Journal of Advanced Computer Science & Technology, 8 (2) (2019) 50-57

Journal of Advanced Computer Science & Technology

Website: www.sciencepubco.com/index.php/JACST

Research paper

A survey about various generations of lexical analyzer

Zakiya Ali Nayef *

Department of Computer Science Nawroz University Duhok, Kurdistan Region – Iraq
**Corresponding author E-mail: Zakiyaduhoki87@gmail.com*

Abstract

Lexical analysis helps the interactivity and visualization for active learning that can improve difficult concepts in automata. This study gives a view on different lexical analyzer generators that has been implemented for different purposes in finite automata. It also intends to give a general idea on the lexical analyzer process, which will cover the automata model that is used in the various reviews. Some concepts that will be described are finite automata model, regular expression and other related components. Also, the advantages and disadvantages of lexical analyzer will be discussed.

Keywords: *Finite Automata; Regular Expression; Lexer; DFA; NFA; PDA and Turning Machines.*

1. Introduction

Lexical analysis is the procedure used in changing a series of characters into a series of tokens. A program that operates lexical analysis is called lexer, scanner, a lexical analyser, or tokenizer. A lexer sometimes exists as one function which is called by another function or a parser. it can also be joint with the parser in scanner less parsing Lexical analyser is broadly used in variety of software such as compiler for programming language. Furthermore, the first phase of a compiler is the Lexical analysis. Its function is to turn a raw character or byte input stream coming from the source file into a token stream by dividing the input into pieces and discarding irrelevant details.

1.1. Definitions on lexical analyser page layout

The following are several definitions related to the models, components and concepts involved in the lexical analyser operation:

a) Lexical analyser

A lexical analyser (lexer) can behave as pattern recognition engine which separates string of individual characters into tokens [10]. The spaces, newlines, comments, and others none related elements that separate the tokens will be removed [11]. It works together with parser in order to categorize the token based on the regular expression. While an example, in a compiler, the lexical analyser would classify the tokens into regular expression for instance INTEGER, DOUBLE, IDENTIFIER and etc.

b) Lex

It refers to the set of constructs and buffered input routines that translates regular expression from token into lexical analyzer [6].

c) Token

It is the fundamental unit in languages element or lexical which is no longer able to be divided [10]. In programming language, it is a series of characters with collective meanings and integer values which represent the lexeme. The categories of token are type token (num and id), alphabetic tokens (keywords for example DATE and FINAL) and punctuation token (IF, return and void). These categories typically used in programming languages. In addition, newlines and empty space are not considered as tokens as it will be removed.

d) Pattern

Pattern acts as a set of rule which is connected with the token so as to describe the input string by regular expression. As an instance in programming language of Pascal, the identifier token “id” can be describe by pattern as “letter (letter | digit)*”.

e) Lexeme

Lexeme refers to the original string which is coordinated by the example of a token. As an illustration, the RELATIONAL token in JAVA contains four lexemes; <, >, <= and >=. Therefore, it will return RELATIONAL token to the parser, at any time the lexical analyzer read any of these lexemes.

f) Finite automata

Finite automata can be divided into deterministic finite automata (DFA) and non-deterministic automata (NFA) [2]. Both of these sub models can be defined as a set of five elements which are K, ∑, f, S and Z. The definition for each element for these sub models are shown in figure 1.

On the other hand, each of these sub models is interchangeable. An NFA can be translated into DFA and DFA can be more simplified by using simplification technique [9].

Element s	NFA	DFA
Σ	L is a finite symbol set	
S	S is the subset of K. S is a nonempty start state set	S ∈ K, S is the unique start state
K	K is a finite symbol set, each element of which is called as a state	
F	F is the mapping from K*Σ to 2^K	F is the single value mapping from K*Σ to 2^K
Z	Z is a subset of K where it is refers to accepting state and final state	

Fig. 1: Differences between NFA and DFA (Adapted from [2], [7]).

g)Regular expression

Regular expression is used to explain the rule that guides a set of character string. The rules described by regular expression are easier and suitable to be processed by the computer [2]. The quality of the production of regular expressions can influence the efficiency of the lexical analysis process [4].

Operator	Explanation
$	The end of line
r1/r2	r1 followed by r2
r1 | r2	r1 or r2
(r)	r(used by grouping)
r+	Means one or more occurrences of r
r*	Means zero or more occurrences of r
r?	Means r is optional
"s"	Means string s literally
[s]	Mean s defining as character class
^	The beginning of line
.	Means any character but a newline
\c	character c literally (means used when c would normally be use as a lex operator)
[^s]	Means to matching character not in s character class
[a-b]	Used to defining a range of characters (a until b) in a character class
r{m,n}	Means m to n occurrences of r

Fig. 2: List of Regular Expression Operators.

1.2. Why we use lexical analyzer

The lexical analyzer is the only phase that processes input character by character, so speed is critical. Lexical Analyzer is mainly the part of compiler which:

- Lexemes are Translates into tokens (for compilation references they are arranged in a symbol table) with the help of Lex,
- To serving token requests it communicates with parser.
- Skips over white spaces by discarding comments
- Allows parser to detect errors by keeping track of current line number.

Any time you are converting an input string into space-separated strings and/or numeric values, you are performing lexical analysis. Writing a cascading series of else if (strcmp (...) ==0). Statements counts as lexical analysis. Even such nasty tools as sscanf and strtok are lexical analysis tools. You'd want to use a tool like flex instead of one of the above for one of several reasons:

- The error handling can be made much better.
- Flexibility in handling different things you recognize with flex.
- Lex scanners are faster. If you are parsing a lot of files, and/or large ones, this could become important.

1.3. Advantage and disadvantage of lexical analyzer

1.3.1. Advantages of lexical analyzer

- Paser writing using lexical analysis is much easier.
- The parser can start with tokens and concern itself only with syntactical matters. Instead of having to build up names such as"net_worth_future" from their individual characters, this could lead to efficiency of the program or efficiency of the execution.
- However, since the lexical analyzer is the subsystem that must examine every single character of the input,
- It can be a compute-intensive step whose performance is critical, such as when used in a compiler.
- Wrong words are never seen by the parser. It is more efficient for the parser to handle words, not characters.

1.3.2. Disadvantages of lexical analyzer

1) At the lexical level alone few errors are detectable.
2) Lexical analyzer has an exceptional restricted view of the source text.
3) Lexical analyzer cannot tell whether a string is misspelt from a keyword or an identifier.
4) The lexical analyzer cannot detect characters that are not in the alphabet or strings with no pattern.
5) The lexical analyzer stops when an error is found, (but other actions are possible).
6) Lexical analyzer maintenance can be complicated and is not very efficient.

7) Lexical analyzer can be time-consuming.

2. Related work

Lexical analyzer is extensively used in a lot of area of research. These are the many of the researched that had been done. It is a very challenging approach in query matching on XML stream as the query data is enormous and requires a lot of processing time. Syntactic Twig-Query Matching (STQM) that used the concept of parser and lexical analyzer is able to process the queries on XML and returning the results immediately and continuously [1]. Mongolian language is adhesive and contains huge amount of dictionary. In order to identify the words in speech recognition, a lexical analyzer is required. A Mongolian lexical analyzer with the usages of dictionaries and NFA methods is proposed to improve the speed of analyzing the language [3]. GLAP model had been proposed to reduce the analysis of time complexity and the design of lexical analyzer [6]. In this model, it focuses on a very restricted sub-set of the entire dictionary in least cost [6]. Lexical analyzer translates lexemes into token via Lex which communicates with parser for serving token requests. After that, it removes the comments and skips over white spaces. It also will monitor the current line number so that the parser can indentify incoming error [12]. Braune et al. [13] in this study describes learning software that visualizes the different transformations; on the other hand, it used a fixed instance and consequently it amounted to a “canned demo” only. JFLAP [14] is interactive learning software meant to focus on automata theory. It visualizes the main algorithms but only partly automates the construction and instead guides the students during the algorithms, warning them about any errors. The GaniFA applet [15] focused on the compiler construction rather than aiming at automata theory. The HaLeX library[16] provides some functions to represent , manipulate REs and FAs and Haskell datatypes . on the other hand, its focus is on a particular formalization of the algorithms, instead of visualization, and it provides a built-in dot graph and output that is not appropriate for interactive visualization. jFAST [17] did not support lexical specifications using REs and did not visualize any of the transformations between the different representations. it only allowed the simulation of different types of finite state machines (including DFAs and NFAs) and interactive construction.

3. Review on different applications of lexical analyzer generators

Lexical analyzer (lexer) uses finite automata to perform its operation. The models of finite automata involved are DFA and NFA. The common steps for the generation of lexer are as follow: Implementation

- Step 1: Specify the kind of token from the input language with regular expression.
- Step 2: Translates the regular expression into NFA
- Step 3: Simplified and convert the NFA into DFA
- Step 4: Minimizing the DFA

Lexical analyzer generator is a software tool that automatically constructs a lexical analyzer from a related language specification. While a typical lexical analyzer generator is a Lex and a Lexical analyzer generating tool is a Lex compiler.

3.1. Lexical analyzer generators

From the review on 10 papers the following methods were used to generate a lexical analyzer from a related language.

1) Finite automata[2]: The steps used in this study where:

a) Using Finite Automaton to Construct the P LlO which is the Lexical Model

- Using Regular Expressions to Describe the Lexical Grammar of PLIO
- Translating the lexical grammar of PLIO given by regular expressions to a DFA. and also translating lexical grammar to lexical model.

b) Construct the Control Program

The process of identifying words is described by the DF A. The control program and the DF A act as the lexical an alyzer which is more effective and simple.

Fig. 3: The Control Program.

2) Mongolian language [3]: This study worked on Mongolian Lexical Analyzer which was introduced based on Non-deterministic Finite Automaton NFA. MABRF is an eight-tuple, whose main function is to recognize the Mongolian words by a representation forms. The automaton comprises of one stem automaton together with a set of suffix automatons. However, MABRF suffixes and stems are linked by pointers.

In the construction of the MABRF, each word's lexical information was stored in e states as shown below. In a stem state, the information, such as part of speech, segmentation, representation forms subclasses(1,2,3),, type of suffix, nominal characters, vowel connection , transliteration in Latin, already has or does not have characteristics of the first character(consonant or vowel) and characteristics of the last character(hard consonant ,vowel and soft consonant are included.

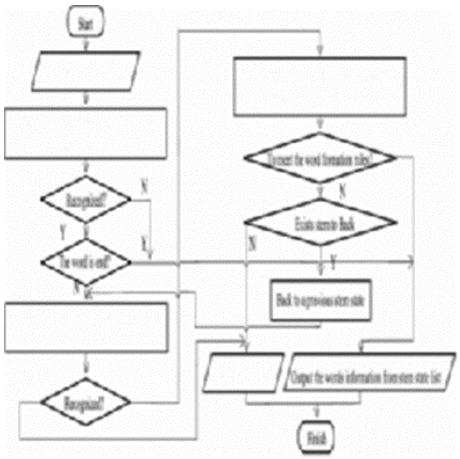

Fig. 4: Mongolian Flowchart for Word Recognition Algorithm.

MABRF is a complex concept. To explain the concept of the problem we selected some suffixes and stems to begin a part of the lexical analyzer. e.g.

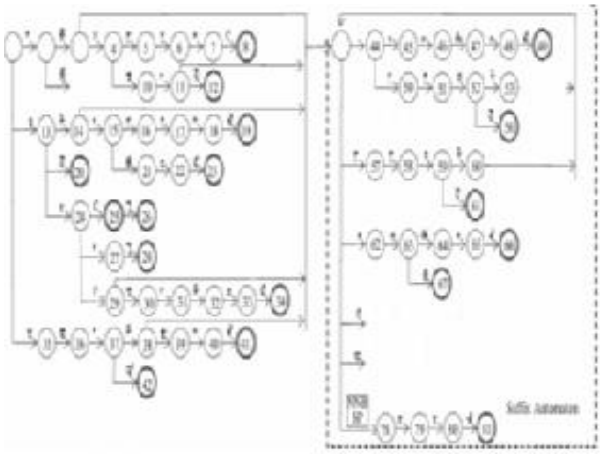

Fig. 5: Stem Automaton.

3) Regular Grammar: this study is similar to that of finite automata and follows the same rules. Which involves building the model for a lexical analyzer by using regular grammar? In build this model by using the regular grammar, The regular grammar that explains the lexical model can be translated into a deterministic finite automaton DFA. The process of identifying words is explained by the DF A. The control program and the DF A act as the lexical analyzer which is more effective and simple.
4) GLAP model [6]: This study an algorithm was designed for GLAP model in order to generate tokens this concept was named Tokenizer and is explained below.

Algorithm: Tokenizer(S)
Where S = Input string.
Output: A set of tokens
Step 1: Initialize S.
Step 2: Define the symbol table.
Step 3: Repeat while scanning (left to right) S is not completed
i) If empty space (blank)
a) Ignores and eliminate it.
ii) If operator op // relational, arithmetic, etc.
a) Find its type.
b) Write op.
iii) If keyword key // while, if, for, etc.
a) Write key.
iv) If identifier id // a, b, c, etc
a) Write id.
v) If special character sc // (,), etc.
a) Write sc.
Step 4: Exit

The above Tokenizer () algorithm was implemented in Turbo C++ Version 3.0 and the simulation of the input was done using both a valid and an invalid string.

5) VLex: VLex is an incrementally visualizing lexical analyzer generator that allows user to determine at which locations (and thus with which steps) the underlying generator algorithms proceed. VLex is implemented in C# and uses MS Visual Studio 2008. It is freely available from www.vlex-tool.net. Its central data structure is the automaton, which is shared between all three types (i.e., NFA, DFA, and minimized DFA).Each of the major algorithms (NFA and DFA construction as well as DFA minimization) is im-

plemented as a separate module. VLex visualizes the algorithms typically implemented in a lexical analyzer generator in the lex tradition, i.e., converting an RE via an NFA into a DFA and then minimizing this DFA. The visualization works incrementally, and the user can choose any location to continue the algorithms; for example, the user can pick a regular (sub-) expression to drive the RE-to-NFA conversion, or a DFA-state and input character to drive the NFA-to-DFA conversion. VLex can also animate the different automata during the scanning phase.

VLex uses a custom graph drawing algorithm to achieve a compact layout and that is suitable for the incremental visualization style.

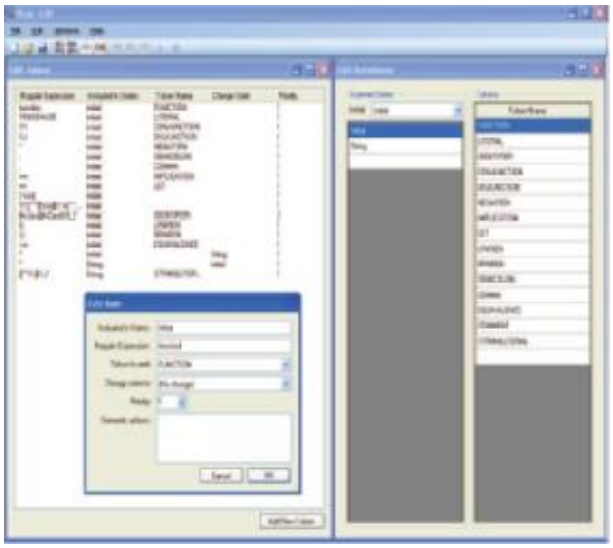

Fig. 6: Graphical User Interface of A VLEX.

6) FLEX: This study defined FLEX as a fuzzy lexical analyzer generator that has all the characteristics of LEX except the symbol \/" in the Lex expressions should be written as \n="now.

FLEX has the following characteristics:

1) Any LEX expression can be followed with a number between 0 and 1 and a \/".

The degree of the expression is the \/n" part, where n is a number between 0 and 1. Degrees are not nested, i.e., no sub exprsions is allowed to be written with a degree if an expression is written with a degree. For example, fa=0:6gbc=0:7 is an invalid expression. Degrees cannot be written within a pair of \ [" and \]".

2) Apart from the three parts in a LEX program, a fourth part can be used to denote actions for different ranges of degrees.

Where QUESTION, REJECT and ACCEPT are all key words for FLEX

a) Strings are accepted as tokens;
b) Questions on-line are given to the users; if the user answers \yes'' the string is accepted and if the user answers \no "it is rejected;
c) If the fourth part is not given, the string is rejected, the following rules are assumed by default:

7) (jFAST): This research described the utility and design of a Java Finite Automata Simulation Tool (jFAST).jFAST is an instructional software package used as an easy learning and easy-to-use software tool for teachers and students in order to determine and see the insights of a finite state machines. It is designed to be a equivalent alternative to the widely and highly used JFLAP tool. The jFAST software assists teachers to form an easy or complex Finite State Machines that be displayed to students by the use of a classroom projector. jFAST also enables teachers to distribute Finite State Machines to students through email, so that students can actively learn through hands-on manipulation and visual simulation. Students can also create their own Finite State Machines using the well-known, instinctive drag-and-drop graphical user interface of current computing.

jFAST assists in the simulation of various regular finite state machines. The FSMs implemented in jFAST comprises of: NFA, DFA, Turing Machine, Pushdown Automata and State Machine.

The figures below show a Jfast graphical user interface implementing an NFA and PDA

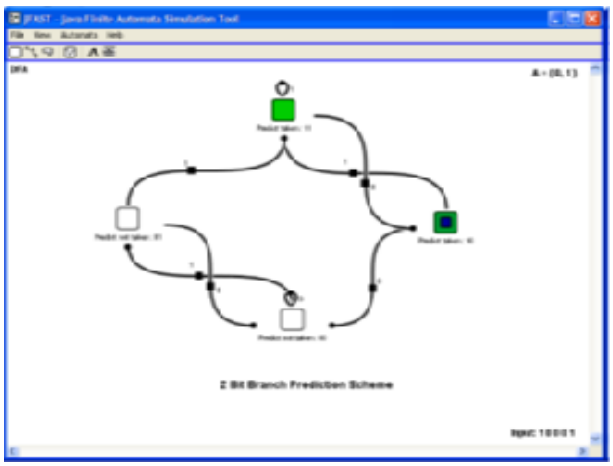

Fig. 7: jfast Design Interface.

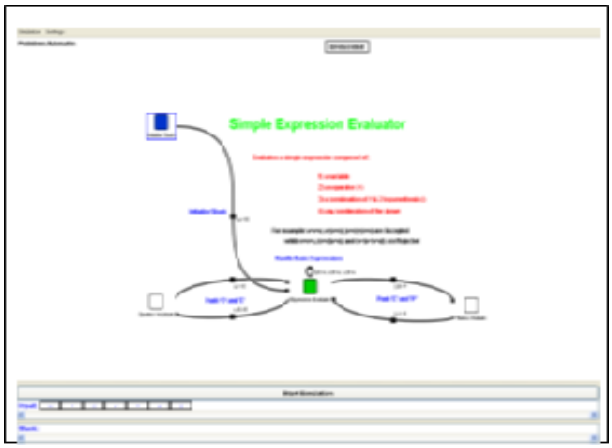

Fig. 8: jFAST Simulation Interface.

8) Jflap: A New Approach in Jflap is introduced in this study for converting NFA to RE this JFLAP 4.0 new approach used in converting an NFA to a regular expression (RE) is more visible and simpler to comprehend for users than that of JFLAP 3.1. In this approach it first begins with an NFA. E.g., the figure below shows that an NFA must be in an exact format with exactly one final state, which should not be the starting state and must also have exactly one transition coming from every pair of states. JFLAP assists the user in cutting down several transitions from pairs of states and by adding them between two states that did not have a transition previously. All states excluding the starting state and final state are removed one after the other. From the example below we start removing state q2.

When q2 is eliminated, its expression has to be included or added into all the other transitions. The figures below shows an example of the regular expressions that will be included or added on the arcs after eliminating state q2. Figure 9 shows the resulting NFA after eliminating state q2. Likewise, state q1 is eliminated and the result of the regular expression is obtained.

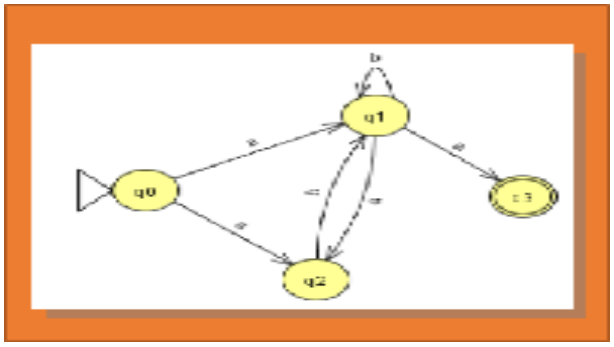

Fig. 9: Converting an NFA.

Table 1: Table in Eliminating State Q2

From	To	Label
0	0	ø
0	1	a+ab
0	3	ø
1	0	ø
1	1	b+ab
1	3	a
3	0	ø
3	1	ø
3	3	ø

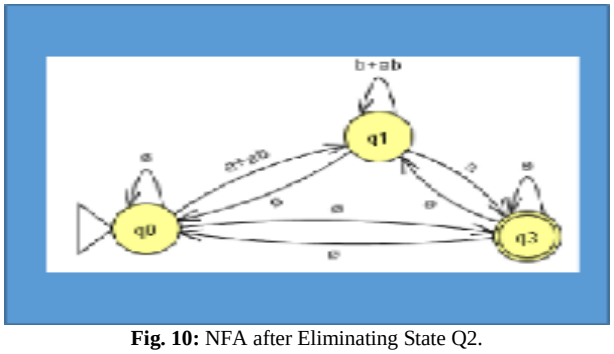

Fig. 10: NFA after Eliminating State Q2.

The equivalent regular expression ((a+ab) ((b+ab)*))

3.2. Comparing the different lexical generators

The following lexical generators are compared based on how the apply to automata theory.

Table 2: Summary of Different Lexical Generators

METHODS IMPLEMENTED	DFA	NFA	REG-X	PDA	TURNIG-MACHINE
FINITE AUTOMATA	✓		✓		
MONGOLIAN		✓			
REGULAR GRAMMAR	✓		✓		
GLAP MODEL					
VLEX	✓	✓	✓		
JFAST	✓	✓		✓	✓
JFLAP		✓	✓		
FLEX	✓	✓	✓		

From the review on the different lexical generators it was observed that all the works implemented were mainly for educational purposes.

Table 3: Purpose for Different Lexical Generators

METHODS IMPLEMENTED	PURPOSE
FINITE AUTOMATA	General educational purpose
MONGOLIAN	For speech recognition using dictionary words
REGULAR GRAMMAR	Used in softwares for analysing e.g. document editor
GLAP MODEL	Used for time complexity analysis
VLEX	Used as a visualization tool for teachers and students
JFAST	Used as a visualization tool for teachers and students
JFLAP	Used as a visualization tool for teachers and students
FLEX	Not defined

4. Conclusion

Lexical analysis is used in so many softwares such as compiler and speech recognition. It has played an important role in programming aspects and finite automata is said to be the methodology behind the lexical analysis. As a result of this it is undeniable that finite automata serve as the core concept of all the softwares in the world since these softwares are implemented by programming languages which are compiled by a compiler that is used in all lexical analyser processes

The review done in this study used the principle of lexical analysis in modelling lexical analyser generators. It was observed that not many works was done on this study over a period of 12 years. The articles found for this study where published between the years 2000 to 2012 but were limited. However, from the review more work was modelled for DFA, NFA and REGULAR EXPRESSIONS. And only one discussed on PDA and TURNING MACHINES.

In conclusion, for the future more works should be done on lexical analyzer generators focusing on PDA and TURNING MACHINES. Also an improvement on the limitations found in the use of lexical analysers should be improved in further studies by using other approaches to still simplifier lexical models.

References

[1] C.-P. Chou, K.-F. Jea and H.-H. Liao, "A syntactic approach to twig-query matching on XML streams," The Journal of Systems and Software, vol. 84, p. 993–1007, 2011. https://doi.org/10.1016/j.jss.2011.01.033.

[2] H. Luo, "Research of Using Finite Automaton in the Modeling of Lexical Analyzer," in International Conference on Information Management, Innovation Management and Industrial Engineering, 2012. https://doi.org/10.1109/ICIII.2012.6339811.

[3] S. Loglo, Sarula and HuaShabao, "Research on Mongolian Lexical Analyzer Based on NF A," in IEEE, 2010. https://doi.org/10.1109/ICICISYS.2010.5658760.

[4] Z. Gejun, S. Yuqiang, Y. Ruimin and G. Yuwan, "A Simplification Algorithm of Regular Grammar Production," in International Conference on Information Science and Engineering, 2009.

[5] T. Kos, T. Kosar, J. Knez and M. Mernik, "Improving End-User Productivity in Measurement Systems with a Domain-Specific (Modeling) Language Sequencer," in CEUR Workshop Proceedings, 2010.

[6] B. Bhowmik, A. Kumar, A. K. Jha and R. Kumar Agrawal, "A New Approach of Complier Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages," International Journal of Computer Applications, vol. 6, no. 11, pp. 21-25, 2010. https://doi.org/10.5120/1116-1462.

[7] A. Mateescu, A. Salomaa, K. Salomaa and S. Yu, "Lexical Analysis with a Simple Finite-Fuzzy-Automaton Model," Journal of Universal Computer Science, vol. 1, no. 5, pp. 292-311, 1995. https://doi.org/10.1007/978-3-642-80350-5_28.

[8] G. Dodig Crnkovic and M. Burgin, "Unconventional Algorithms: Complementarity of Axiomatics and Construction," Entropy, vol. 14, pp. 2066-2080, 2012. https://doi.org/10.3390/e14112066.

[9] HE Yan-xiang, WU Chun-xiang, WANG Han-fei. Compile Principle [M]. Beijing Machinery Industry Press, 20 10.

[10] Alfred V.Aho, Ravi Sethi, Jeffery D. Ullman, Addison- Wesley, 2007. Compilers- Principles, Techniques, and Tools.

[11] Torben Ægidius Mogensen, May 28, 2009. Basics of Compiler Design", lulu, Extended Edition.

[12] William M. Waite, Assad Jarrahian, Michele H. Jackson, Amer Diwan, 2006. Design and Implementation of a Modern Compiler Course, ACM 1595930558/06/0006. https://doi.org/10.1145/1140124.1140132.

[13] B. Braune, S. Diehl, A. Kerren, and R. Wilhelm.Animation of the generation and computation of finite automata for learning software. In Proc. 4th Intl. Workshop Implementing Automata, LNCS 2214, pp. 39–47. Springer, 2001. https://doi.org/10.1007/3-540-45526-4_4.

[14] R. Cavalcante, T. Finley, and S. H. Rodger. A visual and interactive automata theory course with JFLAP 4.0. In Proc. SIGCSE'04, pp. 140–144. ACM Press, 2004. https://doi.org/10.1145/971300.971349.

[15] S. Diehl, A. Kerren, and T. Weller. Visual exploration of generation algorithms for finite automata on the web. In Proc. 5th Intl. Conf. Implementation and Application of Automata, LNCS 2088, pp. 327–328. Springer, 2000. https://doi.org/10.1007/3-540-44674-5_29.

[16] J. Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In Proc. ACM Workshop on Functional and Declarative Programming in Education, pp. 133–140. University of Kiel Technical Report 0210, 2002.

[17] T. M. White and T. P. Way. jFAST: A Java finite automata simulator. In Proc. SIGCSE'06, pp. 384–388. ACM Press, 2006. https://doi.org/10.1145/1121341.1121460.