-generated kernels while using the flex front-end to process the rule sets and build the resulting finite automaton. Vaishali Bhosale et al., (2015) identified the likelihood of fuzziness in keywords due to the addition, deletion, substitution, typing and letter sequence errors and their implementation. The implementation approach is to use a fuzzy automaton to enable flexibility or fuzziness in the process of token recognition referred to as a fuzzy lexical analysis [20]. Swagat Kumar Jena et al., (2018) describes their approach to develop parallel lexical analyzer by dividing the input program into fragments equal to the number of available cores in the system [21]. Their work shows the benefits of a multi-core machine by parallelizing the process of tokenization and enhancing the scanner's performance by increasing the number of cores. Wuu Yang et al., (2002) identified the issue of the longest-match rule's applicability and proposed a model [22]. The approach comprises of two processes: the first is to calculate the regular set of token sequences generated by the non-deterministic finite automaton, while the automaton processes elements of an input regular set and the other is to check if there is a non-trivial intersection with a set of equations between a regular set and a context-free language. Russell et al., (1992) developed extensions of the parallel procedure language and a run-time framework to support parallel procedure models in C programming language [23]. In order to reduce the need for expensive process control blocks to be implemented, a novel method for nesting parallel process contexts in multiple stack frames is used in the run-time framework and the performance data for two parallel programs utilizing their proposed system is provided. Amit Barve et al., (2012) explored and compared the results of three approaches for the parallelization of lexical analysis tasks of the compilation process. It is observed that the execution speed achieved by these methods is relatively high and the highest speed attained is 8.64 [24] and is anticipated to boost further as CPU cores and control statements in source code increases. The round robin scheduling algorithm was used to schedule the CPU. Venkatesan Packirisamy et al., (2005) explained how OpenMP could be used in Multi-core Processors. This has been studied in two parts - extracting fine grained parallelism and extracting speculative parallelism. The study also recommends some hardware strategy which may boost the efficiency of speculation on the thread level [25]. Umarani Srikanth, (2010) modeled framework for concurrent execution of lexical analyser tasks for cell processor by dividing the source program into fixed set of blocks using dynamic block splitting algorithm for performing lexical analysis in parallel [26]. Tokenization is performed by using the Aho-Corasick algorithm to search for a string in multicore processors at high speed against large dictionaries. Amit Barve et al., (2014) designed an algorithm that can be used by multi-core machines for performing lexical analysis of multiple files concurrently [27]. Experiments which can save a significant amount of time in the lexical analysis process by spreading files through a number of CPUs. Dancheng Li et al., (2012) explains the structural features and functionality of the scanner and defines the configuration of three essential modules and describes the objective and functionality of each module in a lexical analyzer in detail [28]. They found that there is scope for improving lexical analyzer efficiency so that each lexical analyzer module is improved by integrating the grammar analysis process, semantic analysis and some other sections, making it easier to upgrade and improve the Prolog compiler. Ami Marowka, (2008) discussed about the performance and the programmability of Threading Building Blocks, which is a new paradigm built on standard template library of C++ for parallel programming [29]. It enables programmers to focus on the algorithmic problem without having to deal with concurrency issues such as load balancing, scheduling, deadlocks, synchronization and race conditions. Yujia Zhai et al., (2017) proposed a framework for using the natural language processing in compilation process using maximum likelihood word segmentation algorithm during the lexical and syntax analysis process to provide a more efficient interface between humans and computers [30]. Aleksandr A. Maliavko, (2018) discusses the algorithms used to verify the correctness of the syntax and the lexic using the scanner program written in E1 language [31]. It describes the components of the formal definition of the lexic and syntax structure of the El-language that are used to construct the scanner and parser phases of the compiler in C++ programming language using the Webtranslab client-server package. It also describes the scanner algorithm used in the compilation that performs macro-processing, file inclusion, deletion of insignificant characters, correction of language words in internal token representation. Oreste Villa et al., (2009) implemented a string search process on the multithreaded Cray XMT shared memory machine using Aho-Corasick algorithm [32]. They used the functionality and some algorithmic techniques of the XMT multi-threaded architecture and succeeded in achieving scalable high performance, which is independent of the analyzed input stream and the corresponding set of patterns. Ryoma Sinya et al., (2013) have developed simultaneous finite automaton, which is a new automaton model for effective parallel computation of the finite state machine by extending the automaton to include the simulation of transitions [33]. Xiang Wang et al., (2019) modeled multi pattern regular expression matcher framework for commodity server machines for achieving high performance by using two techniques: graph decomposition to transform matching of regular expression into a sequence of string and finite machine matching and accelerate the matching of string and finite automata using SIMD operations, which brings major improvement in throughput [34]. Michela Becchi et al., (2013) implemented a DFA compression technique that results in comparable compression levels with lower observable memory bandwidth limits [35]. Amit Barve et al., (2017) introduced a parallelization tool for object-oriented programs on multi-core architecture that automatically converts sequential code into its compatible parallel code and it is found from experimental studies that the performance and use of multi-core architecture is substantially improved [36].

5. HIGH LEVEL IDEA IN IMPLEMENTING LEXICAL ANALYZER:

The following subsection describes the core high level idea in implementing lexical analyzer pertains to

- RQS1: What are the different techniques for implementing lexical analyzer?

5.1 Construction methods:
Lexical analyzers can be constructed in two ways:

- First method involves writing a program to do the lexical analysis i.e., creating a lexical analyzer by hand.
- Another method involves automatic generation of the lexical analysis program from a formal description of the tokens of the language using scanner generator tools.

5.2 Construction approaches:
There are two approaches to build Lexical analyzers:

- Constructing a DFA from NFA.
- Constructing a DFA from an augmented pattern without creating an intermediate NFA and improving Efficiency of Token Recognizers by minimizing the DFA states so that it affects the space and time requirements of a DFA-based pattern matcher simulator.

5.3 Construction steps:
There are two ways where we can build lexical analyzer.
5.3.1 NFA to DFA:
The following steps are involved in the construction of lexical analyzer using NFA to DFA**:**

- Specify regular expressions for each syntactic category
- Construct NFA for each regular expression using Thomson's construction rules
- Convert NFA to DFA using Subset construction that simulates the behaviour of the RE
- Generate executable code to implement the constructed DFA

5.3.2 Regular Expression to DFA:
The following steps are involved in the construction of lexical analyzer using the method proposed by Aho, Sethi, and Ullman [1] i.e., Regular Expression to DFA without constructing NFA:

- Specify regular expressions for each syntactic category
- Regular expression is augmented with a delimiter '#' to indicate accepting state
- Attach a unique integer to each of the symbols which indicates its position.
- Build a syntax tree for augmented pattern where the symbols are placed in the leaf nodes and the operators are placed in the interior nodes. Each and every leaf node in the syntax tree is given a position value, which is a unique integer.
- Compute the functions nullable, firstpos, lastpos and followpos for every node in syntax tree by making depth first traversals over syntax tree.
- Construct DFA using AhoSethiUllmann's Algorithm
- Minimize DFA states using Hopcroft's algorithm by identifying similar states in the constructed DFA in previous step
- Generate executable code to implement the constructed minimized DFA

5.4 Implementation strategies
There are three implementation strategies to convert DFA to executable code [37]:

- table-driven lexical analyzer
- direct-coded lexical analyzer
- hand-coded lexical analyzer

All of these lexical analyzers work in the same way by modelling the DFA. The next input character is repeatedly read and the DFA transition caused by that character is simulated. This process stops when the word is recognized by the DFA. These three deployment methods vary in the specifics of their runtime costs. They all, however, have the same asymptotic complexity, constant cost per character and roll back costs. The variations in the performance of well-constructed lexical analyzer change the constant cost per character, but not the asymptotic nature of scanning process.

5.4.1 Table-Driven Lexical Analyzers

The table-driven method uses a lexical analyzer code for regulate and set of transition tables that encode information about the syntactic categories, the transition function and the type of token for the purpose of simulating the DFA for each input. As shown in Figure 2, the lexical analyzer generator takes set of lexical patterns in terms of regular expressions and generates transition tables that drive the lexical analyzer code with input characters.

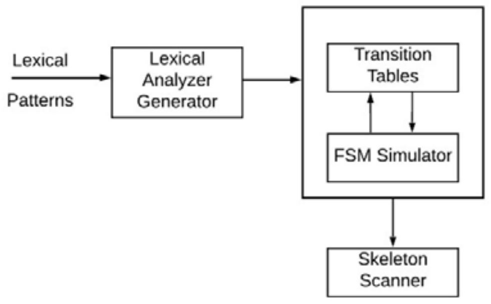

Fig. 2: Generating a Table-Driven Lexical Analyzer

5.4.2 Direct-Coded Lexical Analyzers

To boost the efficiency of a table-driven approach, we need to reduce the cost of its basic operations to scan the input character and measure the next DFA transition. Direct-Coded lexical analyzers reduce this overhead by replacing the explicit representation of the DFA's state and transition graph with an implicit one to simplify the two-step, table-lookup calculation and to eliminate the memory references. The functionality of this approach of building scanner is same as table-driven, but it has a lower overhead per input character.

5.4.3 Hand-Coded Lexical Analyzers

The overhead of the DFA simulation was reduced by the direct-coded scanner; the overhead of interfacing between the lexical analyzer and the rest of the system can be reduced by hand-coded approach to further improve the efficiency of lexical analyzer. The mechanism to read and manipulate characters on input and the operations required to generate a copy of the original lexeme on output can be enhanced by diligent implementation of scanner. Most of the popular open-source compilers depend on scanners that are hand-coded. From the implementation point of view, the characters of the source program are read from a file, but I/O reading is always slow. Hence, input buffering is used for scanning the input program.

5.5 Buffering the Input Stream

When computers do not have a large memory, the tokens to be formed from the source language are to be read from the storage devices. Owing to the limitations in the main memory capacity, the source program is usually stored in secondary storage devices. However, it is a highly time-consuming process when the scanner program reads the source language from the secondary storage devices. Therefore, a fixed block of input data is stored into a buffer and then scanned by the lexer program. There are two ways in which buffering can be done: one buffer scheme and two buffer scheme which is called double buffering. In one buffer technique, single buffer of fixed size is used to hold the input data. The problem with this buffering technique is that if a lexeme size crosses the boundary of the buffer, the buffer has to be refilled in order to scan the rest of the lexeme and the first part of the lexeme will be overwritten in the process. In double buffering scheme two buffers are scanned alternately. Each buffer is N character long, where the variable N represents the number of characters on a single disk block. Using system command, input characters of length N will be read into each buffer. When one of them reaches the end of the current buffer, the other buffer is filled. This scheme fixes the issue indicated in a single buffer scheme if the lexeme length is longer than the buffer length.

The lexical analyzer uses two pointers - a lexeme beginning pointer (lb) and a forward pointer (fp) to keep record of the portion of the scanned input string. The current lexeme is a string of characters between these two pointers. Initially, both pointers are set to the beginning of the lexeme being scanned. Then the fp pointer begins scanning forward until a match is found for a pattern. When the end of the lexeme is found, the pointer fp is set to the character at the right end of the lexeme and then token and attribute corresponding to that lexeme is returned. Both pointers lb and fp are then set to the beginning of the next token, which is the character immediately past the lexeme. If less than buffer size characters remain in the input stream while the buffer is being refilled, the EoF (end of file) marker will be read in the buffer after the last input character being read.
In the code for advancing fp, whenever we advance fp we must check whether we have reached the end of one buffer or not as in that case we have to refill the other buffer before advancing fp. So, we have to make three tests, one for checking whether the first buffer is full or not, second for checking whether the second buffer is full or not and third to check the end of input. We can reduce these three tests to one if we add a special character called sentinel at the end of each buffer. This sentinel is not part of source program. Let us choose EoF as the sentinel so that the buffers look as in fig.3.

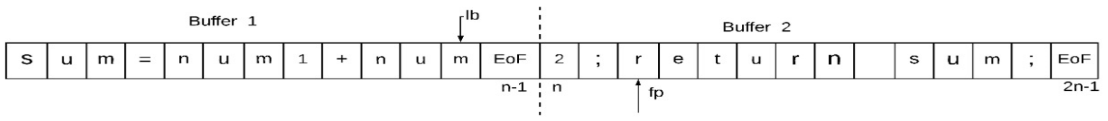

Fig. 3: Double buffering with sentinel at the end of each buffer
Using sentinels, we can write the code to see whether pointer fp points to the sentinel EoF by conducting only one test. If EoF is reached, which could be the end of first or second buffer or the end of a file, the code will conduct more tests. If N is large, the average number of test cases per character read is approximately 1, which improves the buffering efficiency the source program.

6. CURRENT APPROACHES FOR IMPLEMENTING LEXICAL ANALYZER:

The following section presents the current approaches and research methodology used for the implementation of the lexical analyzer referred to in the selected studies. The review of the proposed scanner implementation techniques established eight scanner generation approaches in the selected studies that lead to RQS2 and are presented in Table 3.

- RQS2: What are the current approaches for implementing lexical analyzer?

With the emergence and use of multi-core architectures, it reflects the latest evolution of modern computing with its efficiency and cost benefits. As in modern age of supercomputers, the most of system processors belong to the family of multi-core processors. In multi-core architecture, the latest trend is scalability, meaning that the number of cores per chip is growing. The team headed by Dr. Wim Vanderbauwhede at Glasgow University succeeded in installing 1,000 cores on a single chip (Cooter, 2016). Present day's high-end multi-core personal computer systems are used to perform complex tasks simultaneously. As the number of cores in system continues to grow, the question always emerges from software designers as to whether software applications running on these platforms can make maximum use of the inherent parallelism offered by these platforms. To meet these challenges, software programmes should be optimised for simultaneous execution, and these parallelized platforms should also be supported by system software and it should not result in bottlenecks being scalable. In order to achieve a concurrent compilation process, the various stages of the compilation process need to be remodelled to fit the multi-core design system. The current trend in the generation of scanners is to use the processor affinity mechanism on multi-core systems to adapt parallelism in the lexical analysis tasks to increase performance in terms of overall execution time of lexing process of compilation compared to traditional sequential lexing on single processor system.

Extensive research has been carried out to enhance the efficiency of the conventional scanner design in order to take full benefits of the multi-core framework. For the very first time, the researchers Mickunas and Shell (Mickunas & Shell, 1987) outlined various stages in the compilation where parallelization is feasible and proposed a novel theoretical approach for dividing the lexical analysis process into scanning and screening, suggesting that input can be scanned simultaneously.

Table 3: Summary of current approaches being used by scanner generation in the reviewed articles

Study	Objective	Research methodology
Daniele Paolo Scarpazza et al. [19]	To optimize Flex's original kernel to run on each of the eight IBM cell processor Synergetic Processing Elements.	Proposed an algorithm for parallel regexp-based tokenization that exploits the large amount of thread-level and data-level parallelism available in multi-cores framework. It is based on the Deterministic Finite Automation (DFA) model designed for predication-like branch removal and SIMDization.
Umarani Srikanth [26]	To model framework for concurrent execution of lexical analyser tasks for cell processor.	Proposed method based on dividing the source program into fixed set of blocks using dynamic block splitting algorithm for performing lexical analysis in parallel. Tokenization is performed by using the Aho-Corasick algorithm to search for a string in multicore IBM Cell processors at high speed against large dictionaries.
Swagat Kumar Jena et al. [21]	To develop parallel lexical analyzer for C language.	The method is to break the source file into N blocks where each block contains M lines with the exception of the last block and store each block in memory in terms of files and build N lexical program threads and perform lexical analysis in parallel for each file (say fi); where: N = Total number of cores available M $=\frac{\text{Total number of lines in source program}}{\text{Total number of cores}}$
Amit Barve et al. [9]	To implement lexical analyzer to run in parallel on multi-core machines.	The method based on an open source automatic lexer generator Flex and exploiting the concept of processor affinity and partitioning code written in C/C++ programming language based on for-loop looping structures
Amit Barve et al. [24]	To implement lexical analyzer to run in parallel on multi-core machines using divide and conquer algorithm design paradigm.	Approach is based on dividing the source code into a fixed number of blocks equal to the number of available CPUs by specifying the pivot positions. White space character, various constructs and lines-based pivot elements were considered.
Amit Barve et al. [15]	To implement fast parallel lexer for multi-core machines.	Modified version of the algorithm given by Amit Barve et al. [9, 24]. An algorithm is proposed to record block markers of source code into a text file which will be read later and based on read markers, processes are forked and

		using processor affinity processes are allocated to different CPUs and algorithm efficiency is improved by assigning processes to a free processor as and when a process is generated. This method avoids waiting time for a process to be allocated to a processor.
Amit Barve et al. [16]	To implement lexical analyzer to run in parallel on multi-core machines using OpenMP	Developed new method to improve the algorithm designed by authors earlier for parallel lexing by generating the blocks in memory and using OpenMP each block is processed in parallel on multi-core machines to minimize waiting period for a process in I/O queue.
Amit Barve et al. [27]	To implement parallel tokenizer for multi-core machines on multiple files.	Modified version of the algorithm proposed by Amit Barve et al. [9, 24] to extend parallel lexical analysis on multiple files.

7. EFFICACY OF APPROACHES PRESENTED IN THE SELECTED STUDIES:

The approaches discussed in the studies reviewed and outlined in Sections 6 support the premise that techniques for scanner implementation that supplement parallel processing in the lexical analysis tasks using the processor affinity principle on multi-core systems can improve the performance in terms of overall execution time of lexing process of compilation compared to traditional sequential lexing on single processor system. Analyzing the combined results answer RQS3.

- RQS3: What conclusions can be drawn from the findings of the selected studies on the efficacy of lexical analyzer implementation techniques?

Daniele Paolo Scarpazza et al. [19] exhibits the implementation of his proposed Cell processor algorithm delivering potential output between 8 and 14 Gbps per chip under realistic conditions that comply with Lucene's tokenizing rules (a common open source search engine library) operating on Wikipedia pages. Its proposed solution includes a detailed list of optimization strategies that promise to be helpful to other multi-core architectures and easily portable to other SIMD-enabled processors, and there is a general trend in architecture design towards more and broader SIMD instructions. Umarani Srikanth [26] has built a parallel lexical analyzer that runs on the simulator of the IBM Cell Processor and shows execution time results by varying the size of the code and the number of processing elements. With larger file sizes, efficiency gains are more pronounced as the overhead in the switching process becomes more negligible as the load on each synergetic processing element rises with an increase in the size of the file. Swagat Kumar Jena et al. [21] implemented the algorithm for Parallel Lexical Analyzer using OpenMP multi-threading library for C programming language. Using multi-core compactable tools that are available to test the program for parallelization, the efficiency of algorithm is evaluated. Multi2sim and Intel-VTune amplifier are tools used for performance evaluation and tested on an HP machine with 4 GB RAM and 2.03 GHz processor speed with 4 cores installed with Ubuntu 14.04 LTS. By using a set affinity function and some OpenMP built-in features, all the threads of the lex program were bind to all CPUs. Using the ps-eLF command, the running of all the lexical analyzer program threads was verified on all cores. Source files of different sizes are used for algorithm testing as an input to the parallel lexical analyzer and tested under the Multi2sim simulator tool. The result shows the improvement in speed performance with the increased number of CPUs and the sudden drop in speed due to the lack of adequate process for a CPU when the current assigned task is completed due to the Round Robin scheduling used in simulation. As the number of cores rises and executed in parallel, the performance improvement in the lexical analysis phase is clearly observed. Amit Barve et al. [9] implemented lexical analyzer to run in parallel on multi-core machines using processor affinity and flex open source tool. The C / C++ source code is split for parallel execution on the basis of for loops and other constructs are not considered. The experimental result shows the speed-up efficiency gain with the increased number of CPUs and the speed-up dip is due to a lack of work for the CPU when it has completed the assigned task. This is due to the Round Robin scheduling technique was used in simulation. Amit Barve et al. [24] implemented lexical analyzer to run in parallel on multi-core machines using divide and conquer algorithm design paradigm. In implementation they have proposed an approach to divide the source program into a fixed number of blocks equal to the number of available CPUs by defining pivot locations based on whitespace character, programming constructs and lines. Up to 2 Constructs, algorithms based on line and whitespace character perform better than algorithms based on constructs but 4 constructs onwards, the algorithms based on the construct perform better than the algorithm based on white space character and perform poorer than the algorithm based on line. It is noted that from 512 constructs onwards, the construct-based technique outperforms line-based algorithms and the other techniques. The result shows that the maximum speed reached was 8.64 and is expected to increase more as the number of CPUs and constructs increases and thus enhances the overall time of compilation. The scheduling of processes to different CPUs was achieved using round robin in the implementation process. Adapting an effective scheduling algorithm will certainly increase the performance further. Amit Barve et al. [15] suggested an updated version of the technique given by Amit Barve et al. [9, 24]. An experimental outcome shows that proposed approach outperforms the round robin based parallel lexical analysis which is previously explored. The highest speed reached was 4.14 and is expected to grow further if the number of CPUs increases and the overall compilation time is further improved. Amit Barve et al. [16] proposed a new approach using OpenMP for parallel lexical analysis and showed an improvement in the lexical analysis process by automatically generating the C code with 10,000 possible parallel constructs and attained maximum speed of 6.84 for 7 CPUs. It is evident from the experimental outcome that the memory block algorithm implemented in the method outperforms the previously explored round robin approach of parallel lexical analysis. Amit Barve et al. [27] revised the algorithm proposed by Amit Barve et al. [9, 24] to extend parallel lexical analysis to multiple files. It is apparent from studies that a significant amount of time in the lexical analysis process can be saved by distributing input files to the available CPUs. With the rise in the number of processors, there is no doubt that the total time in the compilation will decrease compared to the serial approach. Speedup may be further examined if individual files may also be scheduled for lexical analysis on multiple processors using one of the approaches previously explored in Amit Barve et al. [9, 24].

8. CONCLUSION:

The focus of this study was to carry out a systematic analysis of current research work on the implementation strategies of lexical analyzers. It is known from the review that various software tools for lexical analyzers have been developed in the past that are ideally suited for sequential execution. With the emergence of multi-core architecture systems, the various phases of the compilation process need to be revamped to fit the multi-core architecture technologies in order to attain a parallelism in compilation tasks and thereby minimize the time of compilation. An extensive analysis provides a deeper insight towards the lexical analyzer. Among the results recorded in the reviewed articles, it is observed, some of the high-level trends in scanner generation are the adaptation of parallel processing in lexical analysis tasks by using multi-core processor affinity principle to increase the efficiency of the compiler's runtime compared to the sequential execution of lexical analysis tasks on a single processor system. This trend of implementation shows that parallel lexical analyzer tends to perform lexing tasks better than a conventional sequential scanner. The selected literature review studies have shown the efficacy of incorporating parallelism in lexical analysis to minimize the considerable amount of time needed for text scanning and thus increase performance compared to conventional sequential tokenization process on a single processor. The paper thus proposed a future research directions based on the findings discussed.

9. FUTURE WORK:

From the detailed analysis of lexical analyzer implementation studies, we can focus on RQS4.

- RQS4: What are the challenging issues and needs in the implementation of lexical analyzer?

The development of computing power is moving rapidly towards massive multi-core platform due to its power and performance benefits. In order to exploit the maximum capabilities of multi - core technologies, system software such as compilers should be engineered for parallel processing. The sophisticated framework has to be designed for parallel execution of lexical analysis tasks by adopting algorithm design paradigm best suited for parallel computing and CPU scheduling strategies for efficient utilization of processors in multi-core system. Thus, the experimental results of the proposed tokenizer show substantial progress in performance in the lexical analysis phase engineered with parallel processing compared to the conventional sequential version of the lexical analysis in terms of compilation time and also affect the time and space requirements of the DFA based parallel pattern matcher simulator.

REFERENCES:

[1] Aho, A. V., Lam, M. S., & Sethi, R. (2009). Compilers Principles, Techniques and Tools, 2nd ed, PEARSON Education.

[2] Lesk, M. E., & Schmidt, E. (1975). Lex: A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hills, New Jersey.

[3] Mickunas, M. D., & Schell, R. M. (1978, December). Parallel compilation in a multiprocessor environment. In Proceedings of the 1978 annual conference (pp. 241-246).

[4] Kitchenham, B. (2004). Procedures for performing systematic reviews. Keele, UK, Keele University, *33*(1), 1-26.

[5] Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. MIS quarterly, 26(2), 8-23.

[6] Glesner, S., Forster, S., & Jager, M. (2005). A program result checker for the lexical analysis of the gnu c compiler. Electronic Notes in Theoretical Computer Science, *132*(1), 19-35.

[7] Barve, A., & Joshi, B. K. (2014). A Smart source code editor for C. International Journal of Computer Science, Engineering and Information Technology, *4*(3), 23-28.

[8] Lai, X. (2014, June). A design of general compiler for NC code in embedded NC system. In 2014 9th IEEE Conference on Industrial Electronics and Applications (pp. 1515-1519). IEEE.

[9] Barve, A., & Joshi, B. K. (2012, September). A parallel lexical analyzer for multi-core machines. In 2012 CSI Sixth International Conference on Software Engineering (CONSEG) (pp. 1-3). IEEE.

[10] Reps, T. (1998). "Maximal-munch" tokenization in linear time. ACM Transactions on Programming Languages and Systems (TOPLAS), *20*(2), 259-273.

[11] Barve, A., & Joshi, B. K. (2013). Automatic C Code Generation for Parallel Compilation. International Journal on Advanced Computer Theory and Engineering (IJACTE), 2(4), 26-28.

[12] Marowka, A. (2004, July). Analytic comparison of two advanced c language-based parallel programming models. In Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (pp. 284-291). IEEE.

[13] Omori, Y., Joe, K., & Fukuda, A. (1997, August). A parallelizing compiler by object-oriented design. In Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97) (pp. 232-239). IEEE.

[14] Marković, I. M. (2018, November). An Application for Visual Representation of Deterministic Finite Automaton Generated by JFlex. In 2018 26th Telecommunications Forum (TELFOR) (pp. 420-425). IEEE.

[15] Barve, A., & Joshi, B. K. (2016). Fast parallel lexical analysis on multi-core machines. International Journal of High-Performance Computing and Networking, *9*(3), 250-257.

[16] Barve, A., & Joshi, B. K. (2015). Improved Parallel Lexical Analysis using OpenMP on Multi-

core Machines. Procedia Computer Science, *49*(1), 211-219.

[17] Scarpazza, D. P., Villa, O., & Petrini, F. (2007, March). Peak-performance DFA-based string matching on the Cell processor. In 2007 IEEE International Parallel and Distributed Processing Symposium (pp. 1-8). IEEE

[18] Scarpazza, D. P., Villa, O., & Petrini, F. (2008, May). Exact multi-pattern string matching on the Cell/BE processor. In Proceedings of the 5th conference on computing frontiers (pp. 33-42).

[19] Scarpazza, D. P., & Russell, G. F. (2009, June). High-performance regular expression scanning on the Cell/BE processor. In Proceedings of the 23rd international conference on Supercomputing (pp. 14-25).

[20] Bhosale, V., & Chaudhari, S. R. (2015). Fuzzy Lexical Analyser: Design and Implementation. International Journal of Computer Applications, *123*(11), 01-07.

[21] Jena, S. K., Das, S., & Sahoo, S. P. (2018). Design and Development of a Parallel Lexical Analyzer for C Language. International Journal of Knowledge-Based Organizations (IJKBO), *8*(1), 68-82.

[22] Yang, W., Tsay, C. W., & Chan, J. T. (2002). On the applicability of the longest-match rule in lexical analysis. Computer Languages, Systems & Structures, *28*(3), 273-288.

[23] Clapp, R. M., & Mudge, T. N. (1992). Parallel language constructs for efficient parallel processing. University of Michigan, Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science. IEEE, 230-241.

[24] Barve, A., & Joshi, B. K. (2012, December). Parallel lexical analysis on multi-core machines using divide and conquer. In 2012 Nirma University International Conference on Engineering (NUiCONE) (pp. 1-5). IEEE.

[25] Packirisamy, V., & Barathvajasankar, H. (2005). Openmp in multicore architectures. University of Minnesota, Tech. Rep. 1-15.

[26] Srikanth, G. U. (2010, June). Parallel lexical analyzer on the cell processor. In 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (pp. 28-29). IEEE.

[27] Barve, A., & Joshi, B. K. (2014). Parallel lexical analysis of multiple files on multi-core machines. International Journal of Computer Applications, *96*(16). 22-24.

[28] Li, D. C., Cai, X. C., Han, C. Y., & Liu, Y. X. (2012). The Research and Analysis of Lexical Analyzer in Prolog Compiler. In Applied Mechanics and Materials (Vol. 229, pp. 1733-1737). Trans Tech Publications Ltd.

[29] Marowka, A. (2008, December). Towards high-level parallel programming models for multicore systems. In 2008 Advanced Software Engineering and Its Applications (pp. 226-229). IEEE.

[30] Zhai, Y., Liu, L., Song, W., Du, C., & Zhao, X. (2017). The applications of natural processing language in compiler principle systems. IEEE. 245-248.

[31] Maliavko, A. A. (2018, October). The Lexical and Syntactic Analyzers of the Translator for the EI Language. In 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE) (pp. 360-364). IEEE.

[32] Villa, O., Chavarria-Miranda, D., & Maschhoff, K. (2009, May). Input-independent, scalable and fast string matching on the Cray XMT. In 2009 IEEE International Symposium on Parallel & Distributed Processing (pp. 1-12). IEEE.

[33] Sinya, R., Matsuzaki, K., & Sassa, M. (2013, October). Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In 2013 42nd International Conference on Parallel Processing (pp. 220-229). IEEE.

[34] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., & Zhu, H. (2019). Hyperscan: a fast multi-pattern regex matcher for modern cpus. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19) (pp. 631-648).

[35] Becchi, M., & Crowley, P. (2013). A-dfa: A time-and space-efficient dfa compression algorithm for fast regular expression evaluation. ACM Transactions on Architecture and Code Optimization (TACO), *10*(1), 1-26.

[36] Barve, A., Khomane, S., Kulkarni, B., Ghadage, S., & Katare, S. (2017, December). Parallelism in C++ programs targeting objects. In 2017 International Conference on Advances in Computing, Communication and Control (ICAC3) (pp. 1-6). IEEE.

[37] Cooper, K., & Torczon, L. (2012). Scanners, Engineering a compiler. Elsevier. 25-82.

[38] Aithal, P. S., & Pai T, V. (2017). Opportunity for Realizing Ideal Computing System using Cloud Computing Model. *International Journal of Case Studies in Business, IT and Education (IJCSBE)*, *1*(2), 60-71.

[39] Aithal, P. S., & Pai T, V. (2016). Concept of Ideal Software and its Realization Scenarios. *International Journal of Scientific Research and Modern Education (IJSRME)*, *1*(1), 826-837.

****