Scholastic Seed Inc.

www.gjeis.com

global Journal of ENTERPRISE INFORMATION SYSTEM

A Finite State Automaton is a Tool to Represent Formal Language

– Bhawna Kaushik*
Assistant professor, G.L.Bajaj Institute of Management
kaushikbhawna311@gmail.com https://orcid.org/0000-0003-3688-5341

– Mayank Saini
Assistant professor, G.L.Bajaj Institute of Management
sainimayank0211@gmail.com https://orcid.org/0000-0002-3820-4485

ARTICLE HISTORY

Paper Nomenclature: View Point (VP)
Paper Code: GJEISV15I1JM2023VP2
Submission at Portal (www.gjeis.com): 13-Jan-2023
Manuscript Acknowledged: 15-Jan-2023
Originality Check: 24-Jan-2023
Originality Test (Plag) Ratio (Ouriginal): 07%
Author Revert with Rectified Copy: 27-Jan-2023
Peer Reviewers Comment (Open): 31-Jan-2023
Single Blind Reviewers Explanation: 12-Feb-2023
Double Blind Reviewers Interpretation: 20-Feb-2023
Triple Blind Reviewers Annotations: 27- Feb-2023
Author Update (w.r.t. correction, suggestion & observation): 28-Feb-2023
Camera-Ready-Copy: 14-Mar-2023
Editorial Board Excerpt & Citation: 19-Mar-2023
Published Online First: 31-Mar-2023

ABSTRACT

Purpose: In an introductory formal languages course, upper-level undergraduates and first-year graduate , students are introduced to concepts like automata theory, grammar, constructive proofs, computational efficiency, and decidability. These subjects are difficult or daunting for many students learning to code since they are on the periphery of the field of Computer Science. This misconception is understandable since students are often tasked with designing and providing accurate machines and grammar without the experimental opportunities and real-time feedback crucial to their development as learners. The purpose of the present research work is that tools for creating computations should be included in the instruction of computation theory.

Design/Methodology/ Approach: The present study is mainly based on secondary data. The data and relevant statistics for this study have been collected from different sources.

Findings: It details the deployment and usage in the classroom of a library called FSM, which is meant to provide students with the chance to explore and test their ideas using state machines, grammar rules, and query language. Before committing to a rigorous demonstration of correctness, students can conduct randomized tests.

Originality/ Value: This research shows students may conduct usability tests on their ideas like that used in computer programming classes. Students may quickly include their algorithmic developments in their constructive proofs thanks to the library's convenient implementation options.

Paper type: View Point.

KEYWORDS: Finite State Automaton | Formal Language | Automata Theory

**Corresponding Author (Bhawna Et. Al)*

- Present Volume & Issue (Cycle): Volume 15 | Issue-1 | Jan-Mar 2023
- International Standard Serial Number: Online ISSN: 0975-1432 | Print ISSN: 0975-153X
- DOI (Crossref, USA) https://doi.org/10.18311/gjeis/2023
- Bibliographic database: OCLC Number (WorldCat): 988732114
- Impact Factor: 3.57 (2019-2020) & 1.0 (2020-2021) [CiteFactor]
- Editor-in-Chief: Dr. Subodh Kesharwani
- Frequency: Quarterly
- Published Since: 2009
- Research database: EBSCO https://www.ebsco.com
- Review Pedagogy: Single Blind Review/ Double Blind Review/ Triple Blind Review/ Open Review
- Copyright: ©2023 GJEIS and it's heirs
- Publishers: Scholastic Seed Inc. and KARAM Society
- Place: New Delhi, India.
- Repository (figshare): 704442/13

Introduction

Automata, mathematical models of classical computing, have been a significant part of theoretical computer science [1]. It all began with a landmark work by Kleene, and in only a few short years, this area of mathematics has grown into a vibrant field of study. Finite automata have always been fundamental to the study of computer science. They are so popular because they capture something essential, as seen by the many distinct ways the family of rational languages described by finite automata has been characterized. The relationship between finite automata and associated applications in computer science is a great illustration of how theory and practice may fruitfully interact. Programming language theory, compiler building, switching circuit design, computer controller, neural network, text editor, and lexical analyzer all owe much to finite automata.

Models and analyses are used in studying computers and computing in theoretical computer science. It involves various subfields of computing to create models and analytical techniques.

The field of automata theory investigates abstract machines used for computing. In the 1930s, before computers were widely available, A. Turing researched an abstract machine that, in terms of what it could calculate, had all the powers of modern computers. The purpose of Turing's work was to provide a clear description of the limits of what can and cannot be done by computing machines; his results hold not just for his abstract Turing machines but also for the practical computers in use today.

Researchers in the 1940s and 1950s focused on simpler machines, which we now term "finite automata." These automata were first proposed as a model for brain activity, but they have also been useful in various other contexts. For example, they have been implemented in tools for designing and verifying the behaviour of digital circuits, creating lexical analyzers, scanning large bodies of text, and validating systems of various kinds with finite states. Aside from this, linguist N. Chomsky also started looking at formal "grammar" in the late '50s. These grammars constitute the foundation for some crucial parts of today's software and have tight ties with abstract automata.

These theoretical advancements mentioned above have clear, practical implications for modern computer scientists. Certain ideas, such as finite automata and certain types of formal grammar, are employed in the development of crucial pieces of software.

In this introductory chapter, we first quickly review the fundamental ideas of automata theory before moving on to the fuzzy sets and the three main extensions to those sets that are necessary for our investigation.

The Central Concepts of Automata Theory

We begin by discussing the core concepts and terminology used throughout automata theory. Among these ideas are the alphabet, threads, and languages.

Definition: An alphabet is a collection of symbols that is not empty. Common alphabets for it are indicated by the symbol, and they are as follows:

1. The symbol represents binary 0 and 1.

2. $\sum$ = a, b, c,... z, the set of all lowercase letters.

3. All ASCII characters or all readable ASCII characters.

Definition: A string (also known as a word) is a series of characters, usually of fixed length and drawn from a common alphabet. Binary strings like 011 and 011011 are used as examples. A string's size equals the total number of symbol places it contains.

The length of 011 is 3, for instance. A chord length is often written as |w|. For instance,

|011-11| = 6.

Definition: e, for "empty," is a specific case of string in which no symbols appear. The length of e is defined to be zero. Any letters in the alphabet may be used for this string.

The standard notation for collecting all strings in an alphabet is ∑ *. I'll give you an example:

$$\Sigma^* = \{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\} \text{ or } \Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots.$$

We would prefer not to include an empty string in some situations. The symbol $\sum^+$ represents the set of all lines that are not empty.

As a result, two equivalents are as follows:

1 . $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots.$

2 . $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}.$

∑

Definition: In this example, we'll use two strings x and y. Then the string generated by duplicating x and appending a copy of y is referred to as xy, or the connectives of x and y.

Example: Consider the expressions x = 0111 and y = 011011. After that, both xy and yx equal 0110110111. For each w, the ratios ew = we = w hold. That is, e is the identifier for concatenation.

Definition 1.2.5: A collection of strings drawn from the same pool. ∑*, where ∑ languages are defined by their alphabets. If ∑ is an alphabet, and * ∑ ⊆L, Thus, L is a language beyond. Remember that a language that exceeds does not have to include cords that contain all the components of ∑. The collection of binary integers whose value would be a prime: 10, 11, 101, 111,..., is an abstract example.

1. The set of binary numbers whose value is a prime: {10, 11, 101, 111, …}.
2. Σ^* is a language for any alphabet Σ.
3. ϕ, the empty language, is a language over any alphabet.
4. $\{\varepsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Note that $\phi \neq \{\varepsilon\}$; the former has no strings and the latter has one string.

All alphabets have a fixed number of letters. Hence there is only one serious limitation on what may be considered a language. Even though there is theoretically an endless number of strings possible in a language, in practice, languages are limited to just the letters of a finite alphabet.

Finite State Automata

In this part, we'll go through what a finite automaton is, how its types are related to one another, what a regular language is, and how to minimize a finite automaton. The philosophy of computer languages and the development of compilers relied heavily on finite automata. The behavior of Discrete Event Systems may be modeled using robots (DES). Like systems are commonly found in manufacturing, database transaction management, telephony or computer networks, and therapeutic systems like patient monitoring. Applications of finite machine learning may be found in computing science, mathematics, and algebra. In an online search, finite automata are used for knowledge discovery from the text to identify a predefined keyword list.

Definition: Mathematically, a finite state animal is a pattern with limited input and output.

An FSA may be characterized analytically as a 5-tuple.

$A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite nonempty set of states; Σ is a finite nonempty set of inputs called input alphabet; $\delta : Q \times \Sigma \to Q$ a function called transition function. This describes the change of states for each alphabet of Σ during the transition; $q_0 \in Q$ is the initial state; and $F \subseteq Q$ is the set of final states (it is assumed here that there may be more than one final state).

A string $x \in \Sigma^*$ is accepted by a finite automaton A if $\delta(q_0, x) = q$ for some $q \in F$. The final state is also called an *accepting state*.

An expanded transition function is required to provide the concept of an automaton's language with high precision. That's the operation that accepts a string as input (represented by w) and returns the same state (represented by q). The symbol for this is by δ^* and is defined as $\delta^*(q, w) = \delta(\delta^*(q, x), a)$ for $w = xa \ \forall \ x \in \Sigma^*, a \in \Sigma$.

NOTE: The output of a memoryless automaton has no other influence than the input. A robot with finite memory is one in which the result relies on the state in extra to the information. A Moore machinery's outputs are independent of any input other than the machine's current state. A Mush machine is a kind of automata in which the result at any given time is contingent on both the current state and the input.

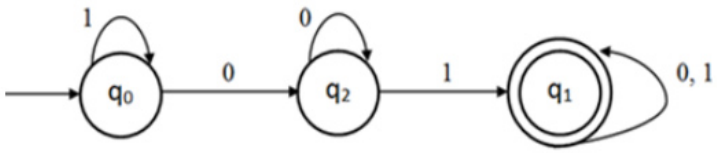


Figure 1.1: Transition diagram for a DFA

Table 1.1 displays the transformation table associated with the program. In this case, a circle represents the agreeing states, but also an arrow represents the initial state.

Table 1.1: Transition table of Figure 1.1

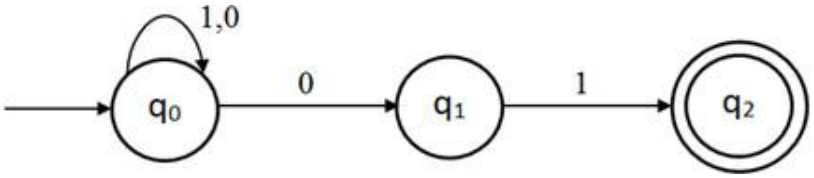
Q / Σ →	0	1
→ q0	q2	q0
(q1)	q1	q1
q2	q2	q1

The language of a DFA A_1 is denoted by $L(A_1)$, and is given as $L(A_1) = \{w \mid \delta^*(q_0, w) \in F\}$.

That is, the array of characters w that route to a passage from the initial state q0 to one of the terminal (accepting) phases is the languages of A1 (in terms of the transition diagram, it is the set of all labels along all the paths that lead from the starting state to any accepting state).

Definition: A "5-tuple" is an example of a nondeterministic discrete automaton (NDFA).

$A_2 = (Q, \Sigma, \delta, q_0, F)$, where Q , Σ , q_0, F are same as those in DFA and $\delta : Q \times \Sigma \to 2^Q$ a function called transition function. This takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.



Keep in mind that an NDFA and a DFA are otherwise identical, with the main distinction being in the number of states involved (in the former instance, there are many states involved, while in the latter case, there is only one).

Example

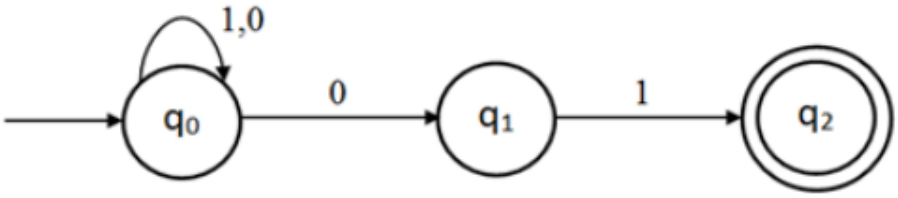


Figure1: Transition diagram for NDFA

Formally, the NDFA shown in Figure may be outlined as $(\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, where motivated strategies are defined by the table.

Table: Transition table of Figure 1.2

Q / Σ	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	ϕ	$\{q_2\}$
(q_1)	ϕ	ϕ

An NDFA's transitioning function may be specified in the same way as a DFA's, through transition tables. There is one key distinction: even if an NDFA set consists of a single entity, each item in the table represents a set. Additionally, the correct entry seems to be the empty set, , if the supplied input symbol does not have a movement from a given state. All states with arrows and the circles represent the initial and final states, respectively.

L (A2) denotes the language of an NDFA A2, and the following is a definition of that language:

$$L(A_2) = \{w \mid \delta^*(q_0, w) \cap F \neq \phi\}.$$

That is, the language of A_2 is the set of all strings $w \in \Sigma^*$ such that $\delta^*(q_0, w)$ contains at least one accepting state.

Equivalence of DFA & NDFA

Surprisingly, any language represented by some NDFA may also be expressed by some DFA, despite the number of countries that a NDFA is simpler to design than a DFA. In addition, the DFA often contains more transitions than the NDFA but approximately the same number of stages. While the greatest NDFA for a given language has exactly n states, the cheapest DFA may have 2n states. The built DFA enters a state that corresponds to array of states of NDFA after scanning the input signal sequence w. That's because the sets which include @ at least one allowing state of both the NDFA are also acceptable states of the DFA, we may deduce that it DFA and NDFA allow its same strings and, by extension, the same language.

The theorems that follow establish the above statement.

Theorem 1.1: If $D=(Q_D,\Sigma,\delta_D,\{q_0\},F_D)$ is the DFA constructed from NDFA

$N=(Q_N,\Sigma,\delta_N,q_0,F_N)$ by the subset construction, then $L(D)=L(N)$ [5].

Theorem 1.2: To the extent that a given language L is recognized by at least one Nationally Recognized Foreign Acknowledgement Authority (NDFA) [5], L is recognized by at least one DFA.

Formal Languages and Finite Automata

There are several uses for formal language theory in the field of Computer Science. In the early 1950s, linguists tried to define legitimate sentences accurately and describe their structural components. To establish a formal grammar, they sought to provide a rigors mathematical description of the laws of grammar. They reasoned that providing a detailed description of natural languages (languages like English, Hindi, etc.) would facilitate automatic machine translation. In 1956, it was Noam Chomsky [5] who first presented a formalized model of a grammar. Despite its apparent lack of use in expressing natural languages like English, it proved effective for characterizing computer languages.

Classification of Languages

We have, $G=\{V_N,\Sigma,P,S\}$ grammar's formal definition, where VN and represent sets of symbols such that S VN. In terms of the content of their outputs, Noam Chomsky categorized grammars into four broad categories (type 0 to type 3).

Before we get into the various forms of manufacturing, we need to define one thing.

Definition: In a play of this kind, $\tau A\rho\rightarrow\tau\alpha\rho$, , where A is a variable,, the left context,, the right "context, and, the replacement string.

Example:

a) In lmABlmn ,→lmAlmn lm is the left context, lmn is the right context, and AB .= α

b) In ε A , A and →AC are the left and right contexts respectively, and .ε = α Here the production erases C.

c) In ε→A , the left and right contexts are ε , and ε= α . The production erases A in any context.

Definition: A production without any restrictions is called *type 0* productions and a type 0 grammar is a phrase structure grammar having type 0 productions.

Definition: A production of the form ⟩⟨ | ⟩A | is called type 1 if ∑ ≠ ⟨ (here erasing of A is not allowed).

For example, AC → AcCb is a type 1 production".

When all of a grammar's outputs are 1s, it is said to be 1st-type, or frame of reference, and the language created by such a grammar is 1st-type, or context-sensitive (Lcsl).

Definition: "A type 2 production is of the form, where A → ⟨⟩

$A\in V_N$ and $\alpha\in(V_N\cup\Sigma)^*$ (here L.H.S. has no left or right context).

For example, bB→ ab, A →S are type 2 productions.

A grammar is said to be of type 2 or context free, if it contains only type 2 productions, and a language is called a type 2 or context free language (Lcfl) if it is generated by type 2 grammar.

Definition: A production of the form VN∈ aA,where A,B→ b,B →B ∑∈,a,b is called a type 3 production".

Type 3 or regular grammars create only other type 3 or regular products (here S is allowed, but in this instance S does not occur on the right-hand side of any other production), while Lrl languages are those that can only be formed by type 3 grammars.

Topics in Regular Expressions and Finite Automata To algebraically represent subsets of strings, tools like regular expressions come in handy. Those are the languages that finite state machines can understand (regular languages). Regular expressions over an alphabet are defined recursively as.

1. Regular expressions include any symbol at the end of a string (i.e., an element of), as well as the operators. and.

2. It is also a regular expression for two regular expressions, R1 and R2, to be joined together. Its notation is R1 + R2.

3. It is also a regular expression to concatenate two regular expressions, such as R1 and R2. Its symbolic representation is R1R2.

4. Regular expression R's closure or iteration, R*, is likewise a regular expression.

5. If R is a regular expression, then the order of evaluation of R is also a regular expression, denoted by (R).

6. Applying the aforementioned principles once or several times yields exact regular expressions over 6, which may then be used in a recursive fashion to create further regular expressions.

The following theorems describe the relationship between finite robots and regular expressions, as well as between reverse - engineering and regular expressions.

Theorem: The pattern R is registered by the transition system if and only if there is a route from the initial state to the end state with value w.

Theorem: A regular expression may be used to describe any set L that can be processed by a finite automaton M.

NOTE:

i. First, each regular expression may have an equivalent finite automaton, and any finite automata can have a regular expression.

ii. If two finite automata accept the same set of strings, we say that they are equivalent.

iii. If P and Q are regular expressions over, then they are comparable if and only if they represent the same set. Equalities between P and Q hold if and only if the respective finite automata are equivalent.

iv. The class of regular sets over is shown to be equivalent to the class of regular languages over. Further, we may build a finite automaton A accepting L whose: a. States correspond to variables if and only if G is a regular grammar producing the regular language L(G).

v. A0 is the starting point. b.

vi. P's productions are reflected in the transitions. c.

Pumping lemma is a theorem that proves that an input string must satisfy the criterion of being a regular set. Because of this theorem, we can "pump" several different input strings from a single string.

Conclusion

The FSM library equips users with the tools they need to create and test out state machines, grammatical structures, and conditionals. It lends credence to the idea that creating a constructive proof is the best way to establish the reality of a machine or language. Therefore, the method presented in the proof is one that can be realized using FSM. Teachers and students may now use digital tools in addition to traditional paper and pencil sketches to gain competence and identify flaws in a design. They may, instead, use FSM testing infrastructure to build tests with real-time feedback. Students may then build and create unit tests for their constructive algorithms, which not only reinforces their Computer Science education but also encourages active reasoning and the study of formal languages. There has been encouraging reaction from students, and the examples included in the piece came from the library. Our hope is that CS departments everywhere will embrace our method of combining instruction in the theory of computing with hands-on experience in building computational models.

In the future, we want to add additional constructors to the library, especially those that help with state reduction. We want to enhance the library by adding a user-friendly graphical user interface. We do not want students to use a graphical interface like those described in the linked work to build their own machines and grammars. Instead, we want students to keep up the practise of writing code in order to build machines and grammars, which can then be presented graphically in

order to animate execution and visualize their structure. Also, regular expressions and Turing machine extensions will have better support. Although the later machines can't do much more computing than a regular Turing Machine, they may simplify the implementation of certain designs, which is useful for classroom use. Finally, we want to transform the library into an embedded DSL (like Racket's hygienic macros [4]) that can be used in the classroom and where the proofs can be automatically verified by a computer (e.g., using tools like DrACuLa [2] and Coq [8]).

References:-

- Aho A. V. and Ullman J. D., "Foundations of Computer Science," Computer Science Press, New York, 1994. [5]. Hopcroft J. E. and Ullman J. D., "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley, 1979.
- Zadeh L. A., "Fuzzy sets," Inform. And Control, vol. 8, pp. 338-353, 1965.
- Hersh H. M. and Caramazza A., "A fuzzy set approach to modifiers and vagueness in natural language," J. Exp. Psychol., vol. 105, pp. 254-276, 1976.
- Klir G. J. and Yuan B., "Fuzzy Sets and Fuzzy Logic-Theory and Applications," Prentice Hall, Englewood Cliffs, NJ, 1995.
- Zimmerman H. J., "Fuzzy set theory and its Applications," Kluwer Academic Publishers, 1996.
- Dubois D., Ostasiewicz W. and Prade H., "Fuzzy sets: history and basic notions," in: D. Dubois, H. Prade (Eds.), "Fundamentals of Fuzzy Sets," The Handbooks of Fuzzy Sets Series, Kluwer, Dordrecht, pp. 21–124, 2000.
- Zadeh L. A., "Toward a generalized theory of uncertainty (GTU)–an outline," Information Sciences, vol. 172, no. 1, pp. 1–40, 2005.
- Walker C. L. and Walker E. A., "The algebra of fuzzy truth values," Fuzzy Sets and Systems vol. 149, pp. 309–347, 2005
- Novak V., "Are fuzzy sets a reasonable tool for modelling vague phenomena?," Fuzzy Sets and Systems, vol. 156, pp. 341–348, 2005.
- Tong R. M. and Bonissone P. P., "A linguistic approach to decision making with fuzzy sets," IEEE Trans on Syst., Man, Cybern. SMC-9, pp. 716-723, 1980.
- Ding X, Wang Z, Rovetta A, et al. Locomotion analysis of hexapod robot. In: Miripour-Fard B (ed.) Climbing and walking robots. IntechOpen, 2010, pp. 291–309.
- Tedeschi F and Carbone G. Design issues for hexapod walking robots. Robotics 2014; 3(2): 181–206.
- Zielinska T, Goh T and Chong CK. Design of autonomous hexapod. In: Proceedings of the first workshop on robot motion and control (RoMoCo), Kiekrz, Poland, 29 June 1999, pp. 65–69. New York: IEEE.
- Saranli U, Buehler M and Koditschek DE. Design, modeling and preliminary control of a compliant hexapod robot. In: Proceedings of 2000 IEEE international conference on robotics and automation (ed BS Carlisle), San Francisco, CA, 24–28 April 2000, pp. 2589–2596. New York: IEEE.
- Simpson J, Jacobsen CL and Jadud MC. Mobile Robot Control – The Subsumption Architecture and occam-pi. In: Welch P, Kerridge J and Barnes F (eds) Communicating process architectures. Amsterdam: IOS Press, 2006, pp. 225–236. 6. Li M, Yi X, Wang Y, et al. Subsumption model implemented on ROS for mobile robots. In: 2016 Annual IEEE systems conference (SysCon) (ed B Rassa), Orlando, FL, 18–21 April 2016, pp. 1–6. New York: IEEE.

Annexure 15.11

Submission Date	Submission Id	Word Count	Character Count
24-Jan-2023	(Turnitin)	3512	21408

Analyzed Document	Submitter email	Submitted by	Similarity
4.2 VP2_Bhawna _GJEIS Jan to Mar 2023.docx	kaushikbhawna311@gmail.com	Bhawna Kaushik	07%

turnitin
ORIGINALITY REPORT

7%	6%	1%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	archive.org Internet Source	3%
2	ir.amu.ac.in Internet Source	1%
3	www.jiit.ac.in Internet Source	1%
4	Lecture Notes in Computer Science, 2006. Publication	1%
5	www.careerage.com Internet Source	<1%
6	ipfs.io Internet Source	<1%
7	epdf.tips Internet Source	<1%
8	edoc.pub Internet Source	<1%
9	lib.unisayogya.ac.id Internet Source	<1%

Reviewers Memorandum

Reviewer's Comment 1: The paper titled "A Finite State Automaton is a Tool to Represent Formal Language" is a well-written and informative piece on the use of finite state automata in formal language representation. The author presents a clear and concise explanation of the key concepts and techniques involved in the design and implementation of these automata, making it accessible even to those with limited prior knowledge of the subject.

Reviewer's Comment 2: One of the strengths of this paper is the depth of knowledge demonstrated by the author. It is evident that they have a strong understanding of the subject matter and are able to explain complex ideas in a clear and concise manner. This expertise is further demonstrated by the way the author connects finite state automata to other concepts in computer science.

Reviewer's Comment 3: The important aspect of this paper is its attention to detail. The author provides a thorough explanation of the underlying principles and theories of finite state automata, as well as a step-by-step guide to constructing and testing these models. This level of detail is especially helpful for readers who are new to the field, as it provides a solid foundation for further exploration and experimentation.

Citation

Bhawna Kaushik and Mayank Saini
"A Finite State Automaton is a Tool to Represent Formal Language"
Volume-15, Issue-1, Jan-Mar 2023. (www.gjeis.com)

https://doi.org/10.18311/gjeis/2022
Volume-15, Issue-1, Jan-Mar 2023
Online iSSN : 0975-1432, **Print iSSN** : 0975-153X
Frequency : Quarterly, Published Since : 2009

Google Citations: Since 2009
H-Index = 96
i10-Index: 964

Source: https://scholar.google.co.in/citations?user=S47TtNkAAAAJ&hl=en

Conflict of Interest: Author of a Paper had no conflict neither financially nor academically.

Editorial Excerpt

The article has 7% of plagiarism which is the accepted percentage as per the norms and standards of the journal for publication. As per the editorial board's observations and blind reviewers' remarks the paper had some minor revisions which were communicated on a timely basis to the authors (Bhawna and Mayank), and accordingly, all the corrections had been incorporated as and when directed and required to do so. The comments related to this manuscript are noticeably related to the theme "**A Finite State Automaton is a Tool to Represent Formal Language**" both subject-wise and research-wise. This paper is well-researched and well-written that provides valuable insights into the use of finite state automata in representing formal languages. It is a must-read for anyone interested in this topic or working in related fields. The paper's clarity, depth of knowledge, and relevance to real-world applications make it a valuable contribution to the field of computer science. After comprehensive reviews and the editorial board's remarks, the manuscript has been categorized and decided to publish under the "**viewpoint**" category.

Acknowledgement

The acknowledgment section is an essential part of all academic research papers. It provides appropriate recognition to all contributors for their hard work and effort taken while writing a paper. The data presented and analyzed in this paper by (Bhawna and Mayank) were collected first handily and wherever it has been taken the proper acknowledgment and endorsement depicts. The authors are highly indebted to others who facilitated accomplishing the research. Last but not least, endorse all reviewers and editors of GJEIS in publishing in the present issue.

Disclaimer

All views expressed in this paper are my/our own. Some of the content is taken from open-source websites & some are copyright free for the purpose of disseminating knowledge. Those some we/I had mentioned above in the references section and acknowledged/cited as when and where required. The author/s have cited their joint own work mostly, and tables/data from other referenced sources in this particular paper with the narrative & endorsement have been presented within quotes and reference at the bottom of the article accordingly & appropriately. Finally, some of the contents are taken or overlapped from open-source websites for knowledge purposes. Those some of i/we had mentioned above in the references section. On the other hand, opinions expressed in this paper are those of the author and do not reflect the views of the GJEIS. The authors have made every effort to ensure that the information in this paper is correct, any remaining errors and deficiencies are solely their responsibility.