STEM Techniques in Programs for Recognizing Regular Languages in the Teaching of the "Language Processors" Course

Juliana Dochkova-Todorova
"St. Cyril and St. Methodius"
University of Veliko Tarnovo, Bulgaria
doskova@ts.uni-vt.bg

Maya Hristova
"St. Cyril and St. Methodius"
University of Veliko Tarnovo, Bulgaria
maia.hristova@ts.uni-vt.bg

STEM похвати при програмите за разпознаване на автоматни езици в обучението по дисциплината „Езикови процесори"

Abstract

The paper presents a study on the possibilities of applying STEM approaches in teaching language processors, with a focus on algorithms for recognizing input words. The authors examine different implementation options for recognizing regular languages and analyse the methodical aspects related to them.

Typical mistakes made by students when creating the programs are systematized. Suggestions have been made on error detection methodologies, aimed at improving students' mastery of algorithmic thinking.

The study also draws attention to the integration of artificial intelligence tools in solving programming tasks.

Keywords: *Finite Automata; Teaching Language Translators; Language Recognition; Regular Expressions.*

Introduction

Formal languages and finite automata constitute a central component of theoretical computer science [1]. They provide a mathematically rigorous framework for representing symbols and rules, allowing the precise description of valid sequences of symbols, commonly referred to as words. These models are grounded in set theory and formal logic, which makes it possible to analyze computational problems with a high degree of precision. Through formal grammars and automata, it becomes feasible to characterise entire classes of languages and to determine whether specific problems are solvable or unsolvable within a given computational model [1]. This theoretical foundation is essential for understanding not only the structure of languages themselves but also the underlying principles of algorithmic processes and computation in general. By studying these models, students and researchers develop the ability to reason about abstract systems, prove properties of languages, and establish equivalences between different representations of the same language.

In addition to their theoretical significance, formal languages and automata have a profound impact on practical software development [2]. They form the basis for the design and construction of fundamental tools such as compilers and interpreters, where knowledge of grammars, parsing techniques, and state machines ensures the accurate translation of high-

level programming languages into machine-executable instructions. Beyond this core application, these models are employed in code optimization, error detection, and verification of program correctness. Moreover, they are instrumental in security-related tasks, such as filtering malicious input or identifying dangerous patterns, as well as in text-processing applications including search engines, data parsing, and natural language processing. By bridging the gap between abstract theory and concrete implementation, the study of formal languages and automata equips students with the skills to design efficient algorithms, reason systematically about computational processes, and apply formal methods in diverse technological contexts. This integration of theory and practice exemplifies the essential role of formal models in both education and real-world software engineering [2].

In turn, finite automata are one of the tools with which modern computer and software systems are built [2], [3]. The theory of automata and languages is a key part of the foundations of computer science. Its study contributes to the understanding of the working principles of computers and the methodologies that are used [1].

Formal languages and finite automata are grounded in strong theoretical foundations. They also have a significant practical impact, playing a crucial role in modern software development. The study of these models informs the design and implementation of essential tools in computer science. A primary example is the construction of compilers and interpreters, where knowledge of formal grammars, parsing techniques, and automata theory ensures the accurate translation of high-level programming languages into machine-executable instructions. Moreover, these concepts are directly applied in the optimization of code, the detection and handling of errors, and the enforcement of syntactic and semantic correctness throughout the compilation process.

Regular languages also play a special role in computer science, as they are a well-studied class of formal languages that can be recognised by a finite automaton, either deterministic or nondeterministic. This class of languages corresponds to regular languages described by regular expressions and to languages generated by regular grammars.

The fundamental role of regular languages for many theoretical areas in computer science is determined by their formal rigor, computational efficiency and practical applicability. Compilers, search engines, verification systems and text processing tools are developed using them. Furthermore, regular languages provide a good theoretical basis for the development of more complex model classes of languages such as context-free and context-aware languages.

On the other hand, important in computer science education are the development of algorithmic thinking, knowledge of various mathematical models, and a number of other cognitive and practical skills that relate directly to STEM education components.

Algorithmic thinking can be expressed by the ability to solve problems through clear instructions for a finite number of steps, executable by a human or computer. Part of this thinking is the ability to break down a task into subtasks. Furthermore, defining and implementing algorithms requires skillful recognition of different patterns. Correct estimation of complexity, the amount of time a program takes to run, and the memory required are also essential for developing effective algorithms.

The essence of mathematical modeling is the representation of real or abstract systems by various mathematical structures in order to analyse and predict behavior. In courses such as Language Processors, students model systems and algorithms using automata, regular expressions, and context-free grammars.

STEM approaches play a crucial role in the teaching of the Language Processors discipline by fostering algorithmic thinking, mathematical reasoning, and technical competence. Through tasks such as modelling systems with finite automata, constructing lexical and syntactic parsers, and implementing compiler modules, students develop the

ability to decompose complex problems into manageable components and design systematic procedures for their resolution.

By integrating theoretical concepts with practical applications, STEM-based instruction enables learners to apply formal methods to real-world computing tasks. This connection between abstract models and hands-on implementation not only strengthens students' analytical and problem-solving skills but also prepares them to develop software solutions that are both correct and efficient.

In this context, the integration of STEM-oriented techniques into the learning process provides a number of significant benefits that extend beyond the acquisition of factual knowledge. One of the most important outcomes is the cultivation of abstract and algorithmic thinking, which equips learners with the ability to decompose complex problems into manageable components and to design systematic procedures for their resolution. Such skills are fundamental not only in the study of formal languages and automata theory, but also in broader areas of computer science and engineering.

Furthermore, STEM-based methods contribute to the development of strong mathematical and technical competences, particularly in domains closely connected with the theory and practice of formal languages. For instance, learners gain practical experience in designing and implementing compiler modules, where the application of formal grammars and parsing algorithms becomes an essential exercise in bridging theoretical constructs with real-world implementations. At the same time, exposure to interdisciplinary STEM approaches fosters the capacity to adopt and adapt various engineering strategies for problem solving, encouraging students to apply analytical reasoning, experimentation, and iterative design in diverse computational contexts.

Taken together, these benefits illustrate the pedagogical and practical value of incorporating STEM principles into computer science education, ensuring that learners not only acquire theoretical knowledge, but also develop the cognitive and technical skills necessary to apply it effectively in professional and research settings.

The main objectives of this study are to analyse the methods for solving formal language recognition problems and to systematize the students' errors. For the purposes of this study, observations were carried out over the last two academic years during the delivery of the course "Language Processors" to computer science students at the Faculty of Mathematics and Informatics, St. Cyril and Methodius University of Veliko Tarnovo. The study presents three approaches for solving a basic formal language recognition problem and systematizes methodological notes related to their teaching. An analysis is made of the errors made by students in solving the problem, the reasons for them and the possibilities of avoiding them. Finally, some ideas for the use of AI in teaching by lecturers and students are proposed.

EXPOSITION

1. Theoretical foundations and application

Students studying computer science and software engineering are introduced to the theoretical foundations of computer science during their studies through the courses Automata and Language Theory, Language Processors, Language Translators and Translation Methods. Some of the topics covered relate to finite automata and regular languages in the following sequence:

1. Regular languages (RL) and regular grammars (RG)
2. Properties of regular languages
3. Finite-state automata (FA) – recognisers and transducers
4. Deterministic finite automaton (DFA)

5. Nondeterministic finite automaton (NFA)
6. Determinization of DFAs
7. Equivalence of automata and regular grammars
8. Minimisation of DFAs
9. Algorithms for regular grammars
10. Regular expressions (RE) and regular languages
11. Equivalence of regular expressions and automata
12. Lexical analysis as a stage of compiler and interpreter work

The basic definitions needed for the algorithms of the subject of this study are given in many textbooks on automata theory [4], [5], [6] and online tutorials [7], [8], [9]. Only the main ones will be presented informally here for the purpose of exposition:

- *An alphabet* is a set of symbols called *letters*. *A word* is a sequence of letters of an alphabet. In this sense, strings and computer programs are words. An example of an alphabet is the set {a, b} whose letters are 'a' and 'b'.
- *A formal language* (FL) is a set of words. Most often, an FL includes all possible words that are constructed according to certain rules. The rules are specified differently according to the goals of specific tasks – description of word properties, formal grammars (FG), finite automata, stack automata, regular expressions, etc.
- According to the Chomsky hierarchy, *a regular language* is a formal language defined by a class of formal grammars known as regular grammars. To define such a language all the above ways can be used, for example the language $L = \{a^{(n)}\ b^{(m)} |\ n \geq 1,\ m \geq 0\}$ includes all words starting with any number of letters 'a', having at least one such letter, and ending with any number of letters 'b'.
- *A DFA* is a kind of pattern consisting of different states that are connected together by transitions. One of the states is initial, and some of them are final. For each state and for each possible input letter, there is a transition function defined to transition to another state. The automaton starts from the initial state and processes the letters of the input word as it transitions from state to state. Once all letters of the input word have been read, the last state reached determines whether or not this DFA recognises the input word. Recognition is performed only in the states that are final.
- Automata are represented by graphs called diagrams, where the operation of an automaton corresponds to a path in this graph.
- In NFA, unlike DFA, there can be multiple transitions to different states from each state and for each letter. Thus, for a single input word, more corresponding paths in the graph are obtained. NFA recognises an input word if at least one of these paths reaches a final state of the automaton.
- *A regular expression* (RE) is a character model that also describes a set of words using sequences of alphabet letters and special characters such as *, +, and parentheses, where * indicates repetition, the symbol + selects a letter from several possible ones, and parentheses change the precedence. For example, the RE "$a\ (a+b)^*$" describes all words that are arbitrary sequences of the letters 'a' and 'b' beginning with the letter 'a'. *The regular language* is the set of words that can be described by a given RE.

Multiple statements about these basic concepts make up the theory of formal languages, but also directly reflect the needs of practice. In turn, equivalence claims connect the different ways of specifying regular languages: RG, DFA/NFA, RE, etc. But they are also based on proven algorithms for obtaining an equivalence structure that specifies the same set of words, i.e. equivalence means matching the corresponding regular languages.

A practical application is in regular language recognition tasks where an algorithm is constructed to determine whether a word belongs to a specified formal language. The application extends to various areas of computer science – where text, data and commands are handled. The main areas are:

- Development of compilers and interpreters – In lexical analysis when splitting the input program into keywords, identifiers, numbers, etc.;
- Regular expressions in practice – In searching and replacing text and in validating formats such as email, phone number, URL, ISBN, etc.;
- Security – filtering dangerous strings and recognizing attack patterns;
- Natural language processing – tokenization and morphological analysis.

A major aspect in motivating students to study complex theoretical concepts is precisely their application in practice. Analysing concrete cases from compiler construction, text processing, and search applications provides a clear illustration of the skills and knowledge students are expected to acquire.

Working on formal language recognition problems prepares students to design fast and secure software solutions by giving them the necessary mathematical and algorithmic foundation.

2. Approaches to implementing programs for regular language recognition

We will consider the following basic problem of regular language recognition:

Problem. *To write a program to recognise a given formal language. After starting, the user is expected to enter an input word, and then it is checked whether it is from the language or not. As a result, whether the word is from the language is output. Use only assignment commands, input and output commands, conditional operators and loops*.

The input word is remembered as a string or in a one-dimensional array. The formal language may be specified in various ways, for example:

- with a general appearance of the words and a description of their properties;
- DFA;
- RE;
- NFA;
- formal grammar.

Of algorithmic and practical interest are the first three ways. For the other two ways, the following considerations hold:

- The realization of a NFA operation corresponds to the realization of more than one DFA according to most possible transitions of each step of the automaton operation. Therefore, in this case, a deterministic algorithm is applied at the beginning and thus the problem is reduced to one DFA.
- For languages specified by FG, one initially proceeds to equivalent finite automata via the equivalence algorithms.

In solving these three variants of the regular language recognition problem, we will focus on the three main approaches considered in the Language Processors course at the Faculty of Mathematics and Informatics and offered separately in examples and problems from textbooks and online learning materials such as [5], [10], [11]:

- Approach 1: An algorithm based on the properties of words in the language
- Approach 2: An algorithm implementing the performance of a DFA recognizing the words of the language
- Approach 3. An algorithm based on the RE defining the given language.

Approach 1

The first methodological approach applied to the problem of regular language recognition is centered on the design and implementation of algorithms that exploit intrinsic word-level properties such as length, ordering of symbols, and frequency of occurrence. Within this framework, the structural features of words are systematically analyzed in order to identify patterns that are characteristic of a specific formal language. Such features may include constraints on permissible symbol sequences, distributional regularities, or recurring syntactic structures that collectively form a distinctive linguistic "signature".

This approach is particularly effective in scenarios where explicit grammatical descriptions are either unavailable or computationally expensive to process, since the recognition task can be reduced to the identification of quantifiable word attributes. By focusing on measurable structural indicators, the algorithm is able to approximate language membership decisions with a relatively high degree of efficiency. Consequently, this method has found widespread application in the development of formal language recognition programs, where its simplicity and computational feasibility make it a valuable tool for both theoretical experimentation and practical implementation in software systems.

The analysis of the input word requires a detailed consideration of the formal properties that are characteristic of the language. These include the word length, the character frequency, the ordering of characters and subwords, symmetries and repetitions, and the positions of specific subwords.

The first step of the property-based algorithm involves analysing the description of the language. To illustrate, consider the language $L = \{a\ b^n\ a \mid n \geq 1\}$. Each word in this language has a strict structure and exhibits the following properties:

- a minimum length of three characters, since the word starts with exactly one letter 'a', followed by one or more letters 'b', i.e. $n \geq 1$, and ends with exactly one letter 'a';
- the first and last characters are always 'a';
- the intermediate characters consist solely of 'b';
- each word contains exactly two occurrences of the character 'a' – at the beginning and at the end.

In the second step of the algorithm, the given word is broken down into characters and the number of characters is checked to see if they satisfy the conditions of the language properties. Let us consider the word "abbba". The beginning of the word w[0] = 'a' consists of one character 'a' and this matches the language properties L. The intermediate subword w[1:-1] = 'bbb' contains only characters 'b' whose length is at least 1, which in turn also satisfies the condition. The last subword w[-1] = 'a' again contains one character 'a'. As a conclusion, we can confirm that the word w = 'abbba' belongs to the language under consideration.

In a next step of the algorithmic approach, a further check of the ordering of the symbols is performed, since in many cases it is not enough that the number of symbols corresponds to the properties of the language, but it is also important in which order they occur. For the language L under consideration, each word would have the following form:

$$w = \underbrace{a}_{beginning} \quad \underbrace{bbb\ldots}_{only\ b,one\ times\ minimum} \quad \underbrace{a}_{end}$$

i.e. the letter 'a' is not allowed among the letters 'b'. Accordingly, this step would have to check that the first character of the input word is 'a', that the last character is also 'a', and that the intermediate part contains only the letters 'b' in the correct order – that all characters between the first and second characters are only 'b'.

The algorithmic method of recognizing regular languages through input word properties is closely related to the STEM approach. As such, it develops algorithmic thinking. It is also intuitive enough and this makes it suitable for students who have not sufficiently mastered formal modelling with automata. It consists of basic constructs – counters, conditions and loops and this makes it easy to implement. Moreover, it is flexible enough, which allows working with more complex conditions.

One drawback of this method is its limited applicability beyond regular languages, as the relevant property can be derived more efficiently for regular languages. As the complexity of language classes increases—for example, in the case of context-free and context-sensitive languages—more advanced programming techniques and language constructs become necessary.

Approach 2

The second approach, discussed within the framework of the *Language Processors* course, is grounded in one of the fundamental tools of formal language theory, namely the DFA. This approach emphasizes the systematic construction and analysis of automata as a means of formalizing and solving language recognition problems. A well-established technique associated with this methodology is the so-called "*set and simulate operation*" method, which provides a structured procedure for the creation of an automata-based recogniser.

In this method, the behavior of the automaton is explicitly described by simulating its response at each individual step of processing the input word. More specifically, the set of possible states is examined and updated as the automaton reads the successive symbols of the input sequence, thereby enabling a step-by-step account of the recognition process. This makes it possible to capture not only the final acceptance or rejection of the word, but also the intermediate computational states through which the automaton transitions.

As a classical and widely adopted approach, the *set and simulate operation* method serves as a pedagogical bridge between the theoretical underpinnings of automata theory and their practical application in the construction of recognisers. Its stepwise nature ensures transparency in the analysis of automaton behavior, making it a particularly effective method for teaching, demonstrating, and implementing language recognition algorithms.

This approach involves two main phases. In the first, the automaton is formally defined as $A = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states;
- Σ is the input alphabet;
- δ is the function of transitions, $\delta : Q \times \Sigma \to Q$;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of final states.

In the second phase of the approach, a simulation of the automaton's operation on a given input word is performed. Starting from the initial state, the automaton reads the symbols sequentially, applying the transitions functionδ to each symbol to determine the next state. Finally, it checks whether the reached state belongs to the set$F$. If yes – the word is recognized [1]. This visualization can be implemented by programming code or by means of various online tools (see Fig. 1).

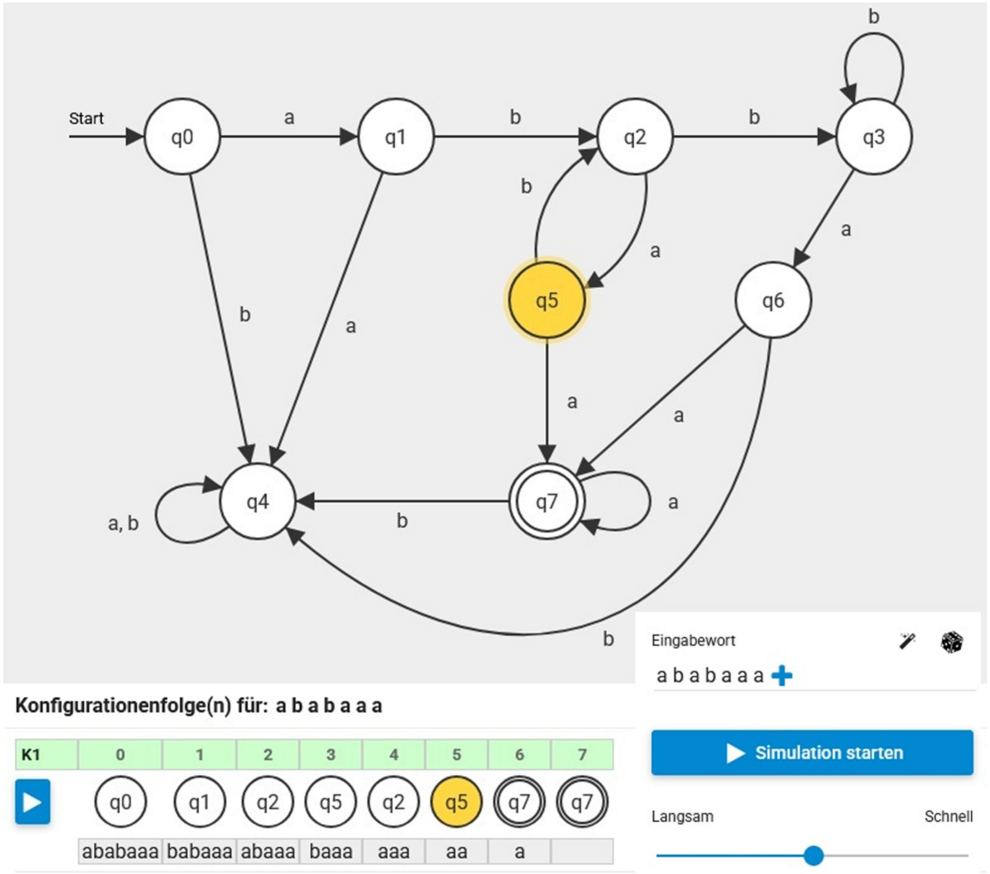

Fig. 1. Online tool [20] for the simulation of DFA performance

The use of the DFA (specifying and simulating its operation) is an effective approach in the teaching of the disciplines "Language Processors" and "Formal Languages and Grammars", since it allows the integration of theoretical knowledge with practical algorithmic thinking. By specifying automata and simulating their operation in the form of a running program, students develop not only an understanding of formal methods, but also practical skills that find application in areas such as lexical analysis, language design, and system verification [12], [13].

Approach 3

The method of recognizing regular languages through regular expressions (RE) finds application in the process of learning the Language Processors course. This method provides an intuitive connection between theoretical models of formal languages and real programming implementations. This property makes it particularly effective in practical problem solving [12].

Regular expressions could serve as a formal means to describe regular languages, a subset of formal languages that can be recognised by deterministic or nondeterministic finite automata. The formal equivalence between regular expressions and finite automata is clearly established in automata theory [1]. Regular languages coincide with languages of RE.

Using this approach, a regular expression is initially formulated that describes the syntax of the words of the language. Consider the example formal language $L = \{ab^n a \,|n \geq 0\}$. It could be represented by the template ab^*a, representing the RE. The next stage involves a programming test using appropriate language constructs and/or language libraries. The last step directly checks whether the given input word matches the regular template. If this condition is satisfied, then the word belongs to the language.

The use of regular expressions enables syntactic pattern analysis, data filtering and automatic validation of input to programs. It is used in tools such as lexical analyzers (lex, flex), compilation systems, text editors and network filters [12].

The advantage of the considered approach lies in its practicality and ease of implementation, especially in an educational environment where students learn the equivalence of formal concepts in real code [13]. Its main drawback is that it is only applicable to regular languages. More complex languages, such as those with nested structures or dependent counts of different subwords, cannot be defined using regular expressions and require the use of stack automata or context-free grammars [1].

Recognition of formal languages, and in particular regular languages, can be implemented using any of the approaches considered - an algorithm based on word properties, using DFA, or using regular expressions and direct testing. Although diverse in form and implementation, for these languages they are equivalent in expressive power.

As stated above, according to automata theory, any regular language can be represented in various equivalent ways, such as by a finite automaton, a regular expression, and an regular grammar. Specifically:

- any regular expression can be transformed into a NFA and subsequently into a DFA;
- any DFA can be algorithmically transformed into a regular expression;
- for every DFA there exists an regular grammar defining the same language;
- for every regular grammar there exists a NFA defining the same language.

Algorithms that check whether a word belongs to a language by evaluating length, ordering, or frequency of characters were specified as Approach 1. These algorithms, when they do not use a stack-type data structure or recursion, are equated in expressive power to DFAs because they are based on a finite number of states and local verification. This makes them only applicable to regular languages.

In turn, regular expressions provide a compact and expressive means of defining syntactic patterns. They are widely used in practice - in language processors, compilers, lexical analyzers and other text analysis systems. REs can be implemented using compiled patterns in programming languages such as Python, Java or C++, allowing direct word testing with minimal effort.

Since RE and DFA are mutually convertible, the use of regular patterns in a software context actually implements the concept of a finite automaton without explicitly building a machine model.

Tab. 1 Equivalence of different approaches

Approach	Expressive power	Transform to DFA	Transform to RE	Constraints
1. Property-based algorithm	Regular	✓	✓	Cannot account for dependencies in properties
2. DFA	Regular	-	✓	Lacks support for comparing the frequency of subwords in the accepted language
3. Regular expression	Regular	✓	-	Cannot express contextuality

For any regular language, the approaches are equivalent in terms of being able to accept the same sets of words. The choice of approach depends on the context, i.e. whether it is a formal proof, a simulation, a program implementation or a lexical analysis.

Table 2 Comparison of the three approaches

Approach	Form of implementation	Suitable for automation	Suitable for proofs	Equivalence with RE
1. Property-based algorithm	Conditional checks on start/end characters and occurrence counts	Partial	No	Yes
2. DFA	States and transitions	Yes	Yes	Yes
3. Regular expression	Template	Yes	Yes	Yes

A generalized comparison between the different approaches shows that learning the Language Processors discipline requires systematic mastery of different methods for recognizing formal languages. There is a theoretical equivalence between regular expressions, finite automata and regular grammars that is well established in many research papers on formal language theory. These approaches have equal expressive power with respect to regular languages.

The correspondence between right-linear grammars and finite automata has also been proved, and this fact is discussed in detail by [6]. In his work, he demonstrated formal transformation rules between different representations. In turn, [14] analyzed the descriptive complexity of going from automata to regular expressions and vice versa, stressing that although transformation is possible, it is not always efficient.

In modern teaching, regular expressions are often used in beginning topics and exercises because of their syntactic compactness and intuitiveness. A practice-oriented survey for generating regular languages using regular expressions is presented by [15], with the authors pointing out their advantages for fast recognition in real-world systems. In [16], a method for synthesizing regular expressions from examples is proposed, which has been successfully applied to training tasks in regular languages. This approach creates opportunities for semi-automated learning and adaptive knowledge checking.

The structure of finite automata can be considered as a mechanism for formal recognition of regular languages [17], including minimization and simulation strategies. This approach allows precise control over recognition states and transitions and is particularly suitable for simulation learning and building language compilers.

While algorithms based on word properties (such as length, ordering, character frequency), such as Approach 1, are not formally defined in automata theory, they have significant educational value. They are widely used in the initial stages of learning to form algorithmic reasoning. In the context of academic disciplines such as Language Processors, this approach supports the transition between theory and practice by providing concrete and implementable steps.

In [18], we demonstrate the application of regular expressions combined with automata in decoding output from neural networks for handwriting recognition. This demonstrates a concrete software application of the considered approaches and reinforces their relevance not only in education but also in practice.

The considered approaches for regular language recognition are theoretically equivalent but have different advantages depending on the application context. DFAs offer formal rigor and transparency, regular expressions offer expressive compactness, and property-based algorithms offer an intuitive transition to practically applicable algorithms. In this respect, their use in the Language Processors course helps to build a solid foundation.

3. Methodological notes on approaches to problem solving

Consider the regular language L = $\{(ab)^n b^m a^k | n \geq 1, m \geq 0, k \geq 2\}$ to the basic problem described above. Regardless of the approach chosen to solve the problem, several key stages are passed during learning:

1. Analyzing the problem condition using example words from within and outside the language;
2. Compose an algorithm in accordance with the previously considered approaches;
3. Program in a pseudocode or programming language;
4. Creating test words for verification and possibly going through several tests and program adjustments.

During the first stage, both the language word of minimum length abaa and other words of similar length are determined: abbaa, abaaa, ababaa, abbbaa, etc. These words could be used for the test in stage 4. Helping to understand the condition and create the algorithm is also the identification of some words that are not in the language. Particularly useful at this stage are online tools such as [19] for simulating the operation of a finite automaton, see Fig. 1. This makes it faster and easier to track the automaton's performance for a given input word by illustrating each transition with an animation. Visualization and interactivity support better understanding and even increase motivation when learning concepts.

For Stage 3 of the problem solving, the following implementations of the three approaches are considered during the Language Processors exercises, reflecting the algorithm chosen in the second stage.

3.1. Approach 1 – property checker program:

In regular language recognition tasks, even when the languages are specified by regular expressions or finite automata, this subapproach is frequently employed, relying on a structural analysis of the words in the language. To illustrate the approach under consideration, we will use the language L mentioned above.

As mentioned, the method used requires to perform a correct analysis of the structure of the words that belong to L. Such words have:

- One or more sequences consisting of the subword ab , as for $(ab)^n, n \geq 1$;
- It is possible to follow any number of charactersb due to the requiremen $m \geq 0$;
- The word ends with at least two letters a and this is defined by the condition $k \geq 2$.

This structure allows the use of a deterministic algorithm that does not require simulating an automaton or building a regular expression, but is based on checking the ordering and number of characters. Such an algorithm could be described by the following pseudocode:

```
Function is_in_language(w):
    index← 0
    n← 0
    while there is an ab at the current position:
        index← index + 2
        n← n + 1
    if n < 1→ reject
    as long as there are 'b' characters:
        index← index + 1
    as long as there are 'a' characters:
```

```
        count_a← count_a + 1
        index← index + 1
    if count_a ≥ 2 and index = length→ accept
    otherwise→ reject
```

The presented algorithm considers a linearly given input word and checks its properties without explicitly needing to simulate the automaton's operation. This contributes to make the approach used suitable for educational purposes and illustrates the efficient recognition of regular languages by generalizing their characteristic properties.

The method is applicable to relatively simple regular languages where blocks of letters and words with certain repeatable features are distinguished. For more complex constructions, this approach becomes impractical or inapplicable, which highlights the role of formal models.

3.2. Approach 2 - Create a NFA/ DFA automaton:

As mentioned, regular language recognition is a key aspect of the theory of formal languages and automata. In the course of the Language Processors course, problems are also solved using Approach 2, which is based on constructing a NFA, converting it to a DFA, and simulating its operation through a program and using the facilities of various online resources.

Let us consider the language L, also used in the description of Approach 1 in Section 3.1. According to the equivalences discussed above, any such language can be specified by a finite automaton, a regular expression, or a formal grammar of type 3 in Chomsky's classification.

The NFA that could be used in this case is given in Fig. 2 and has the following states:

- $q0$ – Home, expecting the first 'a' from 'ab';
- $q1$ – after 'a', expects 'b';
- $q2$ – end of the first or consecutive block 'ab', possibly moving to the next letter 'a' (for a new block 'ab') or to the next subword – for b^m this is state q3, and for a^m this is state q4;
- $q3$ – for the subword b^m, where $m > 0$, it can stay in this state or go to q4;
- $q4$ – this is reached after the first letter 'a' of the subword a^k;
- $q5$ – second and subsequent letters 'a' are added and this is the final state.

This automaton has the following transitions:

- $q0 \xrightarrow{a} q1$
- $q1 \xrightarrow{b} q2$
- $q2 \xrightarrow{a} q1$ (repeat ab)
- $q2 \xrightarrow{b} q3$
- $q2 \xrightarrow{a} q4$ (if $m = 0$)
- $q3 \xrightarrow{b} q3$
- $q3 \xrightarrow{a} q4$
- $q4 \xrightarrow{a} q5$
- $q5 \xrightarrow{a} q5$.

The nondeterminism of the automaton here is determined by the state $q2$ and the input letter 'a'. This can mean a new beginning (transition to the state $q1$) or the beginning of the last subword consisting of the letters 'a' (transition to $q4$).

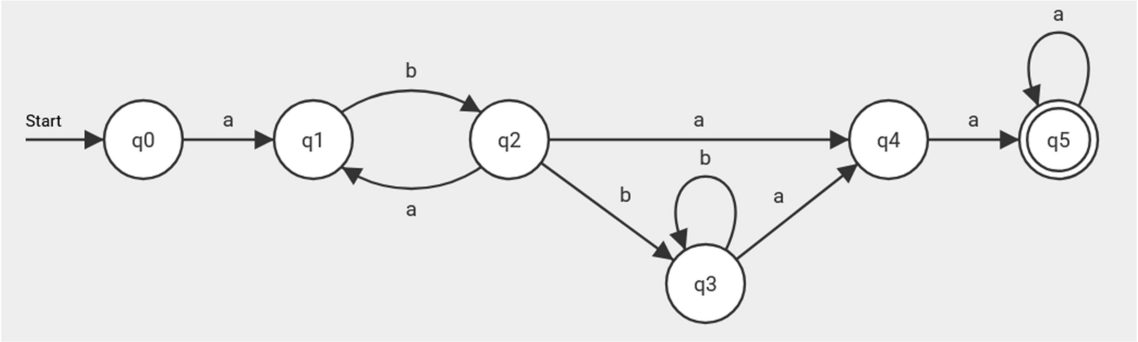

Fig. 2 NFA for the language under consideration

Applying the NFA to DFA conversion algorithm for this automaton yields the equivalent DFA given in Fig. 3.

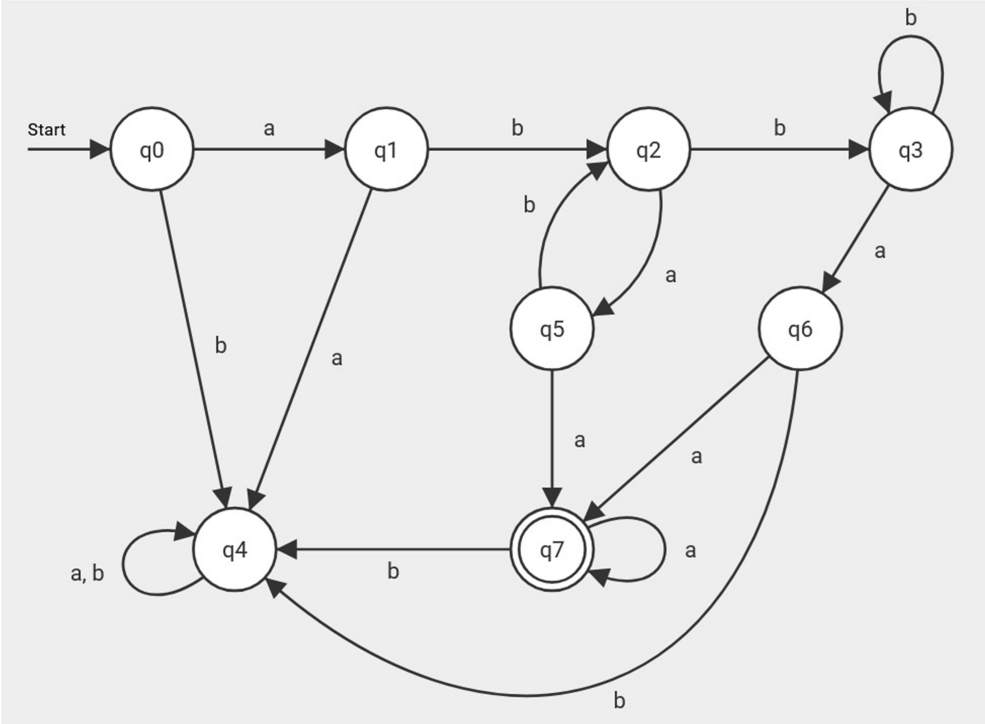

Fig. 3. DFA for the language under consideration

The DFA of Fig. 3 has the following table of transitions given in Tab. 3.

Tab. 3. Table of transitions for DFA

State	Input 'a'	Input 'b'
q0	q1	q4
q1	q4	q2
q2	q5	q3
q3	q6	q3
q4	q4	q4
q5	q7	q2
q6	q7	q4
q7	q7	q4

The initial state of the deployed DFA is $q0$. The symbol 'a' from $q0$ leads to $q1$, and 'b' directly to $q4$, which is a prerequisite for a loop. The loop from $q1 \rightarrow q2 \rightarrow q3$ handles the alternations of the letters 'a' and 'b'. In turn, the state $q4$ leads to a loop – any input leads back to state $q4$. The final part of the input word characters is controlled by the states $q5$, $q6$ and $q7$, and on reaching $q7$ the automaton can read more characters 'a' or read 'b' and return to the state $q4$.

This automaton could be implemented using the following pseudocode:

```
function recognise(word):
    state = q0
    for symbol in word:
        if symbol in transitions[state]:
            state = transitions[state][symbol]
        else:
            return False
    return state in final_states
```

This approach demonstrates the equivalence between the theoretical formulation of regular languages and their practical implementation via finite automata. The diagram and table of transitions aid in visualizing and understanding the logic, and the pseudocode shows that automata can be applied directly to validate words of the language.

For demonstration in the learning process it is often necessary to use the capabilities of various online tools. These include JFLAP, which supports manual construction and an automatic conversion function [11], as well as various NFA → DFA web applications that interactively convert NFA to DFA and display tables and diagrams [20], [21]. Another possibility is provided by the web/desktop simulators AutomataVerse / Automaton Simulator / FSA-Animate, which allow to define a NFA, perform a conversion, step-by-step simulation and create diagrams [22], [23], [24].

3.3. Approach 3 – RE

Another approach applied in the learning process is based on regular expressions. For the considered language L, the structure of words is described by three sequential blocks:

- one or more repetitions of the subword 'ab';
- any number of letters 'b';
- at least two letters 'a'.

This leads to a RE of the form $(ab)^+b^*a^2a^*$, and in a full matching formalisation its form could be $\hat{}(?:ab)+b*a\{2,\}\$$. Equivalent expressions are $\hat{}ab(?:ab)*b*a\{2,\}\$$ and $\hat{}(?:ab)\{1,\}b*a\{2,\}\$$. The different variants are interchangeable, since they use equivalent operations to describe at least one iteration of 'ab' (plus +, a quantifier {1,}, or a "shift" of the first 'ab' before the star).

Another RE for this language is ab (ab)* b* aa a*. Again, online tools could be used to check for equivalence of REs and to check for equivalence of REs and finite automata [19]. REs can be visualized with syntactic diagrams as the one shown in Fig. 4.

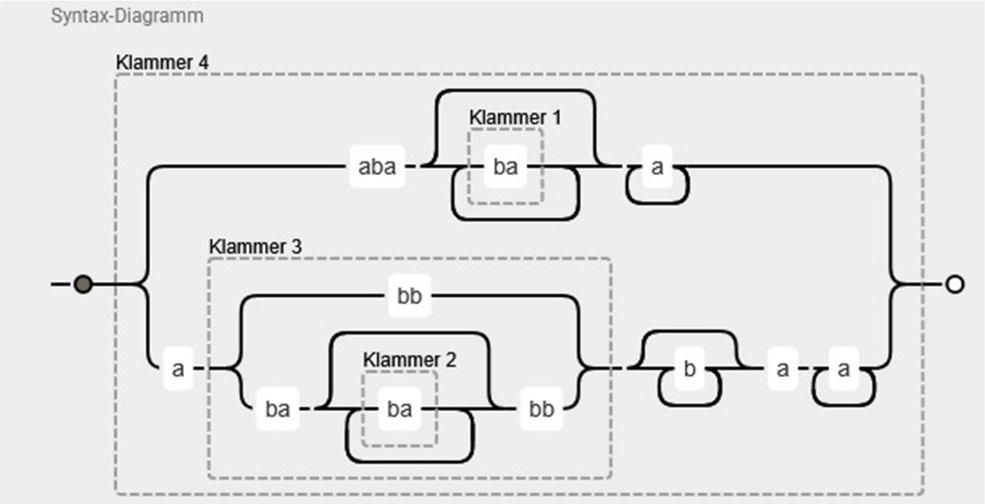

Fig. 4. Syntactic diagram of the language under consideration

RE accurately describe regular languages and are equivalent to finite automata (NFA/DFA). Each RE can be efficiently transformed to a NFA, and then to a DFA. Also, each DFA defines a RE using standard state exclusion algorithms. Therefore, the "RE", "NFA/DFA", and "algorithm by property" approaches are equivalent in expressive power for regular languages; they differ in specification convenience, analysis capabilities, minimization, and implementation resource consumption.

In practical environments (Python, Java, .NET, JavaScript), it is important to use fullmatch via^...$ or the corresponding method (fullmatch, matches()), since a subword is often sought. The RE used does not contain any non-regular mechanisms, so it can be implemented with DFA without exponential backtracking action. The results of unit tests and benchmarks confirm that in a practical implementation the recognition is efficient.

Table 4. Comparison between RE and DFA

Criterion	Regular expression (RE)	Finite Deterministic Automaton (DFA)
Specification	Short and readable: ^(?:ab)+b*a{2,}$	More detailed in terms of states and transitions
Implementation	One line in many languages (regex engine)	Concise description of machine operation, easy generation of validators
Performance	Depends on the flavor (often backtracking); here it is regular and fast	Guaranteed linear in input length (DFA)
Analysis/demonstrability	Harder formal proof with RE alone	Easy invariants, proof by states
Modification	Small changes that are often easy	Adds and minimizes states; harder to implement by human
Code generation	Built into languages	Requires code and a table for transitions

In the final stage of problem solving, students are directed to test and validate the program they have created. This is particularly relevant for their further independent work on similar problems. The program tests, called Property-based test, should include input words in two groups - words belonging to the language and words outside the language. The choice of these words is based on the first stage of problem solving, when the properties of words in the language are considered in detail. It is recommended to choose values of the variables used in

the description that are on the boundaries of the intervals and in combination for the different variables.

4. Methodological notes on automatic tests and student errors

During the course of the Language Processors Discipline, students work independently on problems similar to the one considered here and solve such problems as part of the written exam. The following is a summary of the most common mistakes they make with the different approaches, an analysis of the causes of these mistakes, and a systematization of recommendations for avoiding them.

Table 5 presents the errors in the independent solutions of the considered problem, made most often by students in the training. To each of them are added which is the corresponding approach, examples of words that are misrecognised, and the reasons for the error.

Table 5. Classification of main errors in students' solutions

Approach	Error	Examples	Reason
1, 2, 3	Incorrect conceptual interpretation of word terms in language	The word *abba* is recognised	Incomplete analysis in the language terms in Stage 1 and gaps in the test words in Stage 4
1, 3	Incorrect division of the input word into a sequence of obligatory and optional subwords	The word *abaa* is not recognised	Incomplete analysis in language terms in Stage 1 and gaps in test words in Stage 4
1, 2, 3	Unprocessed boundary cases and short words	The word *ab* and the blank word are recognised	Not all boundary cases were analysed during condition analysis and testing
1	Generating output before full validation or generating more output messages	More messages for not recognizing the word *abaaba*	Error in program structure
1	Recognition of words with letters not allowed for the language	Word recognition *ababaaaz*	No alphabet check
2	The constructed automaton does not recognise the language of the condition	Words outside the language are recognised, words from the language are not recognised	Incomplete encoding of condition words from language or ignorance of DFA operation
3	Formulation of RE that is not equivalent to the given language	Words outside the language are recognised, words from the language are not recognised	Lack of knowledge of and insufficient experience with RE

Most of the above errors are due to gaps in understanding of the theoretical propositions and the high level of abstraction in them. As a consequence of time pressure and the preference for simplified solution approaches, the initial stage of problem solving is frequently insufficiently examined, leading to incomplete analyses of word structures and, consequently, to errors that are difficult to detect.

A diagram illustrating the types of errors made by students in solving regular language recognition problems is given in Fig. 5. The data were obtained during the last two academic years for the Computer Science major in the Faculty of Mathematics and Computer Science and refer only to Approach 1. The main results are:

- Analyzed solutions to a language recognition problem using Approach 1: total number 38
- Completely correctly solved problems: 9

- Main errors in the remaining solutions:
  - Misinterpretation of the condition: 19
  - Incorrect subword splitting: 11
  - Boundary cases: 16
  - Wrong positioning of the output in the structure: 3
  - Inadmissible letters: 2
- In some decisions there are 2 or even 3 errors

Following are some recommendations to avoid the mentioned errors of students:

- Thorough analysis of the terms of the words in the language and giving more examples for words in the language and for words outside the language;
- Using more and varied words to test the programme in Stage 4;
- Accurately identifying the beginning and end of subwords;
- Analyzing the possible lengths of input words and subwords;
- Adhering to the basic structure of the algorithm: input, processing, output;
- Visualization of the equivalent DFA and tracking its performance with example words;
- Creation of RE to be consistent and separate for each subword.

It follows from the results of this analysis that test words are an important basis for solving the problems to detect errors in the code. On the one hand, they make it easier for the instructor during the exercises because it makes it easier to detect the students' errors and guide them to correct them. On the other hand, with their help, students could search for and correct their own errors. Another good practice is setting an independent work task requiring a reasoned selection of such words.

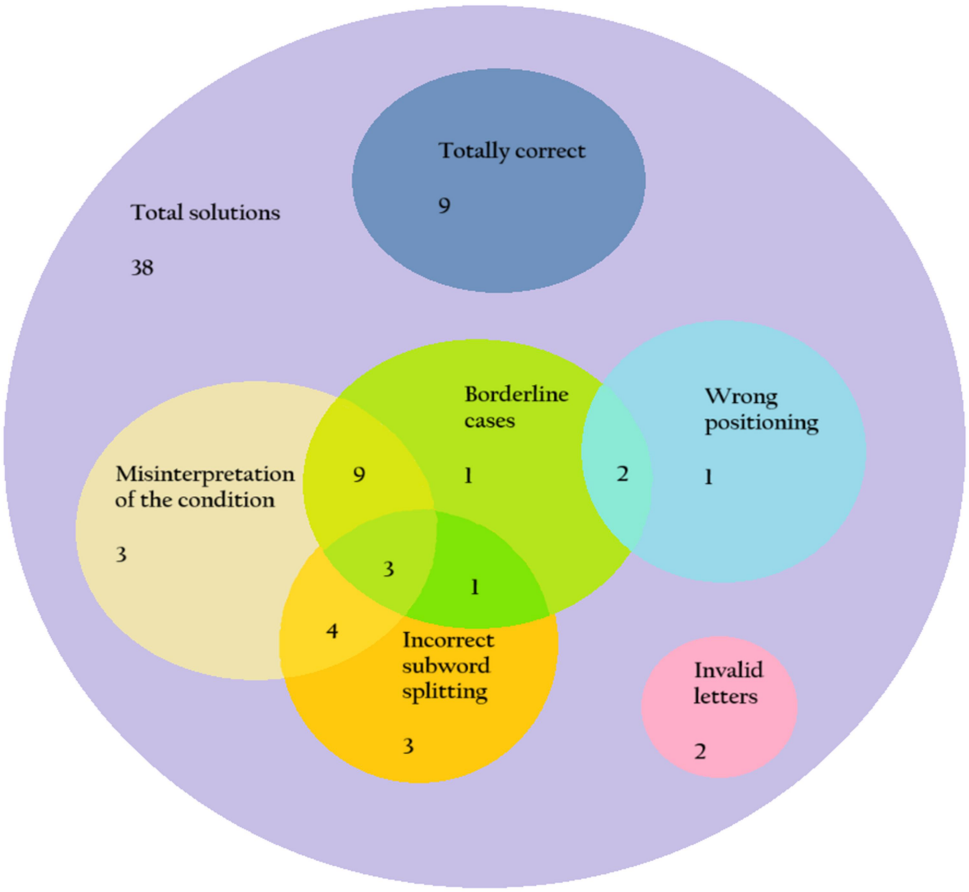

Fig. 5. Analysis of errors in solutions of problems with Approach 1

The role of students' independent work on the tasks set by the lecturer is important for the successful learning of the algorithms and approaches considered. These should be graded in complexity and cover languages with different possible errors, similar to those listed in Table 5. For example, the following formal languages are such for the task considered in this study:

- $L_1=\{(ab)^n (ba)^m \mid n\geq 1, m\geq 0\}$
- $L_2=\{(abc)^n c^m \mid n\geq 1, m\geq 0\}$
- $L_3=\{b^n (ba)^m a^k \mid n\geq 0, m\geq 1, k\geq 0\}$
- $L_4=\{(aba)^n (bab)^m \mid n\geq 0, m\geq 1\}$
- $L_5=\{b^{2n} (ba)^m a^k \mid n\geq 0, m\geq 0, k\geq 1\}$.

Another good practice used in many universities [25], [26] is to assign course projects that are even closer to the practical application of automata. In these, students have to develop their own interpreters for certain small languages, using their knowledge of formal languages and their recognition programs.

5. Use of AI in learning

Artificial Intelligence is making inroads into many areas of modern computing and learning, even in areas such as mathematical proofs and security, with successful results almost everywhere.

Now and in the near future, when many routine activities are performed by AI, the need for programmers who can spot errors in algorithms and programs, test and ensure the required quality of code is expected to increase. These are mostly skills that AI cannot automate [27]. The "sense of rightness" of an AI-generated program can be misleading, and therefore developing critical thinking is another important competency of future programmers.

For now, many universities allow the use of AI in training and even recommend it as an assistant in development in order to help students build the skills needed for their profession. In many cases, however, requirements are placed on how to use and mandatory tagging of the aid, and students are also introduced to important general recommendations and best practices such as correctness checking and context description rules.

In the specific topic of regular languages, tutors could use AI to generate test words, to create help text for tasks and code, to detect errors in student work, and to create a system of additional tasks for practice. In doing so, they should also take a critical approach and personally check the AI-generated text and code.

In terms of practice, AI is used to varying degrees [27] by programmers, but they all understand the limitations of AI and largely direct their work into checking that the generated code meets the objectives, In doing so, developers need a good knowledge of programming, algorithms and data structures. Only then can they conduct successful human verification or create automated tests. In this sense, the basic problem considered here can be modified by adding an AI-generated solution that needs to be verified and corrected by students.

Given the task of writing a C++ program that, for an input word, checks whether it is from the language $L = \{(ab)^n b^m a^k \mid n > 0, m >= 0, k > 1\}$ or not, the AI (ChatGPT, OpenAI) handles it successfully. The program code that is generated meets the requirements. In addition, the AI also generates an explanation of the very approach used to solve the problem in question. Also after the computer program, an explanation of the code and examples of words that belong to the language and those that do not belong are generated.

The AI was also given the task of writing a C++ program that, when given an input word, checks whether it belongs to the same language L using the DFA and a program implementing its operation. This task was again solved correctly.

Another problem the AI had to solve was to write a C++ program that again checks whether an input word belongs to L, but using the third approach presented by RE. The generated solution included a brief analysis of the language and program code that solved the given problem. The explanation of this code is correct and accompanied by brief explanatory notes on how it could be executed.

In the process of working with the AI, the task was also set to write a program (without specifying the programming language used) to recognise a formal language given a generic type of words with the requirement: after starting, input of an input word is expected, then a check is made whether it is from the language and this result is output, only assignment commands, input and output commands, conditional operators and loops are used. In this case, the generated code, Python's default, was unoptimized. The explanation of the code and the preliminary analysis of the given problem contained several inaccuracies, with only a few correct elements. Some of the words that the AI thought belonged to the language under consideration were misidentified as such. Because of these weaknesses of the AI-generated solutions, it is imperative that students are well versed in the theoretical material being studied and check the solutions themselves – their own, others' and AI-generated - carefully enough.

Conclusion

The application of STEM approaches to programming of regular language recognition systems in the study of the discipline "Language Processors" proves to be highly effective in forming a deep understanding of theory and practice. The use of methods based on science, technology and mathematics helps students not only to learn formal models of automata, but also to develop skills in detecting and correcting their own and others' errors. This builds a more sustainable link between theoretical concepts and their practical application in real software solutions.

The incorporation of advanced online tools and techniques from the field of artificial intelligence provides additional opportunities for automation of checks, optimization of algorithms and interactive learning. This allows students to experiment and observe the results of their programs in real time, which stimulates critical thinking and creative problem solving.

In conclusion, STEM approaches are proving to be effective in Language Processor education by encouraging analytical thinking, practical application of knowledge, and independent error detection. Future research may focus on integrating increasingly advanced AI technologies and simulation environments to extend the possibilities for individualized and interactive learning, as well as on developing new methodologies for assessing learned competencies in the context of STEM approaches.

Acknowledgements

This work was partially supported by St. Cyril and St. Methodius University of Veliko Tarnovo, Bulgaria under Project No. FSD-31-328-18/23.04.2025, 2025, "Methodology for Modeling and Developing Information Systems with Artificial Intelligence in the Education Sector".

References

1. Sipser, M. (2012). Introduction to the Theory of Computation (3rd ed.). Cengage Learning.
2. Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Addison-Wesley.
3. Kozen, D. C. (1997). Automata and Computability. Springer.

4. Esparza, J., & Blondin, M. (2023). Automata Theory - An Algorithmic Approach. The MIT Press.
5. Linz, P. (2012). An Introduction to Formal Languages and Automata. Jones & Bartlett Learning.
6. Pettorossi, A. (2022). Automata Theory and Formal Languages - Fundamental Notions, Theorems, and Techniques. Springer.
7. Automata Tutorial - GeeksforGeeks. Available at: https://www.geeksforgeeks.org/theory-of-computation/theory-of-computation-automata-tutorials/ (last view: 18-08-2025)
8. Automata Theory Tutorial. Available at: https://www.tutorialspoint.com/automata_theory/index.htm (last view: 18-08-2025)
9. Theory of Automata Tutorial for Beginners (Safdar Dogar, YouTube playlist). Available at: https://www.youtube.com/playlist?list=PLduM7bkxBdOckkPOjexEV8KKCjqYh1T_3 (last view: 18-08-2025)
10. Murlak, F., Niwinski, D., & Rytter, W. (2023). 200 Problems on Languages, Automata, and Computation. Cambridge University Press & Assessment. DOI: https://doi.org/10.1017/9781009072632
11. JFLAP. Available at: https://www.jflap.org/ (last view: 18-08-2025)
12. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.
13. Lewis, H. R., & Papadimitriou, C. H. (1997). Elements of the Theory of Computation (2nd ed.). Prentice Hall.
14. Gruber, H., & Holzer, M. (2014). "From Finite Automata to Regular Expressions and Back." Int. J. Found. comput. sci.
15. Chaudhari, R. et al. (2022). "A Review Paper on Generating Regular Language Using Regular Expression." EasyChair Preprint.
16. Lee, M., So, S., & Oh, H. (2016). "Synthesizing Regular Expressions from Examples." GPCE.
17. Reghizzi, S. C. (2009). Formal Languages and Compilation. Springer.
18. Strauß, T. et al. (2015). "Regular Expressions for Decoding Neural Network Outputs." arXiv:1509.04438.
19. FLACI - Formal Languages and Compilers and Interpreters. Available at: https://flaci.com/home/# (last view: 18-08-2025)
20. NFA to DFA Converter. Available at: https://nfa-to-dfa-converter.vercel.app/?utm_source=chatgpt.com (last view: 18-08-2025)
21. NFA to DFA (Joey Lemon). Available at: https://joeylemon.github.io/nfa-to-dfa/?utm_source=chatgpt.com (last view: 18-08-2025)
22. Automataverse. Available at: https://www.automataverse.com/?utm_source=chatgpt.com (last view: 18-08-2025)
23. Automaton Simulator. Available at: https://automatonsimulator.com/?utm_source=chatgpt.com (last view: 18-08-2025)
24. FSA-Animate (Alex Klibisz). Available at: https://alexklibisz.github.io/FSA-Animate/?utm_source=chatgpt.com (last view: 18-08-2025)
25. Pädagogisches Landesinstitut Rheinland-Pfalz. Available at: https://inf-schule.de/ (last view: 18-08-2025)
26. Uni Freiburg - Einführung in die Programmierung. Available at: https://proglang.informatik.uni-freiburg.de/teaching/info1/2023/ (last view: 18-08-2025)
27. Dohmke, T. (Blog). Available at: https://ashtom.github.io/developers-reinvented (last view: 18-08-2025)

Received: 26-08-2025 Accepted: 15-12-2025 Published: 29-12-2025

Cite as:
Dochkova-Todorova, J., Hristova, M. (2025). "STEM Techniques in Programs for Recognizing Regular Languages in the Teaching of the "Language Processors" Course", Science Series "Innovative STEM Education", volume 07, ISSN: 2683-1333, pp. 41-60, 2025. DOI: https://doi.org/10.55630/STEM.2025.0705