

https://dspace.org/

Automatic Feedback Generation for the Learning of Regular Expressions

Olaperi Okuboyejo; Sigrid Ewert; Ian Sanders

https://hdl.handle.net/10539/46658
WOS

Downloaded from a DSpace repository. https://dspace.org/



Automatic Feedback Generation for the Learning of Regular Expressions

OLAPERI OKUBOYEJO, SIGRID EWERT, and IAN SANDERS, School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa

Regular expressions (REs) are often taught to undergraduate computer science majors in the Formal Languages and Automata (FLA) course; they are widely used to implement different software functionalities such as search mechanisms and data validation in diverse fields. Despite their importance, the difficulty of REs has been asserted many times in the literature. Due to their abstract and theoretical nature, students sometimes find REs boring and hard to learn. In addition, a review of existing tools that assist students when learning REs shows that the feedback provided is limited. Therefore, this research set out to automatically generate feedback that contains the type and location of the error and hints on how to fix the error for students learning REs. Principles from compiler construction and the theory of computation were used to develop algorithms for error detection. The algorithms detected the presence, type, and location of errors in students' RE solutions. The error details identified by the algorithms were then formatted as feedback usable to students. The performance of the algorithms was evaluated using a test dataset consisting of 249 incorrect REs, and the accuracy of predicting the error positions in the incorrect REs was 82%. In comparison with other tools that generate feedback for students learning REs, the generated feedback in this research is more robust because it contains a summary of the type of error present and the location of the errors in both the incorrectly represented strings (counterexamples) and REs. The developed prototype can be enhanced into a full-fledged academic tool to support teachers and students learning REs in live classrooms, be it a physical or virtual class.

CCS Concepts: • **Theory of computation → Regular languages**; • **Applied computing → Education**;

Additional Key Words and Phrases: Automatic error detection, feedback, regular expressions

ACM Reference format:
Olaperi Okuboyejo, Sigrid Ewert, and Ian Sanders. 2025. Automatic Feedback Generation for the Learning of Regular Expressions. *ACM Trans. Comput. Educ.* 25, 3, Article 36 (August 2025), 26 pages.
https://doi.org/10.1145/3743682

1 Introduction

Feedback is a vital ingredient required by students during learning [19]. It helps to reinforce concepts being taught while exposing and correcting misconceptions [3, 19]. Although practice

This research was supported by the L'Oréal-UNESCO For Women in Science Sub-Saharan Africa Programme and the Postgraduate Merit Award (PMA) of the University of the Witwatersrand, Johannesburg, South Africa.
This research has been presented as part of a thesis report at the University of the Witwatersand, Johannesburg, South Africa.
Authors' Contact Information: Olaperi Okuboyejo (corresponding author), School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa; e-mail: olaperi.okuboyejo@wits.ac.za; Sigrid Ewert, School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa; e-mail: sigrid.ewert@gmail.com; Ian Sanders, School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa; e-mail: dr.id.sanders@gmail.com.



This work is licensed under Creative Commons Attribution International 4.0.

© 2025 Copyright held by the owner/author(s).
ACM 1946-6226/2025/8-ART36
https://doi.org/10.1145/3743682

exercises and other assessments are useful to facilitate learning, it is the feedback given during or after the assessment that helps to consolidate the learning [7, 19]. This research is geared toward generating feedback for students learning **Regular Expressions (REs)**.

REs underlie the search functionality of different software applications including, search engines, the Unix grep utility, and word processors [45]. They are widely used in mainstream programming languages today. REs are often taught as a topic to undergraduate computer science students in a course generally known as **Formal Languages and Automata (FLA)** Theory or Theory of Computation. There are different synonyms for the FLA course. In the 2013 curriculum guidelines for undergraduate degree programs in computer science developed by the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers Computer Society, REs are specified as a topic to be taught in a core course called "Basic Automata, Computability and Complexity" and also in other elective courses [22]. They are also present in the 2023 curricula where they are specified as a Core Tier 1 Topic in the Algorithms knowledge area [46]. A survey of the undergraduate computer science curricula taught at 15 universities across the globe shows that REs are taught to students in different parts of the globe [36]. However, students find them difficult to learn [8, 28] and often get frustrated since they are unable to evaluate their answers. Personalized feedback provided by human tutors is usually a way out. However, personalized feedback is very difficult in today's large classes and virtual classrooms where students learn independently, with some instructors being unable to keep up with the workload [15, 34]. A solution to this problem is the automatic detection of errors in RE exercises and feedback generation, which is the aim of this research.

This research focuses on the syntax of REs taught under the FLA course (which will be referred to as FLA-RE). FLA-RE differs from the representations of REs in programming languages. It uses only the basic operations of the union, concatenation, and Kleene star (including the Kleene plus and exponentiation in some cases). Also, the symbols, i.e., the set of alphabet letters used, are usually few. FLA-RE neither contains complex constructs such as character classes, numerical quantifiers, negative lookaheads, and shorthand symbols nor allows the representation of nonregular languages [21, 33]. Notwithstanding, FLA-RE offers the foundational blocks for writing and understanding REs in general, and is usually the first point of contact with REs for most computer science majors.

There are different learning areas of REs as taught in the FLA course, as presented in different FLA textbooks and as researched upon. These learning areas include *RE Construction*, *RE-Finite Automata Conversion, RE-String Matching*, and *String Generation from REs*. In *RE Construction* exercises, students learn how to write REs that match a set of strings or that represent a given language [5, 8, 12]. *RE-Finite Automata Conversions* focus on converting REs to different types of finite automata [4, 11, 23, 35, 39]. Java Formal Languages and Automata Package is a very popular tool in this area. It is an interactive visualization and teaching tool for formal languages and it guides students through the process of RE and Finite Automata Conversions. In an *RE-String Matching* task, the student is given both an RE and a string and is asked to determine if the RE can produce the string [5]. Finally, the *String Generation from REs* task requires students to generate strings that can (or cannot) be represented by a given RE [5, 11, 42]. The focus of this work is on generating feedback for RE construction exercises.

This research developed algorithms for the automatic detection of errors in an RE and thus the generation of feedback containing the *location of error (mistake), hints on the type of error,* and *bug-related hints for error correction* that can guide students to fix their errors. Alongside the adaptation of existing concepts in the theory of computation, the use of counterexamples to identify the error location in an incorrect RE was introduced. Also, an algorithm to identify the error location in counterexamples was developed.

An evaluation of the predicted error positions in incorrect REs by the developed algorithms indicated a prediction accuracy of 82%. This tool when deployed for use by students would enable independent practice and learning of REs.

The remainder of the article is organized as follows: Section 2 discusses the background and related work while Section 3 presents the error classification for REs. Section 4 describes the process and algorithms developed for detecting different types of errors, while Section 5 details the evaluation process and results. A discussion of the results is presented in Section 6, while Section 7 presents the conclusion and future work.

2 Background and Related Work

2.1 Intelligent Tutoring Systems (ITSs)

ITSs are one of the applications of AI in education [49]. ITSs date back to the 1970s [27]. They are systems designed to recommend learning paths and content, give personalized feedback and support, generate exercises and assessments, engage in adaptive conversation, and provide one-on-one tutoring to students [49]. ITSs can be very powerful to support teaching and learning across several educational levels: kindergarten, primary school, secondary school, and higher education. ITSs come in as a solution to some of the challenges existing in education—one of which is the giving of prompt and personalized feedback to students via automatic feedback systems. In the context of student learning, automatic feedback can be referred to as the usage of computer technology in place of a human tutor in order to give timely feedback to learners.

ITSs and automatic feedback systems have been used across several domains to support students. In the medical domain, they have been used to teach surgical procedures via animations [9]. In engineering [10], they are used to teach students how to operate industrial and electrical appliances. Students learning Physics have also used ITSs to practice problem-solving [17]. Research has been conducted to generate automatic feedback for students learning different computing courses. Programming courses in particular have received great attention. A review reported in [25, 26] documented 102 papers describing 69 different tools developed to give automatic feedback to students learning diverse programming languages. Beyond being used to teach programming, ITSs have been used to teach algorithms and basic data structures [18], as well as to assist in the teaching-learning process of natural deduction in propositional logic (discrete structures) [14]. They have also been used to recommend content and give feedback on the best learning path for a student in the domain of computer networks [48].

The theory of computation is not neglected. For example, systems exist to grade **Deterministic Finite Automata (DFAs)** constructions [2, 12, 13] and the other aspects of FLA theory courses [8, 24, 30, 40–43, 47].

2.2 Feedback in REs

Different feedback types can be given to students during learning. According to the content-based classification of feedback by Narciss [31], the major types of feedback include *knowledge of performance*, *knowledge of result*, *knowledge of the correct result*, and *elaborated feedback*. *Knowledge of performance* feedback is presented in the form of a grade or a score to the learner; *knowledge of result* informs learners of whether their solution to an exercise is correct or not, while *knowledge of the correct result* presents the correct solution to the learner.

Elaborated feedback gives more information to the learner beyond the correctness of a solution or the score obtained. It has five sub-categories namely: *knowledge about task constraints*, *knowledge about concepts*, *knowledge about mistakes*, *knowledge about how to proceed*, and *knowledge about meta-cognition*. Each of these sub-categories is further divided into specific feedback types, including

hints/explanations on the type of task, *number of mistakes*, *location of mistakes*, *hints/explanations on the type of errors*, *hints/explanations on sources of errors*, *bug-related hints for error correction*, *hints/explanations on task-specific strategies*, *hints/explanations on task-processing steps*, *worked-out examples*, and *guiding questions*.

In a domain-specific classification [12], three types of feedback that can be given to students learning automata construction in the FLA course were highlighted. They include *binary feedback, counterexamples*, and *descriptive hint feedback*.

Both the content-based and domain-specific classifications of feedback are related to each other. *Binary feedback* and *descriptive hint feedback* in [12] are similar to the *knowledge of result* and *elaborated feedback* in [31], respectively. Also, *bug-related hints* in the Narciss classification imply *counterexamples* in the former. In the context of REs, a counterexample is a string on which the student's RE does not meet the input specification. It could be a string that the RE represents, which it should not or a string that should be represented but is not.

The literature has recorded several advances in the field of automatic feedback generation for REs, with the focus being the *knowledge of performance* and *knowledge of results* feedback [5, 8, 13, 24, 42]. For *elaborated feedback*, only *bug-related hints* and *worked-out examples* have been developed for students learning REs [5, 13]. However, an attempt has been made at the automatic generation of the *location of mistakes* feedback [8]. Based on the literature study, feedback that includes what type of error the student has made, where the error is, and hints to fix the error has not been reported to be generated for REs.

The Hattie and Timperley [19] model of feedback guided the generation of feedback in the developed system. The model highlights four levels at which feedback can be given: the task level, the process level, the self-regulation level, and the self-level. Details of the feedback types generated are presented in Section 4.6.

2.3 Related Work

This section reports on previous research in any of the learning areas of REs that focused on error detection and feedback generation.

2.3.1 RE Exercises (RegeXeX). RegeXeX [8] is a system built to teach students how to write REs. The system presents *RE construction* exercises, assesses RE submissions and specifies if a submission is correct, incorrect, or has a syntax error. When the answer provided is incorrect, counterexamples are also provided. Using the RE submission and the solution (correct) RE, the RegeXeX tool produces two DFAs that define the language of all strings missing from the submission and the extra strings in the submission that should not be accepted. The two DFAs are thus used to generate counterexamples and determine the equivalence of both REs. If none of the DFAs has final states that are reachable from their start states, then the submitted RE and solution RE are equivalent.

The RegeXeX tool provides *binary feedback* and *knowledge of result* feedback. *Bug-related hints* are also provided in the form of counterexamples. However, the system attempts to show the location of the error, by placing the counterexamples under the student's RE, as a means to visualize the point where the error is. This approach, however, does not highlight the exact position of the error in the submitted RE or in the counterexample.

2.3.2 Automata Tutor. Automata Tutor [11–13] is an online tool. It is used to teach students different topics in the FLA course including REs, finite automata, context-free grammars, and Turing machines. The tool has been used by over 9,000 students across 30 universities in North America, South America, Europe, and Asia [11]. The tool was developed using algorithmic techniques grounded in program synthesis, logic, and the theory of formal languages. It was initially developed

for teaching Finite Automata and other FLA topics, but as newer versions were built, the provision for REs was included.

With respect to REs, the tool provides both *RE Construction* and *RE-String matching* exercises. In the RE construction exercises, performance feedback in the form of a grade is given, as well as counterexample feedback. The system does not generate feedback that contains the location of an error, or possible fixes for incorrect REs.

2.3.3 Simplified RE Algorithm (SRegED). Kakkar [24] developed a technique for the automatic grading of REs. The technique was based on the RE edit distance algorithm (RegED), and a modified version called simplified RegED (SRegED). RegED is the minimal number of edit operations (insertion, deletion, or substitution) that can be made to an incorrect RE submission to make it equivalent to a model solution [24].

The SRegED algorithm calculates the minimal changes needed to be applied to an incorrect RE answer to make it equivalent to the correct solution. The algorithm was successfully used to grade partially incorrect submissions.

The only feedback type offered by this system is the *knowledge of performance* because the algorithm developed can only grade submissions. The algorithm is based on the concept of RE edit distance, and this can only be used to determine the closeness of a student's answer to the correct solution. It cannot be used to give concrete feedback to students.

2.3.4 RE Tutor (ReExpress). The ReExpress tool [5] is a system which presents RE exercises on three of the learning areas namely, *RE construction*, *RE-String matching*, and *String generation from REs*.

The tool grades the learner's submission and gives feedback. Information on how the grading and feedback were generated is not reported.

The feedback types provided are *knowledge of performance* and *worked-out examples*. However, other feedback types are not generated.

2.3.5 Summary. The *knowledge of performance* feedback has been developed in ReExpress, SRegED, and Automata Tutor [5, 13, 24, 42], while the *knowledge of result* feedback has been generated in RegeXeX and Automata Tutor [8, 13]. For the elaborated feedback types, *bug-related hints* have been generated in RegeXeX and Automata Tutor [8, 13], and *worked-out examples* in ReExpress [5].

The feedback types generated in the reviewed tools only inform students of their performance and provide counterexamples (in some situations). However, feedback that is essential for students learning a topic like REs, which contains information on the type of the error, location of the error, and hints/guide to fix the error has not been generated. Beyond knowing how they have performed and whether their answer is right or wrong, the knowledge of what the error is, the location of the error, and what next to do to fix the error and get the right solution is important for the students [12]. This is because it will help to reduce the frustration faced [44], while increasing the motivation to forge ahead in the learning process.

Furthermore, the techniques that have been developed cannot individually generate the required fixes needed to be made by a student.

Therefore, this research sought to develop techniques for the automatic generation of high-level descriptive feedback, that includes the location of the error(s), type of error(s), and hints on how to fix the error. This will enable students to arrive at a correct solution when solving questions on REs.

3 Error Classification for REs

To generate feedback for the learning of REs, it is important to first understand the types of errors that learners can make. This research is based on the error classification framework that was

developed and reported in [32]. According to [32], there are three types of errors made by students when learning REs, namely: *Syntax errors*, *Slight errors*, and *Logical errors*.

To help give clarity to the examples used in this section, we first give the formal definition of an RE.

3.1 Formal Definition of an RE

First we explain some basic terminology. Given two languages A and B, the union of the two languages is the set of strings in either A or B or both. The concatenation of the two languages is the set of all strings formed by taking a string in A and concatenating it with a string in B, where concatenating is simply a string followed by another. The Kleene star of a language is the set of all strings formed by concatenating zero or more strings in the language together while the Kleene plus of a language is the set of all strings formed by concatenating one or more strings in the given language.

R is an RE if R is [21, 45]:

(1) a, for some a in the alphabet Σ,
(2) λ, the empty string,
(3) $\emptyset$, the empty language,
(4) $R_1 + R_2$, where R_1 and R_2 are REs,
(5) R_1R_2, where R_1 and R_2 are REs,
(6) R_1^*, where R_1 is an RE, or
(7) (R_1), where R_1 is an RE.

The REs a and λ represent the languages $\{a\}$ and $\{\lambda\}$, respectively. The RE $\emptyset$ represents the empty language. The REs $R_1 + R_2$ and R_1R_2 represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, respectively. The RE R_1^* represents the language obtained by taking the Kleene star of the language R_1. And finally, the RE (R_1) denotes the same language as R_1.

For simplicity, and as a shorthand notation, two additional representations for REs exist as, R_1^+ and R_1^i [21, 45]. The RE R_1^+ represents the language obtained by taking the Kleene plus of the language R_1, it is equivalent to $R_1R_1^*$. The exponential notation on the other hand R_1^i is used to represent the concatenation of i copies of the language R_1.

Next, the types of errors are explained.

3.2 Syntax Errors

This type of error occurs when students violate the standard rules of writing REs. For example, ending an RE with an incorrect symbol, use of an invalid symbol, improper use of parentheses and other bracket delimiters, and improper use of operators.

For example, the RE $(aa)^+\underline{+}$ has a syntax error. A union operator has been used to end the RE which is invalid, as every union operator must have an operand on its right-hand side.

3.3 Slight Errors

Slight errors are errors that students make due to a minimal slip or mistake. According to [32], a slip is an error that can be corrected by a single edit operation, i.e., a single deletion, insertion, or substitution operation. Types of slight errors in REs include *misuse of an operator*, *omission of an operator*, *incorrect bracket delimiters*, *mismatched parentheses*, and an *incorrect alphabet-symbol*. These are the only operations categorized as slight errors in this work. The types of slight errors are explained in Section 4.3.

For example, if a student writes $((a + b)b)(a + b + \lambda)$ instead of $((a + b)b)^+(a + b + \lambda)$, the Kleene plus required has been omitted, thus making the submission incorrect. This is considered a

slight error because a single edit, an insertion of the Kleene plus operator would fix the error in the incorrect RE.

There is an overlap between some syntax and slight errors; *incorrect bracket delimiters* and *mismatched parentheses* can either be seen as syntax or slight errors. Based on the process of error detection presented in Section 4, syntax errors are detected before slight errors. Therefore, in this research, these error categories are treated as syntax errors.

3.4 Logical Errors

Logical errors occur in REs that do not fully meet the specification of the regular language given in the question. These REs are usually void of syntax errors and cannot be fixed by a single edit like the slight error. There are three types of logical errors and they are:

(1) *Additional Restriction Error*: This error occurs when an RE does not represent some valid strings that are meant to be represented. This indicates the presence of additional restrictions (either implicitly or explicitly) in the student's RE.
(2) *Omitted Restriction Error*: This error occurs when the RE represents invalid strings. This indicates that the RE does not specify all required restrictions as stated (or implied) in the question.
(3) *Incorrect Restriction Error*: This error occurs when the RE does not represent some valid strings while it represents some invalid strings. Here, the RE contains both the additional and omitted restriction error.

For example, consider the following question:

Give an RE for the following language, $L = \{w \in \{a, b\}^* \mid |w| \geq 2$ and every second symbol is a $b\}$.

A sample correct solution to the question is $((a + b)b)^+(a + b + \lambda)$.

If a student's submission to the question is $((a + b)b)^+(\lambda + a)$, strings generated from the RE cannot end with bb; e.g., this RE does not represent the string abb; therefore, the RE contains an additional restriction error.

If another submission to the same question is $(a + b)b(a + b)^*$; the RE incorrectly represents the string $abaa$. The restriction needed to ensure that every second symbol is a b is missing; therefore, the RE contains an omitted restriction error.

The next section explains the process and algorithms developed to automatically detect these errors in students' RE submissions.

4 Error Detection

Given an *RE Construction* exercise, there are usually several correct solutions. All the different correct solutions always represent the same language, that is, they are all equivalent. Therefore, the instructor chooses only one correct solution and enters that into the system. For the remainder of this article, we'll refer to the correct solution being used as "the model RE." Regardless of the correct solution used by the student, it would always be equivalent to the model RE.

Given a student's submitted RE and the model RE provided by an instructor, if the student's RE is not equivalent to the model RE, the task is to identify the error position in the student's RE and give appropriate feedback that can be used to make necessary corrections to the RE.

This section describes the process and the algorithms developed for detecting the different types of errors that can occur in a student's RE.

Given a submitted/student's RE, the flow of error detection is as follows:

(1) *Detect Syntax Errors*: The first step of error detection is to do a syntax check on the submitted RE. It is necessary to do this first because the submitted RE cannot be correct if it has syntax

errors and an equivalence check can only be performed on syntactically correct REs. If the submitted RE has syntax errors, the error details are collected; otherwise, the process moves to the next step.

(2) *Equivalence Check*: This step checks for the correctness of the student's RE. The submitted RE and the model RE are checked for equivalence. If the student's RE and the model RE are equivalent, this means there is no error and the process terminates; otherwise, the process moves to the next step.
(3) *Detect Slight Error*: The next step is to check for the presence of a slight error: if a slight error is present, the error is reported and the process terminates; otherwise, this means there is a logical error and the process continues to the next step.
(4) *Detect Logical Errors*: Finally, we determine the type and location of the logical error present in the student's RE.

Next, the details and algorithms used at each step are explained.

4.1 Detecting Syntax Errors

The process of detecting syntax errors involves two steps, namely, lexical analysis and syntax analysis. These two steps are synonymous with the first two phases of a compiler.

4.1.1 Lexical Analysis. This involves the use of a lexical analyzer to break down the RE into tokens. In a student's RE submission, there are ten possible types of tokens. They are:

—letters or characters of the alphabet (alphabet-symbol),
—the union operator (union),
—the concatenation operator (concat),
—the Kleene star operator (kstar),
—the Kleene plus operator (kplus),
—the exponent (exp),
—the opening parenthesis (lparen),
—the closing parenthesis (rparen),
—the empty string (emptystring),
—the empty language (emptylang)

An additional token EOF (end-of-file) is used to represent the end of an RE.

Each token has a type, value, start position, and end position. For example, given the RE $(a+b)+c$, the tokens are lparen, alphabet-symbol, union, alphabet-symbol, rparen, union, and alphabet-symbol. The fourth token in the RE is of *type* alphabet-symbol, its *value* is b, and its *start* (and *end position*) is 3 (counting starts from index 0).

The lexical analyzer takes in the RE as a stream of individual characters and then groups them into appropriate tokens. If an unrecognized character is seen or if none of the specified tokens can be formed, the lexical analyzer reports the character and its position in the RE as an error location.

For example, if the alphabet of a given question is the set $\{a, b\}$, and the student enters the following RE, $(a + c + \lambda)$, the lexical analyzer will stop at position 3 and report an invalid symbol.

The identification of tokens in the RE and their positions is also important for the reporting of error positions in the other stages of error detection.

4.1.2 Syntax Analysis. The task of syntax analysis is executed by a parser. While the lexical analyzer verifies that all tokens are valid, the parser ensures that the tokens are well-formed. A context-free grammar that represents the rules of a well-formed RE is presented in Figure 1. The

$A \rightarrow A$ union $B \mid B$ (Rule 1)

$B \rightarrow B$ concat $C \mid C$ (Rule 2)

$C \rightarrow D$ kstar $\mid D$ kplus $\mid D$ exp $\mid D$ (Rule 3)

$D \rightarrow$ alphabet-symbol | emptystring | emptylang | lparen A rparen (Rule 4)

Fig. 1. The grammar to represent students' REs.

grammar has four production rules marked Rules 1–4, with non-terminal symbols A, B, C and D. There are 10 terminal symbols which are synonymous with the tokens presented in Section 4.1.1.

As in a compiler, the parser makes an attempt to derive the submitted RE using the grammar. If the parser is able to derive the entire RE from the grammar, it means the submitted RE has no syntax error; otherwise, a syntax error is present. The point at which the student's RE cannot be generated from the grammar is the point of the error, i.e., when there is no rule of the grammar to represent the current token being parsed.

Once an error is detected during the lexical or syntax analysis, the program quits with an informative error containing the current token (or symbol) and position. This information is then used to generate appropriate feedback. For example, given the RE $(a.a)^+$+, when the parser encounters the last character which is a union operator, it expects another operand to its right-hand side (Rule 1), because the union operator is not meant to end an RE. However, since there is no other operand, the current token, which is the union operator, is reported as an error location.

If an RE does not contain syntax errors, the next step is to check if the submitted RE is equivalent to the model RE.

4.2 Checking for the Equivalence of REs

When a student's RE has been verified to be syntactically correct, it is then checked for equivalence with the solution RE. An RE equivalent to the solution RE is correct and does not contain any error.

Two REs are said to be equivalent if they represent the same language, regardless of whether they are similar string-wise or not. Different approaches to determining the equivalence of REs exist. The approach specified in [8] was chosen because it is also useful in the process of detecting logical errors. The method works by converting the REs into DFAs and then comparing the DFAs; if the DFAs are equivalent, then the REs are equivalent.

The steps are as follows:

Given two REs, R_1 and R_2.

(1) Convert both REs into **Non-Deterministic Finite Automata (NFAs)** using the Glushkov algorithm [16, 37]. Let the NFAs be represented by N_1 and N_2.
(2) Convert the NFAs to DFAs using the subset construction method [1, 29]. Let the DFAs be represented by M_1 and M_2.
(3) Minimize the DFAs to get a unique representation of both DFAs using Hopcroft's algorithm [6, 20]. The minimal DFAs are D_1 and D_2.
(4) Compare the two minimal DFAs D_1 and D_2 for equivalence:
    (a) Get the complement of both D_1 and D_2, represented as $D_{\overline{1}}$ and $D_{\overline{2}}$, respectively.
    (b) Next, find the intersection of D_2 and $D_{\overline{1}}$. The outcome of the intersection $D_{2 \cap \overline{1}}$ accepts the language of strings accepted by D_2 but not accepted by D_1.
    (c) Also, find the intersection of D_1 and $D_{\overline{2}}$ represented as $D_{1 \cap \overline{2}}$.
    (d) If the languages of both $D_{2 \cap \overline{1}}$ and $D_{1 \cap \overline{2}}$ are empty, then D_1 is equivalent to D_2. Otherwise, if any of the DFAs accepts a string, then D_1 is not equivalent to D_2.

If the student's RE is not equivalent to the model RE, the next step is to check for the presence of a slight error. In the next section, the approach to detecting slight errors is presented.

4.3 Detecting Slight Errors

As discussed in Section 3.3, if a single edit can make an incorrect RE to be equivalent to the model RE, then we assume the presence of a slight error. In other words, if the RegED between two REs is one, it is considered a slight error. A RegED of one is a reasonable benchmark to cater for possible slips while ensuring that actual errors (that are not slips) are not categorized as slight errors. It is good to note here that there is a restriction on the type of edit operations allowed to those defined in this section. The three types of slight errors considered are, *misuse of an operator*, *omission of an operator*, and *incorrect alphabet-symbol*.

The RegED itself is not computed because it is not needed, and it would be computationally expensive to do so. The algorithm instead performs a check to determine if a RegED of one exists between the model RE and an incorrect RE. If a single edit performed on an incorrect RE makes it equivalent to the model RE, then the RegED between the two REs is one.

An allowable edit that can be performed on an incorrect RE can be an insertion, deletion, or substitution operation, such that the RE remains syntactically valid. Next, the allowable edits that can be classified as slight errors are explained, based on the types of slight errors.

4.3.1 Misuse of an Operator. This occurs when an operator has been used incorrectly, such that the deletion of an operator or its substitution with another will make the language of the incorrect RE equal to the language of the model RE. The allowable edit operations for this to occur are the substitution of a Kleene plus with a Kleene star and vice-versa, the substitution of a concatenation operator with a union and vice-versa and the deletion of a Kleene star or a Kleene plus.

4.3.2 Omission of an Operator. This is when a required operator in the RE solution is missing, such that the insertion of an operator would make the language of the incorrect RE equal to the language of the model RE. The allowable edit operations include the addition of a Kleene plus or star after every symbol which is not already followed by a Kleene plus or star and the addition of a Kleene plus or star after every closing parenthesis ")," which is not already followed by a Kleene plus or star.

4.3.3 Incorrect Alphabet-Symbol. This is when a symbol of the given alphabet for the RE exercise has been used incorrectly in place of another alphabet-symbol. The permitted edit operation is the substitution of every alphabet-symbol in the student RE with every other symbol of the given alphabet. Although this operation could be computationally expensive in REs over a large alphabet, this is usually not the case for the RE construction exercises in a typical FLA class. Notwithstanding, in the occurrence of this, the algorithm developed will still be able to generate results, but would perform slowly, depending on the alphabet size.

4.3.4 Algorithm for Detecting Slight Errors. The algorithm for detecting slight errors is given in Algorithm 1. The inputs to the algorithm are the student's RE (R_1) and the model RE (R_2).

At the beginning of the algorithm, the $error_found$ variable is set to false and the $error_type$ and $error_position$ variables are initialized as not known (Lines 1–3). The first step of the algorithm is to check the incorrect/student RE for the misuse of operator error. All the allowable edits under the *misuse of operator* error are carried out one after the other (Line 5). After each edit, an equivalence check is performed to determine if the language of the new/edited student RE is the same as the language of the model RE (Line 7). If the new student RE is equivalent to the model RE, then the error has been found (Line 8), the error type is set to the *misuse of operator* error (Line 9), and the position where the edit was performed is saved as the error position (Line 10). The algorithm

Algorithm 1: Detecting the Type and Location of a Slight Error

Input : R_1, R_2
Output: Presence and position of a slight error

```
1  error_found ← False
2  error_type ← not known
3  error_position ← not known
4  check for misuse of operator error:
5  foreach allowed edit operation do
6  |  New_R1 ← Perform edit on R1
7  |  if L(New_R1) == L(R2) then
8  |  |  error_found ← True
9  |  |  error_type ← misuse of operator
10 |  |  error_position ← position of edit
11 |  |  break;
12 if error_found == False then
13 |  check for omission of operator error:
14 |  foreach allowed edit operation do
15 |  |  New_R1 ← Perform edit on R1
16 |  |  if L(New_R1) == L(R2) then
17 |  |  |  error_found ← True
18 |  |  |  error_type ← omission of operator
19 |  |  |  error_position← position of edit
20 |  |  |  break;
21 if error_found == False then
22 |  check for incorrect alphabet-symbol error:
23 |  foreach allowed edit operation do
24 |  |  New_R1 ← Perform edit on R1
25 |  |  if L(New_R1) == L(R2) then
26 |  |  |  error_found ← True
27 |  |  |  error_type ← incorrect alphabet-symbol
28 |  |  |  error_position← position of edit
29 |  |  |  break;
30 if error_found == False then
31 |  RE does not contain a slight error
```

then terminates (Line 11). If all allowable edit operations for the *misuse of operator* error have been carried out and the error is not present, the RE is examined for the *omission of operator* error. The same process as in the *misuse of operator* error is again performed. All allowable edits are carried out one after the other (Lines 14 and 15) and an equivalence check is performed after each edit (Line 16). If the error is found, the details are captured (Lines 17–19) and the algorithm terminates (Line 20); otherwise, the algorithm checks for the *incorrect alphabet-symbol* error (Lines 22–29). If the RE has been checked for all possible slight errors and none is detected, the algorithm terminates and reports the absence of a slight error (Lines 30 and 31).

4.4 Detecting Logical Errors

The process of detecting a logical error is in two phases. The first phase involves detecting the type of logical error present, i.e., whether the error is an *additional restriction error*, an *omitted restriction error,* or an *incorrect restriction error*. The second phase then identifies the location of the error.

4.4.1 Detecting the Type of Logical Errors. Algorithm 2 presents the process of detecting the type of logical error present in a student's RE. The approach is based on the method used for equivalence check as explained in Section 4.2. The two DFAs which are the by-products of the equivalence check are used as input to the algorithm. $D_{2 \cap \bar{1}}$ defines the language of strings accepted by D_2 (from the model RE) but not accepted by D_1 (from student's RE). $D_{1 \cap \bar{2}}$ defines the language of strings accepted by D_1 (from student's RE) but not accepted by D_2 (from the model RE).

If the languages of both $D_{2 \cap \bar{1}}$ and $D_{1 \cap \bar{2}}$ are not empty, i.e., if there is at least one string accepted by both DFAs, then the student's RE contains an *Incorrect Restriction Error* (Lines 1 and 2). If only the language of $D_{2 \cap \bar{1}}$ is not empty, then the student's RE contains an *Additional Restriction Error* (Lines 5 and 6). Otherwise, if only the language of $D_{1 \cap \bar{2}}$ is not empty, then the student's RE contains an *Omitted Restriction Error* (Lines 8 and 9).

Additionally, the counterexamples for the student's REs are obtained. In Lines 3 and 4, the counterexamples for the incorrect restriction error are obtained. Since this error category is a combination of both the additional restriction and the omitted restriction errors, two sets of counterexamples are obtained using the method employed by the individual error categories. These are detailed in the next steps. For the additional restriction error, the counterexamples for the student's RE are obtained as indicated on Line 7. For this error type, the first 10 strings from the language $L(D_{2 \cap \bar{1}})$ according to the lexicographic order are obtained as the counterexamples. If the language does not have up to 10 words, all the words in the language are obtained. As explained in Section 4.4.2, no error location is determined for additional restriction errors; therefore, the counterexamples are given as feedback to the students. The number 10 was considered a reasonable number to ensure adequate coverage of the different types of counterexamples and to allow the student to study the pattern of strings not represented by the RE.

For the omitted restriction error, the counterexamples for the student's RE are obtained as shown on Line 10. For this error category, the counterexamples are also used to get error locations in the student's RE; therefore, counterexamples of different lengths are obtained. If the initial counterexample is of length x, all strings of length x, $x + 1$ and $x + 2$ are obtained. If the number of strings up to length $x + 2$ is less than three, the first three strings in that language (in lexicographical order) are obtained. The minimum number of counterexamples obtained is three if there are at least three strings in the language.

4.4.2 Detecting the Location of Logical Errors. This section describes the algorithms developed for the detection of the location of logical errors in students' REs.

Additional Restriction Error. Error locations are only detected for omitted restriction errors and incorrect restriction errors. Error locations are not detected for additional restriction errors. Recall that an additional restriction error is present when an RE does not represent some valid strings, i.e., all strings that can be represented by the RE are valid; however, some strings that are meant to be represented are missing. Since all the strings represented by the RE are valid, in most cases, the RE in itself does not have an erroneous portion to be identified, rather it is missing some components. Although there could be portions of the RE that can be modified to ensure correctness, since they do not directly represent invalid strings, they are rather not identified, so as not to confuse the student. Moreover, the missing portions can, in most cases, be inserted into the RE in multiple valid locations. For example, given the model RE $a^2 + a^3 + a^4 + a^6 + a^8a^*$ and an incorrect RE

Algorithm 2: Detecting the Type of Logical Errors

Input : $D_{2 \cap \bar{1}}$, $D_{2 \cap \bar{1}}$
Output: Type of logical error and counterexamples

1 **if** $L(D_{2 \cap \bar{1}}) \neq \phi$ ***and*** $L(D_{1 \cap \bar{2}}) \neq \phi$ **then**
2 $\quad$ *error_type* ← ***Incorrect Restriction error***
3 $\quad$ *counterexample_add* ← get the first 10 (maximum) strings belonging to $L(D_{2 \cap \bar{1}})$
4 $\quad$ *counterexample_omitted* ← get at least 3 strings belonging to $L(D_{1 \cap \bar{2}})$
5 **else if** $L(D_{2 \cap \bar{1}}) \neq \phi$ **then**
6 $\quad$ *error_type* ← ***Additional Restriction Error***
7 $\quad$ *counterexample_add* ← get the first 10 (maximum) strings belonging to $L(D_{2 \cap \bar{1}})$
8 **else if** $L(D_{1 \cap \bar{2}}) \neq \phi$ **then**
9 $\quad$ *error_type* ← ***Omitted Restriction Error***
10 $\quad$ *counterexample_omitted* ← get at least 3 strings belonging to $L(D_{1 \cap \bar{2}})$

$a^3 + a^4 + a^6 + a^8a^*$; the incorrect RE does not represent the string *aa*. This missing string can be inserted at the beginning of the RE, at the end, or in other places in between. Therefore, feedback is given to the student using several counterexamples, rather than identifying error locations.

Omitted Restriction Error. The technique used to detect the location of an omitted restriction error leverages the counterexamples generated for that RE. This is possible because the counterexamples obtained from an RE with the omitted restriction error are invalid strings, i.e., strings not meant to be represented. It is therefore possible to locate an error position in such a string and then identify the portion of the RE that represents the error position in the string; this is the approach used. The technique is a two-step process. The first step locates the error location in the counterexamples, while the second step identifies the position of the student's RE that represents the error location identified in the counterexample.

The first step is presented in Algorithm 3. The inputs to this algorithm are the model RE (represented as R_2), and the counterexamples obtained from Algorithm 2 (*counterexample_omitted*).

For each counterexample, the error location is identified. If the counterexample is the empty string, or if the length of the counterexample is less than the minimum length of strings in the language of the model RE, the error position is taken as the last character of the counterexample (Lines 4–7). This ensures that the entire RE is used in the second step of the algorithm.

For other strings, an attempt is made to match the counterexample with the model RE (Lines 8–13). In this context, "to match" means to search for the pattern (the model RE) in the counterexample (string). The search is done from the beginning of the string, and it returns the portion of the string that was matched by the pattern. This is done incrementally to identify the error point, i.e., starting with the first token of the model RE and moving from left to right, taking each token at a time, the counterexample is matched with the model RE. At each point, the length of the counterexample that was matched is saved. At the end of the matching process, i.e., after the last token of the model RE has been reached, the longest length of the counterexample that matched the model RE is obtained. If the length obtained (the highest length matched in the counterexample) is less than the length of the counterexample, the next character after the length obtained is the point of error. This is because it is at this point that the model RE could not match the counterexample (Lines 14 and 15).

It is possible for the entire counterexample to be matched due to the repetitive nature of the Kleene plus and the Kleene star, when present in an RE (Line 16); if this is the case, the counterexample is compared with valid strings from the model RE, instead of the model RE itself, to determine the

Algorithm 3: Detecting the Location of Omitted Restriction Errors—Step 1

```
   Input  : R2, counterexample_omitted
   Output : Location of error in counterexamples
 1 longest_match ← 0
 2 length_of_counterexample ← get the length of the counterexample
 3 foreach counterexample in counterexample_omitted do
 4     if counterexample == emptystring then
 5         error_point_in_ce ← last character of counterexample
 6     else if length_of_counterexample < minimum length of string in L(R2) then
 7         error_point_in_ce ← last character of counterexample
 8     foreach Regular Expression unit in R2 do
 9         re_unit ← get the next (first) Regular Expression unit
10         cummulative_re_units ← append re_unit
11         length_of_match ← match cummulative_re_units with counterexample
12         if length_of_match > longest_match then
13             longest_match ← length_of_match
14     if longest_match < length_of_counterexample then
15         error_point_in_ce ← longest_match
16     else if longest_match == length_of_counterexample then
17         valid_words ← get valid words from DFA_sol
18         closest_valid_words ← get closest matches to the counterexample
19         difference_counter_list ← {}
20         foreach close_valid_word in closest_valid_words do
21             difference_counter ← 0
22             foreach char_a in counterexample and char_b in close_valid_word do
23                 if char_a == char_b then
24                     do nothing
25                 else
26                     difference_counter ← difference_counter + 1
27             difference_counter_list ← append difference_counter
28         highest_difference_count ← get highest count in difference_counter_list
29         if highest_difference_count == 1 then
30             error_point_in_ce ← get position of highest_difference_count
31         else
32             error_point_in_ce ← last character of counterexample
```

error position. First, valid words (strings) in the specified language are obtained, and the string edit distance is used to identify the closest valid words to the counterexample being examined. The counterexample is then compared with the closest valid words to identify the position(s) with the highest disparity (Lines 17–32).

After the error location in the counterexamples has been identified, the process moves to the second step, identifying the error location in the RE. The second step is presented in Algorithm 4. The inputs to this algorithm are: the counterexamples (*counterexample_omitted*), their error positions (*error_point_in_ce*), and the student's RE (R_1). The outcome is the error position(s) in

Algorithm 4: Detecting the Location of Omitted Restriction Errors—Step 2

Input : *counterexample_omitted*, *error_point_in_ce*, R_1
Output: Location of the omitted restriction error

```
1  foreach counterexample in counterexample_omitted do
2      if error position and character at error position are unique then
3          selected_counterexamples ← counterexample
4  foreach counterexample in selected_counterexamples do
5      ce_error_range ← get counterexample from index 0 to error_point_in_ce
6      foreach re unit in R1 do
7          current_re_unit ← get the next (first) RE unit
8          current_re_position ← get the next (first) RE position
9          cummulative_re_units ← append current_re_unit
10         if ce_error_range is a member of cummulative_re_units then
11             error_unit ← current_re_unit
12             error_position ← position of current_re_unit
13             break
14     if error position is not found then
15         ce_error_range ← get entire counterexample
16         repeat Lines 6 – 13
```

the student's RE. First, counterexamples with unique error details are selected. The error details consist of the error position and the character at that position. This is important because only unique counterexamples will yield unique error positions. This is indicated in Lines 1–3 of the algorithm.

Next, the algorithm iterates through all the selected counterexamples. For each counterexample, the algorithm attempts to identify the portion of the RE that generates the error portion of the counterexample. For each counterexample, first, the counterexample error range is determined. The counterexample error range is from the start of the counterexample to the error position. Next, starting with the first token of the student's RE, and moving from left to right incrementally, we check if the counterexample error range is a member of the language represented by the cumulative RE (the portion of the RE being examined at the time). If it is a member, then the current RE token being evaluated is the error position. If it is not, the student's RE is navigated from the left until the error position is found or the end of the RE is reached. Sometimes, when the counterexample error range is only a portion of the counterexample, the error position in the incorrect RE might not be identified. In this case, the entire counterexample is then used. This rarely happens; in the entire test data that was used, this only happened for two REs. The reason for attempting to use a counterexample range is to narrow down the position of error, however, the entire counterexample can be used when this is not possible. After all the selected counterexamples have been used in the second step, all unique error positions are reported.

Incorrect Restriction Error. An incorrect restriction error is present in a student's RE if the RE represents invalid strings and does not represent valid strings, both at the same time. In other words, this means that the RE contains both an additional restriction and an omitted restriction error. Therefore, a combination of the techniques used for both the additional restriction error and omitted restriction error is used.

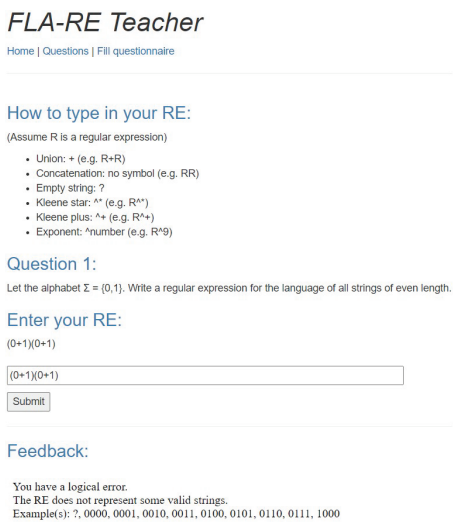


Fig. 2. Screenshot of the developed tool.

4.5 Implementation of the Algorithms

All the described algorithms were implemented using the Python programming language. A sample screenshot of the developed tool is shown in Figure 2. The standard algorithms used for the equivalence check have already been implemented in the FAdo library; therefore, these were reused. FAdo is a library that provides the implementation of several standard automata operations, and it is available for public use [38].

4.6 Feedback Generation

The Hattie and Timperley [19] model of feedback which guided the generation of feedback in the developed system highlights four levels at which feedback can be given: the task level, the process level, the self-regulation level, and the self-level. In this work, feedback was given to students at the task and process levels. In the task level, feedback is about a task, how well the task has been performed or executed. It is usually given to novices or beginners, and it is on this feedback type that other levels of feedback are built. Too much feedback should not be given at the task level, and this feedback should be given immediately and not delayed. Due to the characteristics of this feedback type, feedback on syntax errors was given at this level. When there is a syntax error, the exact syntax error is pointed out to the student. Although other errors might be present in the RE after the resolution of the syntax error, no additional information is given as feedback should not contain extra details at this level. At the second level, feedback is focused on the processes used to complete a task or the processes underlying a task. Feedback should be focused at this level when a surface understanding of the task has already been achieved and should be used to enhance the understanding of the student. Feedback was given at this level for the slight and logical errors. If the student's RE does not have a syntax error, it indicates that the student has the basic understanding of constructing REs, thus surface understanding can be assumed. Since feedback at this level should enhance the understanding of the student, feedback given does not indicate the fix needed. This allows for some learning, as the student uses the details given to fix the error.

Feedback at the self-regulation level guides the student to create internal feedback and to self-assess performance at given tasks; this was not the focus of this research. The last level of feedback,

feedback at the self-level is usually personal, in the form of praise and unrelated to the performance on a task. For example, "you are a brilliant student." This is claimed to be the least effective type of feedback, because it contains no learning information; therefore, it also is not given by the developed system.

The error detection algorithms presented in Section 4 report the type and the location of errors found. Additionally, counterexamples are generated for the logical errors. Using these error details generated by the algorithms, pre-specified templates are populated to generate the actual feedback given to the students.

If the student's RE is a correct solution, the feedback given is: "Your solution is correct." Otherwise, if there is an error, the template used to generate feedback is made up of four components. The *error type* gives generic information about the type of error found. *Error details* gives more specific information on the error found. *Error position(s)* specifies the position of the error in the RE. And lastly, *Hint* gives information on what needs to be done to fix the error. This generic template is used for all error types with the exception of syntax errors, where the error details and hint are combined to make the feedback simple and clear.

5 Evaluation and Results

To evaluate the error detection algorithms developed, the accuracy of the error positions identified both in the incorrect REs and in the counterexamples was assessed. This section reports the approach and findings of this assessment.

5.1 Evaluating the Error Positions in the Incorrect REs

5.1.1 The Dataset. The dataset used for the evaluation comprises incorrect REs, which are answers to five RE construction exercises. The exercises were obtained from the past examination papers of an FLA course at the University of the Witwatersrand, Johannesburg between 2015 and 2017. Next, the RE exercises and a sample solution for each exercise are presented.

Question 1. Give an RE for the following language:

$$L = \{w \in \{a, b\}^* \mid |w| \geq 4 \text{ and the second symbols from the front and the back are equal}\}$$

Model Solution: $((a + b)a(a + b)^*a(a + b)) + ((a + b)b(a + b)^*b(a + b))$

Question 2. Give an RE for the following language:

$$L = \{a^n \mid n \geq 0, n \neq 3\}$$

Model Solution: $\lambda + a + aa + aaaaa^*$

Question 3. Give an RE for the following language:

$$L = \{w \in \{a, b\}^* \mid |w| \geq 4 \text{ and the fourth symbol from the back is a } b\}$$

Model Solution: $(a + b)^*b(a + b)(a + b)(a + b)$

Question 4. Give an RE for the following language:

$$L = \{a^n \mid n \geq 2, n \neq 5, n \neq 7\}$$

Model Solution: $a^2 + a^3 + a^4 + a^6 + a^8a^*$

Question 5. Give an RE for the following language:

$$L = \{w \in \{a, b\}^* \mid |w| \geq 2 \text{ and every second symbol is a } b\}$$

Model Solution: $((a + b)b)^+(a + b + \lambda)$

The developed algorithms were evaluated using both real and synthetic datasets. The total number of data items used for the evaluation was 249 incorrect REs. The real data items were 40 (containing syntax errors) and there were 209 synthetic data items (118 contained slight errors, 37 contained omitted restriction errors, and 54 contained incorrect restriction errors).

The real dataset was obtained from previously answered examination questions on REs by third-year computer science students at the University of the Witwatersrand, Johannesburg.

The synthetic dataset[1] was generated in this research, for two reasons. First, to properly assess the predicted error positions, annotated data were needed, i.e., incorrect REs with the error positions marked. The real dataset obtained did not have the error positions in the REs marked. Second, it did not contain all the types of errors; therefore, there was a need to generate more data for efficient testing. The synthetic data were generated by editing (via deletion, insertion, and substitution) 20 model solutions to Questions 1–5 to create incorrect REs. Four model solutions were used for each question. The generated REs and the positions of the edits performed were noted.

5.1.2 Annotating the Dataset. After the data were collated, the incorrect REs were then annotated with the expected error positions. This subsection highlights how the expected error positions were identified.

For syntax errors, the expected error position is the actual point of error. The 40 REs with syntax errors were checked and the error positions were marked manually. The position of syntax errors is not subjective; therefore, this was possible with the real dataset.

For slight and logical errors, the position of the edit and the type of edit that was performed were used to determine the expected error position in the RE.

(1) *For Insertions*: If the edit performed to derive the incorrect RE was an insertion operation, the characters inserted are marked as the error positions. For example, if the string $+\lambda$ is inserted in the model RE $((a + b)a(a + b)^*a(a + b))$, the incorrect RE with expected error position marked in red (and underlined) is $((a + b)a(a + b)^*a(a + b)\underline{+\lambda})$.
(2) *For Deletions*: If the edit performed was a deletion, the two RE units adjacent to the deleted portion are marked as the error positions. For example, for an RE $(a+b)a(a^*b^*)^*a(a+b)$, if a was deleted at positions 5 and 13 to generate the incorrect RE, the new RE with the expected error positions marked is $(a + b)\underline{)(}a^*b^*)\underline{)^*(}a + b)$. If the deletion occurred at the beginning or end of an RE, the first and last characters of the RE are marked as the expected error positions, respectively.
(3) *For Substitutions*: If a portion of an RE was changed, i.e., a portion of the RE was deleted, and replaced with other symbols at the same position, the newly inserted characters are marked as the error position, just like with the insertion operation.
Where a concatenation operator was used to replace a union operator, the two RE units adjacent to the implicit concatenation are marked as the expected error positions. This is similar to a delete operation because the concatenation operator is represented without any symbol.

5.1.3 Method of Evaluation. With the exception of syntax errors, the error position in an incorrect RE is relative rather than absolute due to the different ways an RE representing a particular language can be written. This means that the error in an RE can be fixed/corrected at different positions in the RE. As a consequence, different positions other than where the error was created in the synthetic data can be possible locations for the correction of an error. Notwithstanding, to get an objective estimate of the performance of the developed algorithms, the assessment was based solely on the expected positions derived from the edits performed on the RE. If any of the error

[1] https://github.com/OlaperiHS/Dataset-Automatic-Feedback-Generation-for-the-Learning-of-Regular-Expressions.

Table 1. Result Summary: Detecting Syntax Errors

No.	Incorrect RE	Feedback
1	$(aa)^{+}\underline{+}$	You have a syntax error. You should not end with this operator/symbol. The error position is marked in red (and underlined): $(aa)^{+}\underline{+}$ Verify the last symbol in your RE.
2	$(\lambda + \underline{w})^4 b^{+} b^{*}$	You have a syntax error. You have entered an invalid character. The error position is marked in red (and underlined): $(\lambda + \underline{w})^4 b^{+} b^{*}$ Check the character at the marked position.

positions predicted by the algorithm intersects with any of the expected error positions, it was marked as a correct prediction, otherwise, an incorrect prediction.

To carry out the actual assessment, each incorrect RE was given as input to the implemented tool. The algorithm then generated feedback which contains the predicted error position. The accuracy of the algorithms was then calculated using the percentage of the correct predictions.

5.1.4 Results. The developed algorithms correctly predicted the error positions of 205 REs out of the 249 used for the evaluation. The percentage of correct predictions was 82%.

Syntax Errors. For the REs with syntax errors, the algorithm correctly predicted the error positions in the 40 incorrect REs used, having an accuracy of 100%.

Table 1 shows two students' answers that had syntax errors with the error positions marked. The second column of the table shows the incorrect RE, with the expected error position marked in red (and underlined). The third column shows the generated feedback, the feedback contains the incorrect RE with the predicted error position marked in red (and underlined).

For the first RE $(aa)^{+}+$, the student's RE ended with a union operator.

In the second RE, $(\lambda + w)^4 b^{+} b^{*}$, an invalid character, letter w was used. All these error positions were correctly identified.

Slight Errors. For the REs with slight errors, the algorithm correctly predicted the error positions in 116 out of the 118 incorrect REs used, having an accuracy of 98.3%. The predictions for the two other REs were not wrong in themselves, but they were not at the expected position of error, as such they were recorded as incorrect predictions to keep with the assessment protocol. More details on this are presented in the discussion.

A summary of the feedback generated for the incorrect REs with slight errors is presented in Table 2.

In the first RE, $((a + b)b)\underline{(}a + b + \lambda)$, a Kleene plus was omitted from the model RE to generate the incorrect RE and this was correctly predicted as an omission of operator error. The position of error was also correctly predicted as seen marked in red (and underlined).

In the second RE, $(a\underline{+}b + bb)^{+}(a + b + \lambda)$, a concatenation operator was changed to a union operator and this was correctly predicted. Likewise in the third RE, $(ab + b\underline{a})^{+}(a + b + \lambda)$, a symbol of the alphabet was swapped to generate the incorrect RE and this was also correctly predicted by the algorithm developed to detect slight errors.

Logical Errors. For the omitted restriction error, the algorithm correctly predicted the error positions of 32 out of the 37 incorrect REs, i.e., 86.5% were correctly predicted. For the incorrect restriction error, the algorithm correctly predicted the error positions of 17 out of the 54 incorrect REs, i.e., 31.5% were correctly predicted.

Table 2. Result Summary: Detecting Slight Errors

No	Model RE	Edit Details	Incorrect RE	Feedback
1	$((a+b)b)^+(a+b+\lambda)$	Deleted $^+$	$((a+b)b)\underline{(}a+b+\lambda)$	You have a slight error: Omission of operator after the marked position The error position is marked in red (and underlined): $((a+b)b\underline{)}(a+b+\lambda)$
2	$(ab+bb)^+(a+b+\lambda)$	Changed concatenation to +	$(a\underline{+}b+bb)^+(a+b+\lambda)$	You have a slight error: Incorrect use of operator at the marked position The error position is marked in red (and underlined): $(a\underline{+}b+bb)^+(a+b+\lambda)$
3	$(ab+bb)^+(a+b+\lambda)$	Changed b to a	$(ab+b\underline{a})^+(a+b+\lambda)$	You have a slight error: Incorrect alphabet symbol at the marked position The error position is marked in red (and underlined): $(ab+b\underline{a})^+(a+b+\lambda)$

Table 3 gives a summary of the results. It shows four students' answers with logical errors.

The table shows the model RE, a summary of the edit that was performed to generate the incorrect RE, the incorrect RE with the expected error position(s) marked in red (and underlined), and the feedback generated (containing the incorrect RE with the predicted error position(s) marked).

The first RE in the table has an additional restriction error. As explained previously the feedback generated does not contain an error location. The feedback, however, contains counterexamples which are valid strings that the RE should represent but does not represent.

The second and third REs in the table contain omitted restriction errors. For the second RE, the string $+\lambda$ was inserted at two different locations of the model RE to generate the incorrect RE, and the algorithm was able to predict both error positions. For the third RE, as seen, the edit performed is the addition of the string $+\lambda+$ to the model RE. Two error positions were predicted, one of which matched the expected error position; therefore, this was also seen as a correct prediction.

The fourth RE in the table has an incorrect restriction error. This error type is a combination of the additional restriction and omitted restriction errors; therefore, the feedback is a combination of the individual feedback from both classes of errors. The first part of the feedback simply shows valid strings that are not represented, while the second part presents incorrect REs with the error positions marked. One of the predicted error positions is the expected error position; therefore, it is seen as a correct prediction.

5.2 Evaluating the Error Positions in the Counterexamples

As discussed in Section 4.4.2, during the process of determining the error position in an RE with an omitted restriction error, the first step is getting the error position in the counterexample. Therefore, the predicted error positions of the counterexamples were also assessed.

Table 3. Results Summary: Detecting Logical Errors

No	Model RE	Edit	Incorrect RE	Feedback
1	$((a+b)a(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$	Deleted $+b$	$((a)a(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$	You have a logical error. The RE does not represent some valid strings. Example(s): *baaa*, *baab*, *baaaa*, *baaab*, *babaa*, *babab*, *baaaaa*, *baaaab*, *baabaa*, *baabab*
2	$((a+b)a(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$	Inserted $+\lambda$	$((a+b)a(a+b)^*a(a+b)+\lambda)+((a+b)b(a+b)^*b(a+b)+\lambda)$	You have a logical error. The RE represents invalid string(s). Example(s): λ The error position(s) is marked in red (and underlined). $((a+b)a(a+b)^*a(a+b)+\lambda)+((a+b)b(a+b)^*b(a+b)+\lambda)$ $((a+b)a(a+b)^*a(a+b)+\lambda)+((a+b)b(a+b)^*b(a+b)+\lambda)$
3	$((a+b)a(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$	Inserted $+\lambda+$	$((a+b)a+\lambda+(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$	You have a logical error. The RE represents invalid string(s). Example(s): λ, $ab aa$ The error position(s) is marked in red (and underlined). $((a+b)a+\lambda+(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$ $((a+b)a+\lambda+(a+b)^*a(a+b)) + ((a+b)b(a+b)^*b(a+b))$
4	$a^2(\lambda+a+a^2+a^4+a^5a^+)$	Deleted a^2	$(\lambda+a+a^2+a^4+a^5a^+)$	You have a logical error. The RE represents some invalid strings and does not represent some valid strings. Your RE does not represent: *aaa* Your RE represents: λ, a, *aaaaaaa* The error position(s) is marked in red (and underlined) $(\lambda + a + a^2 + a^4 + a^5a^+)$ $(\lambda + a + a^2 + a^4 + a^5a^+)$ $(\lambda + a + a^2 + a^4 + a^5a^+)$

5.2.1 The Dataset. The data used for this evaluation are the counterexamples generated for the 91 REs with the omitted and incorrect restriction errors. Each of the 91 REs had at least one counterexample generated for it. Some REs had several counterexamples.

5.2.2 Annotating the Dataset. First, the expected (actual) error positions in the counterexamples were manually marked. Each counterexample was analyzed against the expected language of the RE, and the point in which the counterexample deviates from the given language was marked.

5.2.3 Method of Evaluation. The expected error positions marked in the counterexamples were checked against the predicted error positions. Again, similar to evaluating the REs, if any of the

error positions predicted by the algorithm intersects with any of the expected error positions, it was marked as a correct prediction, otherwise, an incorrect prediction.

5.2.4 Results. The results show that the error positions were all (100%) correctly predicted. The algorithm either predicted the expected error position or one of the correct positions (in situations where they were alternative error positions).

For example, looking at REs 2–4 in Table 3, the error positions in the counterexamples are marked. REs 2 and 3 are solutions to Question 1. The strings represented must be at least of length four, and the second and second-last symbols are expected to be the same, i.e., either both are letter "a" or letter "b". Where the counterexample was the empty string, the empty string was marked as expected. For the other counterexample $ab\underline{b}aa$, the second character was correctly marked to show that it was not similar to the second-last symbol as expected.

RE 4 is a solution to Question 4 where valid strings should be at least of length two and must not be of length five or seven. The counterexamples generated are $\underline{\lambda}$, $\underline{a}$, $aaaaaa\underline{a}$. The empty string and letter "a" are marked as expected as they are single characters. The string $aaaaaa\underline{a}$ has the last character correctly marked as the error position indicating that the character at that position should not be present.

6 Discussion

This section discusses the evaluation results.

6.1 Syntax Errors

The algorithm developed was able to identify the presence of a syntax error, the exact location, and the character(s)/symbol(s) of concern. As such, the feedback given is very explicit, providing adequate information to guide the student. Following the evaluation using the test data, the prediction accuracy for syntax errors was 100%.

6.2 Slight Errors

For the test dataset containing slight errors, a percentage of 98.3% accuracy was achieved. Out of the 118 incorrect REs, only two predictions of the error positions were incorrect. As mentioned, the assessment was fully based on the expected error positions which are determined by where the edit was performed on the model RE to generate the incorrect RE. This was done to ensure the objectivity of the assessment. Notwithstanding, a look at the two REs which were incorrectly predicted indicates that the predictions were not wrong in themselves.

The two REs are $(a+b)b\underline{+}((a+b)b)^*(a+b+\lambda)$ and $(ab+bb)\underline{+}(ab+bb)^*(a+b+\lambda)$. They were obtained from the model REs $(a+b)b((a+b)b)^*(a+b+\lambda)$ and $(ab+bb)(ab+bb)^*(a+b+\lambda)$, respectively, by the change of a concatenation operator to a union operator in the marked positions as shown. However, the predicted error positions were different, indicating the use of a Kleene star instead of a Kleene plus at the marked positions: $(a+b)b+((a+b)b)\underline{^*}(a+b+\lambda)$ and $(ab+bb)+(ab+bb)\underline{^*}(a+b+\lambda)$. A close look at the REs shows that indeed changing the Kleene star in the marked positions into a Kleene plus symbol will result in REs equivalent to the correct ones. Therefore, they are actually correct predictions which would guide an erring student on the right path. It is therefore in order to say that the accuracy of predicting the error positions in REs with slight errors is also 100%.

6.3 Omitted Restriction Errors

The percentage of correct predictions obtained for omitted restriction errors was 86.5%, i.e., 5 out of 37 REs were incorrectly predicted.

Out of these five incorrect REs, two can be claimed to be correct predictions. The two REs are $\lambda+a+aa\underline{+a}aa^*$ and $\lambda+a+aa\underline{+a}a^*$, with the expected error positions marked in red (and underlined). They were both obtained from the model RE $\lambda + a + aa + aaaaa^*$. The first RE had the string aa deleted while the second RE had aaa deleted from the latter part of the model RE marked in red (and underlined) in $\lambda + a + aa + \underline{aaaaa^*}$. As seen, since that latter portion consists of all a's it is a bit subjective to state the exact position where the symbols were deleted from. The expected error position can indeed be any position around the latter part of the model RE. Rightly so, the predicted error positions for both REs were $\lambda + a + aa + aa\underline{a^*}$ and $\lambda + a + aa + a\underline{a^*}$, respectively. As such, one can say the predicted positions were not wrong in themselves.

The other three mispredictions can be attributed to the mode of operation of the algorithm. The algorithm works by first identifying an error position in a given counterexample and then using that position to identify the error position in the incorrect RE. For the three REs, the error positions in the counterexamples were correct. Also, the error positions predicted in the REs were responsible for generating the erroneous part of the counterexamples. However, the predicted error positions were not the same as the expected error positions.

For example, one of the incorrect REs was $(\lambda + a + b)^*(b)(a + b\underline{+\lambda})(a + b\underline{+\lambda})(a + b\underline{+\lambda})$ with the expected error positions marked in red (and underlined). It was obtained by inserting the string $+\lambda$ in several positions of the model RE $(\lambda + a + b)^*(b)(a + b)(a + b)(a + b)$. The counterexample used with its error position marked in red (and underlined) was $\underline{a}aab$. This is in order as a correct solution should ensure that every fourth symbol from the back is a "b." The predicted error position is marked in red (and underlined), as in $\underline{(\lambda + a + b)^*}(b)(a + b + \lambda)(a + b + \lambda)(a + b + \lambda)$. As seen, the predicted error position is actually responsible for representing the erroneous portion of the counterexample; however, it is not the same as the expected error position where the error was introduced. This means the algorithm can be incorrect for a few predictions (3 out of 37). Perhaps the algorithm can be fine-tuned for this type of edge case to improve its performance.

Nonetheless, it is believed that since the student is presented with the error position in the counterexample alongside the identified error position in the RE, the feedback will still be useful.

6.4 Incorrect Restriction Errors

The prediction accuracy in REs with incorrect restriction errors was the lowest at 31.5%.

To generate feedback for an RE with an incorrect restriction error, a combination of the techniques used for both the additional restriction error and omitted restriction error is used. In reality, only the algorithm developed for the omitted restriction error is used to detect the error position in an RE with an incorrect restriction error, since an error position is not predicted for an RE with the additional restriction error. This might be the reason for the low prediction accuracy since an RE with an incorrect restriction error has additional characteristics that one with an omitted restriction error does not have. Therefore, an improvement might be to fine-tune the algorithm to make it more suitable for the incorrect restriction error or to develop another algorithm specialized for incorrect restriction errors. Nevertheless, the student has several counterexamples (from the technique used for additional restriction errors) which can assist in the further understanding of the error present in the RE.

In all, the overall prediction accuracy of the error positions in the incorrect REs, for all the algorithms is 82%. This can be considered a good performance based on the high value. Moreover, to the best of the authors' knowledge based on reviewed literature, no previous research has attempted to identify error positions in incorrect REs. An improvement of the algorithm for the incorrect restriction error will result in a better overall performance.

7 Conclusion and Future Work

In conclusion, in this research, techniques and algorithms to detect the type and location of logical errors in students' REs were developed by leveraging on existing concepts in the theory of computation. Also, the use of counterexamples to identify the error location in an incorrect RE was introduced. In line with this, an algorithm to identify the error location in a counterexample was developed.

Additionally, a dataset of incorrect REs annotated with their actual error positions was created. This can be used to facilitate the testing of algorithms built to detect errors in REs.

To take this research further, a thorough design and testing of a prototype tool can be carried out. This would help to measure the usability of the feedback in a real-life setting and determine whether the generated feedback is able to help students to resolve incorrect RE solutions.

To assess the impact of the developed system on students' learning and academic performance, a randomized control trial can be performed. Given a set of students, first a pre-test can be carried out to assess students' current knowledge level. Students would then be randomly assigned to either a control group or an experimental group. The experimental group would study with the use of the developed system, while the control group would only study conventionally. Subsequently, a post-test would be performed and analyzed using appropriate statistical tests to determine the impact of the system on the students' learning process.

Additionally, an instructor focused study can be carried out. This would survey instructors' perception of the feedback provided by the tool.

It would also be beneficial to represent the feedback generated in a pictorial way or in the form of a simulation. This will help students to "visualize" how strings are represented or not represented by their REs and thus see the error portions in their REs. For example, the RE can be represented in a graphical way (something similar to a state diagram or a finite automaton) and a form of simulation can be used to depict the step-by-step process of how a counterexample is represented (or not represented) by the RE while highlighting the error position in the counterexample and/or RE. Apart from helping to identify an error position, this can help to improve the comprehension of how an RE works to represent strings.

For REs with additional restriction errors, it will be beneficial to generate a summary of the strings not represented by the RE. Currently, the student is presented with a set of counterexamples; however, there is a limit to the number of counterexamples that can be presented to the student such that the feedback does not become overwhelming. Moreover, even if all the counterexamples can be presented, it will be more beneficial for the student to know what is common to all the counterexamples, or the different groups of counterexamples that exist.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques and Tools* (2nd. ed.). Pearson Addison-Wesley, Boston.

[2] Rajeev Alur, LorisD'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 1976–1982. Retrieved from https://www.ijcai.org/Proceedings/13/Papers/292.pdf

[3] Susan A. Ambrose, Michael W. Bridges, Michele DiPietro, Marsha C. Lovett, and Marie K. Norman. 2010. *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Jossey-Bass. DOI: https://doi.org/10.7899/JCE-12-022

[4] Álvar Arnaiz-González, Jose-Francisco Díez-Pastor, Ismael Ramos-Pérez, and César García-Osorio. 2018. Seshat—A web-based educational resource for teaching the most common algorithms of lexical analysis. *Computer Applications in Engineering Education* (2018). DOI: https://doi.org/10.1002/cae.22036

[5] Pavel Azalov, Michelle Cullen, and Robert Rinish. 2004. ReExpress: A tutor for regular expressions mentoring with technology. In *Proceedings of the 5th Conference on Information Technology Education (CITC5 '04)*. ACM, New York, NY, 268. DOI: https://doi.org/10.1145/1029533.1029601

[6] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. 2010. Minimization of automata. arXiv:1010.5318. Retrieved from https://arxiv.org/abs/1010.5318

[7] Paul Black and Dylan Wiliam. 1998. Inside the black box: Raising standards through classroom assessment. *Phi Delta Kappan* 80, 2 (Oct. 1998), 139–148. Retrieved from https://www.proquest.com/scholarly-journals/inside-black-box/docview/218533069/se-2

[8] Christopher W. Brown and Eric A. Hardisty. 2007. RegeXeX: An interactive system providing regular expression exercises. *ACM SIGCSE Bulletin* 39, 1 (2007), 445–449. DOI: https://doi.org/10.1145/1227504.1227462

[9] Can Cemal Cingi. 2013. Computer animation in teaching surgical procedures. *Procedia - Social and Behavioral Sciences* 103 (2013), 230–237. DOI: https://doi.org/10.1016/J.SBSPRO.2013.10.330

[10] Soner Şeker. 2013. Computer-aided learning in engineering education. *Procedia - Social and Behavioral Sciences* 83 (2013), 739–742. DOI: https://doi.org/10.1016/J.SBSPRO.2013.06.139

[11] Loris D'Antoni, Martin Helfrich, Jan Kretinsky, Emanuel Ramneantu, and Maximilian Weininger. 2020. Automata Tutor v3. In *Computer Aided Verification*. Shuvendu K. Lahiri and Chao Wang (Eds.), Springer International Publishing, Cham. DOI: https://doi.org/10.1007/978-3-030-53291-8_1

[12] Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How can automatic feedback help students construct automata? *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 9. DOI: https://doi.org/10.1145/2723163

[13] Loris D'Antoni, Matthew Weavery, Alexander Weinert, and Rajeev Alur. 2015. Automata Tutor and what we learned from building an online teaching tool. *Bulletin of European Association for Theoretical Computer Science (EATCS)* 3, 117 (2015), 144–158.

[14] Cristiano Galafassi, Fabiane Flores Penteado Galafassi, Rosa Maria Vicari, and Eliseo Berni Reategui. 2023. EvoLogic: Toward an ITS for teaching propositional logic. *International Journal of Artificial Intelligence in Education* 33 (2023), 35–58. DOI: https://doi.org/10.1007/s40593-021-00287-7

[15] Graham Gibbs and Claire Simpson. 2005. Conditions under which assessment supports students' learning. *Learning and Teaching in Higher Education* 1 (2005), 3–31. Retrieved from https://eprints.glos.ac.uk/3609/1/LATHE%201.%20Conditions%20Under%20Which%20Assessment%20Supports%20Students%27%20Learning%20Gibbs_Simpson.pdf

[16] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata (in Russian). *Russian Mathematical Surveys* 16, 5 (1961), 1. DOI: https://doi.org/10.1070/rm1961v016n05abeh004112

[17] Arthur C. Graesser, Shulan Lu, George Tanner Jackson, Heather Hite Mitchell, Mathew Ventura, Andrew Olney, and Max M. Louwerse. 2004. AutoTutor: A tutor with dialogue in natural language. *Behavior Research Methods, Instruments, & Computers* 36, 2 (2004), 180–192. DOI: https://doi.org/10.3758/BF03195563

[18] Rachel Harsley, BarbaraEugenio, Nick Green, Davide Fossati, and Sabita Acharya. 2016. Integrating support for collaboration in a computer science intelligent tutoring system. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems (ITS '16)*, Vol. 9684, Springer-Verlag, Berlin, 227–233. DOI: https://doi.org/10.1007/978-3-319-39583-8_22

[19] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of Educational Research* 77, 1 (2007), 81–112. DOI: https://doi.org/10.3102/003465430298487

[20] John E. Hopcroft. 1971. An *n* log *n* algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*. Elsevier, 189–196.

[21] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3rd. ed.). Pearson Education, Inc., Boston, MA.

[22] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY. DOI: https://dx.doi.org/10.1145/2534860

[23] Nenad Jovanović, Dragiša Miljković, Srećko Stamenković, Zoran Jovanović, and Pinaki Chakraborty. 2020. Teaching concepts related to finite automata using ComVis. *Computer Applications in Engineering Education* (2020), 1–13. DOI: https://doi.org/10.1002/cae.22353

[24] Himesh Kakkar. 2017. Automated Grading and Feedback of Regular Expressions. MSc. thesis. Rochester Institute of Technology, New York. Retrieved from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=10550&context=theses

[25] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 41–46. DOI: https://doi.org/10.1145/2899415.2899422

[26] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. *Towards a systematic review of automated feedback generation for programming exercises: Extended version*. Utrecht University, Utrecht, The Netherlands, 1–18 pages. Retrieved from https://dspace.library.uu.nl/bitstream/handle/1874/346321/Extended.pdf

[27] James A. Kulik and J. D. Fletcher. 2016. Effectiveness of intelligent tutoring systems: A meta-analytic review. *Review of Educational Research* 86, 1 (2016), 42–78. Retrieved from https://www.jstor.org/stable/24752869

[28] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '16)*. ACM, New York, NY, 70–80. DOI: https://doi.org/10.1145/2993236.2993244
[29] Kenneth C. Louden. 1997. *Compiler Construction: Principles and Practice* (1st. ed.). PWS Publishing Company, Boston.
[30] Nelma Moreira and Rogério Reis. 2005. Interactive manipulation of regular objects with FAdo. *ACM SIGCSE Bulletin* 37 (2005), 335–339. DOI: https://doi.org/10.1145/1151954.1067537
[31] Susanne Narciss. 2008. *Feedback strategies for interactive learning tasks*. In *Handbook of Research on Educational Communications and Technology*. David Jonassen, Michael J. Spector, Marcy Driscoll, M. David Merrill, and Jeroen van Merrienboer (Eds.), Vol. 3. Routledge, 125–144.
[32] Olaperi Yeside Okuboyejo, Sigrid Ewert, and Ian Sanders. 2021. Goofs in the class: Students' errors and misconceptions when learning regular expressions. In *ICT Education*. GeorgeWells, MoneloNxozi, and BobbyTait (Eds.), Springer International Publishing, Cham, 57–71. DOI: https://doi.org/10.1007/978-3-030-92858-2_4
[33] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 139 (2019), 29 pages. DOI: https://doi.org/10.1145/3360565
[34] Brit Paris. 2022. Instructors' perspectives of challenges and barriers to providing effective feedback. *Teaching and Learning Inquiry* 10 (Jan. 2022). DOI: https://doi.org/10.20343/teachlearninqu.10.3
[35] Carlos H. Pereira and Ricardo Terra. 2018. A mobile app for teaching formal languages and automata. *Computer Applications in Engineering Education* 26, 5 (2018), 1742–1752. DOI: https://doi.org/10.1002/cae.21944
[36] Nelishia Pillay. 2008. Teaching the theory of formal languages and automata in the computer science undergraduate curriculum. *South African Computer Journal* 12 (2008), 87–94. Retrieved from https://hdl.handle.net/10520/EJC28069
[37] Jean-Éric Pin. 2019. Mathematical foundations of automata theory. Lecture notes LIAFA, Université Paris.
[38] Rogério Reis and Nelma Moreira. 2002. *FAdo: Tools for Finite Automata and Regular Expressions Manipulation*. Technical Report. Universidade do Porto.
[39] Susan H. Rodger, Bart Bressler, Thomas Finley, and Stephen Reading. 2006. Turning automata theory into a hands-on course. *ACM SIGCSE Bulletin* 38 (2006), 379–383. DOI: https://doi.org/10.1145/1124706.1121459
[40] Susan H. Rodger, Jinghui Lim, and Stephen Reading. 2007. Increasing interaction and support in the formal languages and automata theory course. *ACM SIGCSE Bulletin* 39 (2007), 58–62. DOI: https://doi.org/10.1145/1269900.1268803
[41] Susan H. Rodger, Eric Wiebe, and Kyung Min Lee. 2009. Increasing engagement in automata theory with JFLAP. *ACM SIGCSE Bulletin* 41 (2009), 403–407. DOI: https://doi.org/10.1145/1539024.1509011
[42] Ariel Rosenfeld, Abejide Ade-Ibijola, and Sigrid Ewert. 2017. Regex Parser II: Teaching regular expression fundamentals via educational gaming. In *Annual Conference of the Southern African Computer Lecturers' Association (SACLA '17)*. Janet Liebenberg and Stefan Gruner (Eds.), Springer, Cham, 99–112. DOI: https://doi.org/10.1007/978-3-319-69670-6_7
[43] Vinay S. Shekhar, Anant Agarwalla, Akshay Agarwal, B. Nitish, and Viraj Kumar. 2014. Enhancing JFLAP with automata construction problems and automated feedback. In *Proceedings of the 2014 7th International Conference on Contemporary Computing (IC3)*. IEEE, 19–23. DOI: https://doi.org/10.1109/IC3.2014.6897141
[44] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 48, 15–26. DOI: https://doi.org/10.1145/2491956.2462195
[45] Michael Sipser. 2013. *Introduction to the Theory of Computation* (3rd. ed.). Cengage Learning, Boston.
[46] The Joint Task Force on Computer Science Curricula. 2024. *Computer Science Curricula 2023 The Final Report*. ACM, New York, NY. Retrieved from https://csed.acm.org/
[47] Vincent Tscherter. 2004. *Exorciser: Automatic Generation and Interactive Grading of Structured Exercises in the Theory of Computation*. Dissertation. Swiss Federal Institute of Technology, Zurich.
[48] Elena Verdú, Luisa M. Regueras, Eran Gal, Juan P. de Castro, María J. Verdú, and Dan Kohen-Vacs. 2017. Integration of an intelligent tutoring system in a course of computer network design. *Educational Technology Research and Development* 65, 3 (2017), 653–677. Retrieved from http://www.jstor.org/stable/45018572
[49] Huanhuan Wang, Ahmed Tlili, Ronghuai Huang, Zhenyu Cai, Min Li, Zui Cheng, Dong Yang, Mengti Li, Xixian Zhu, and Cheng Fei. 2023. Examining the applications of intelligent tutoring systems in real educational contexts: A systematic literature review from the social experiment perspective. *Education and Information Technologies* 28 (2023), 9113–9148. DOI: https://doi.org/10.1007/s10639-022-11555-x

Received 7 December 2022; revised 17 March 2025; accepted 4 May 2025