

A mobile app for teaching formal languages and automata

Carlos H. Pereira | Ricardo Terra 

Department of Computer Science, Federal University of Lavras, Lavras, Brazil

Correspondence

Ricardo Terra, Department of Computer Science, Federal University of Lavras, Postal Code 3037, Lavras, Brazil.
Email: terra@dcc.ufla.br

Funding information

FAPEMIG (Fundação de Amparo à Pesquisa do Estado de Minas Gerais)

Abstract

Formal Languages and Automata (FLA) address mathematical models able to specify and recognize languages, their properties and characteristics. Although solid knowledge of FLA is extremely important for a B.Sc. degree in Computer Science and similar fields, the algorithms and techniques covered in the course are complex and difficult to assimilate. Therefore, this article presents FLApp, a mobile application—which we consider the new way to reach students—for teaching FLA. The application—developed for mobile phones and tablets running Android—provides students not only with answers to problems involving Regular, Context-free, Context-Sensitive, and Recursively Enumerable Languages, but also an Educational environment that describes and illustrates each step of the algorithms to support students in the learning process.

KEYWORDS

automata, education, formal languages, mobile application

1 | INTRODUCTION

Formal Languages and Automata (FLA) is an important area of Computer Science that approaches mathematical models able to specify and recognize languages, their properties and characteristics [14]. Considering the importance of a solid knowledge of FLA on the profile of a B.Sc. in Computer Science, and observing a level of failing of almost 50% in the discipline [15], we propose a platform that can help students in the learning process.

There are several tools approaching the algorithms studied in FLA, although we noticed that one of the most popular tools, JFLAP [7], was created considering the technological advances of its time [2]. Nowadays, we claim that educators must invest time and effort in using mobile devices as educational tools because students always carry smartphones with themselves, and mobile and tablet usage has exceeded desktop usage since October 2016 [13]. Therefore, as the main contribution of this study, we developed an application for phones and tablets with Android operating system (OS) since Android is the most used OS by mobile devices [10].

In this article, we present FLApp (Formal Languages and Automata Application), a mobile application for teaching FLA that helps students by solving problems involving Regular, Context-free, Context-Sensitive, and Recursively Enumerable Languages (levels 3 to 0, respectively), in addition to create an Educational environment, a second contribution of this study. This environment describes and illustrates each stage of the algorithm execution to support students in the learning process. It is worth noting that the application might also be useful in other areas, for example, Regular and Context-free languages are widely employed in Compilers [1].

In a previous study [15], we observed that students presented low performance in algorithms related to grammars and automata. Thus, we develop a tool that cover most FLA content, and implemented features of all levels of Chomsky hierarchy [4]. FLApp contains more features than six of the seven related tools we compare in this article (see Section 2). JFLAP contains more features than FLApp, however, we also implemented some features that are not present in JFLAP.

The remainder of this article is organized as follows. Section 2 presents the related tools. Section 3 introduces

FLApp, its functionalities and what is expected with its use, whereas Section 4 describes its design and implementation. Section 5 conducts an evaluation to ensure the correctness of the implementation. Section 6 reports how the app is being practically used from the educational perspective. Finally, Section 7 concludes by reporting the main contributions.

2 | RELATED TOOLS

There are several tools in the Formal Languages and Automata area. Regarding automata simulators, Chakraborty et al. [2] describe the importance of these tools for the teaching of automaton theory and the way these tools have been developed for more than 50 years. In the early 1990s, it

was observed a renewed interest in automata simulators. It occurs due to the technological advance and the easiness to create better automata simulators [2]. Therefore, our work considers mobile technology as a new way of creates tools to aid in the teaching of FLA. In this section, we describe the most popular related tools whose features are summarized in Table 1. The symbols in the table indicate a level of support for the feature. The symbol ✓ indicates full support, ✓* indicates partial support, and x indicates no support.

2.1 | JFLAP

It is the most used supporting tool in universities for teaching FLA [7]. Developed in Java for desktops, JFLAP has been receiving new features and enhancements since 1996. Its last

TABLE 1 Comparative of the existing tools

Features	FLApp	JFLAP	FADL toolkit	JCT	Language emulator	SCTMF	GAM	Webworks applets
REGEX to NFA conversion	✓	✓	x	x	✓	x	x	✓
FSA to REGEX conversion	x	✓	x	x	✓	x	x	✓
FSA operations (complete, union, intersection, concatenation, complement, Kleene star)	✓*	✓*	x	✓	✓*	x	x	x
NFA to DFA conversion	✓	✓	✓	✓	✓	x	✓	✓
DFA minimization	✓	✓	✓	✓	✓	x	✓	✓
Simulation FSA	✓	✓	✓	✓	✓	✓	✓	✓
DFA to RG conversion	-	✓	x	x	x	x	x	✓
PDA to CFG conversion	✓	✓	x	x	x	x	x	x
Simulation PDA	✓	✓	x	x	x	✓	x	x
Simulation LBA	✓	✓	x	x	x	x	x	x
Simulation TM	✓	✓	x	✓	x	✓	x	x
Simulation MultiTape TM	✓	✓	x	x	x	x	x	x
Simulation TM as Enumerator	✓	✓	x	x	x	x	x	x
Grammar type identification	✓	-	x	x	x	x	x	x
RG to DFA conversion	✓	✓	x	x	x	x	x	✓
CFG to PDA conversion	✓	✓	x	x	x	x	x	x
Elimination lambda rules	✓	✓	x	x	x	x	x	x
Elimination chain rules	✓	✓	x	x	x	x	x	x
Removal useless production	✓	✓	x	x	x	x	x	x
Leftmost derivation	✓	✓	x	x	x	x	x	✓
Leftmost derivation tree	✓	✓	x	x	x	x	x	✓
String check	✓	✓	x	x	x	✓	x	✓
CNF	✓	✓	x	x	x	x	x	x
Removal left recursion	✓	x	x	x	x	x	x	x
GNF	✓	x	x	x	x	x	x	x
CYK parse	✓	✓	x	x	x	x	x	x
Ambiguity check	✓	x	x	x	x	x	x	x
Mobile	✓	x	x	x	x	x	x	x
Educational environment	✓	✓*	x	✓*	✓*	x	✓*	✓*

stable version was released in 2009, but a new BETA version was released in 2015. JFLAP has a graphical editor for drawing automata, where it is possible to create several representations of automata (e.g., FSA, PDA, Multitape TM, Mealy, and Moore Machines), besides using other models to define languages as Regular Expression (REGEX) and grammars (RG and CFG). In addition to the mobile technology, FLApp contains some features—such as converting CFG to GNF and trying to identify ambiguity in a grammar—which JFLAP does not.

2.2 | FADL toolkit

Chakraborty et al. [3] have proposed a compiler technology based approach to model and simulate finite automata. To accomplish this approach, they defined the Finite Automaton Description Language (FADL) that formally models finite automata. They also created a toolkit to compile and interpret this language, in addition to tools to visualize and perform conversions in finite automata. In their toolkit, there are two compilation tools: one with and one without optimization. The only difference is the goal that the optimized tool minimizes the DFA. Among conversion tools, you can convert NFAs to DFAs and DFAs to MTs. Note that this approach is different from the FLApp one, besides the toolkit only implements FSA related features.

2.3 | JCT

Robinson et al. [12] have created a Java Applet for web browser that creates and edits automata through a graphic editor. It is possible to perform the entire set of FSA closure operations in the application. Therefore, this application performs simulations of finite automata and Turing machines. In comparison to their work, FLApp does not have the implementation of the set of FSA closure operations, although FLApp is *not* limited to FSA and TM models.

2.4 | Language emulator

Vieira et al. [16] have proposed Language Emulator, an environment focused on Regular Languages (RL) that supports the teaching of FLA. It implements algorithms on REGEX, Regular Grammar (RG), FSAs, and Mealy and Moore machines. The tool is complete with regard to the class of RL, implementing the techniques and algorithms of that level. However, in contrast to FLApp, it does not have implementation of models referring to other classes of languages.

2.5 | SCTMF

Yandre et al. [5] have proposed the SCTMF (Software for the Creation and Testing of Formal Models) tool with the purpose of assisting the teaching of FLA. This tool consists of the creation

and simulation of formal models in acceptor machines from Chomsky Hierarchy, such as FSA, PDA, and TM. The main focus of the tool is on accepting machines, however, different from FLApp, this tool does not implement conversions between machine models, besides having no grammars features.

2.6 | GAM

Jukemura et al. [9] have described GAM, a tool that simulates finite automata to support the teaching of Theory of Computation. It implements the simulation of FSA, conversion from NFA to DFA, conversion from DFA to REGEX, and conversion from REGEX to NFA- λ . Different from FLApp, this tool implements only models and features of the class of Regular Languages.

2.7 | Webworks applets

Grinder et al. [8] have proposed three web-learning applets that enhance Montana State University's student experience in computer theory learning. These learning applets on the web involve features related to the respective areas of theory of computation: finite state automata, context-free grammar, and regular expressions. In these applets, students solve exercises with access to explanations and practice the construction and simulation of FSA, CFG, and REGEX, among other features. In FLApp, there are no exercises with explanations, but there are explanations for every given input. Moreover, FLApp is *not* limited to the FSA, CFG, and REGEX models.

3 | THE PROPOSED ACADEMIC SUPPORTING TOOL

This section introduces FLApp, a mobile application that covers most content of the Formal Languages and Automata course. Our goal is not only to solve the underlying course problems but also to provide students with the detailed step-by-step of the solution. In a nutshell, users write a grammar or draw an abstract machine, and then they can perform algorithms related to such grammar or machine.

The next subsections describe and illustrate the features by the language level in the Chomsky hierarchy, except for Section 3.1 that describes and illustrates common features. A [G] after the feature name means it refers to grammar features, whereas [M] means it refers to abstract machine features.

3.1 | Common features

3.1.1 | Grammar type identification [G]

This feature identifies at which level of Chomsky hierarchy the grammar belongs. Basically, the algorithm analyzes the

format of grammar rules and classifies it into the most restricted possible level. To evince that that the outputted level is the most restricted, FLApp shows an example of a rule that does not belong to the definition of the higher level.

3.1.2 | Leftmost derivation tree [G]

This feature performs the leftmost derivation process, that is, attempts to generate a string from a grammar. When the given string belongs to the grammar, FLApp displays its derivation tree. In case it finds two different derivation trees, FLApp alerts ambiguity.

3.1.3 | Ambiguity check [G]

This feature searches for ambiguity in a grammar. Since existing algorithms may not halt for unambiguous grammars, the user can limit the time of searching. Using the method of Gorn [6], we generate all possible derivations from the initial symbol. If some derivation results in a previous found string, FLApp acknowledges the ambiguity and displays the two derivations for this same word. Otherwise, when time runs out, FLApp reports that it was not able to find ambiguity in that time frame.

3.2 | Features of regular languages

Features of Regular Languages include: to generate complete Deterministic Finite-State Automaton (DFA), to generate Non-Deterministic Finite-State Automaton (NFA) with Regular Expression, to convert NFA to DFA, and to minimize DFA.

3.2.1 | To generate complete DFA [M]

This feature adds an “error” state in an incomplete DFA to convert it into a complete DFA. An incomplete DFA has a partial transition function, whereas a complete one has a total transition function. To complete a DFA, we add an error state, where all missing transitions go to this state, and all transitions from this state go to itself.

3.2.2 | To generate NFA with regular expression [M]

The user provides a regular expression RE and this feature generates a NFA M where the language accepted by M is equal to the language described by RE ($L(M) = L(RE)$). FLApp relies on Thompson's construction algorithm, which determines how to convert the operations of regular sets (union, concatenation, and Kleene star) to NFA.

3.2.3 | NFA to DFA conversion [M]

This feature converts a NFA into an equivalent DFA. First, FLApp analyzes if the NFA has lambda transitions; if confirmed, it converts the NFA- λ to NFA by calculating the λ -closure (i.e., all reachable states consuming only the empty string) of the states. Therefore, the states the λ -closure of a state s reaches consuming a symbol is now an NFA transition from s . Converting NFA to DFA removes non-determinism by converting the set of states that a transition could reach to only one state in DFA. A transition $\delta(q_0, a) = \{q_1, q_2\}$ in a NFA is converted to $\delta(q_0, a) = \langle q_1, q_2 \rangle$ in the equivalent DFA. In the Educational environment of this feature, the user can step back and forth on the conversion process.

3.2.4 | DFA minimization [M]

Given a DFA D, this feature creates a DFA D', where D' is the smallest possible DFA, such that $L(D) = L(D')$. To minimize a DFA, it is necessary to find the equivalent pairs of states and merge them into a single state. We first establish that all pairs of states are equal and we try to find information that indicates that these pairs of states are distinct. Thereupon, the pairs we cannot differentiate are considered equals. As an illustrative example, Figure 1a shows a DFA to be minimized and Figure 1c outputs the minimized DFA. Again, in the Educational environment of this feature, the user can step back and forth the algorithm execution, as can be seen in Figure 1b.

3.2.5 | Simulation of FSA [M]

The user creates a FSA using the editor and simulates it for an input string. The simulation is demonstrated by the FSA configurations until the acceptance or rejection of the input string. The FSA configurations are demonstrated by the input string, a state, and the position of read head, similarly as further illustrated on the right side of Figures 3 and 4.

3.3 | Features of context-free languages

Features of Context-Free Languages includes: removal of recursion of the start symbol, removal of empty productions, removal of chain rules, removal of variables that do not produce terminals, removal of unreachable symbols, Chomsky Normal Form, removal of direct left recursion, removal of indirect left recursion, Greibach Normal Form, CYK (Cocke-Younger-Kasami) recognition algorithm, Context-Free Grammar (CFG) to Pushdown Automaton (PDA) conversion, simulation of PDA, and PDA to CFG conversion. Although most of these algorithms are classic,



FIGURE 1 DFA minimization feature. User inputs the DFA to be minimized (a), FLApp executes the algorithm step-by-step (b) and outputs the minimized DFA (c)

we implemented our features based on the algorithms described in [14].

3.3.1 | Elimination of recursion of the start symbol [G]

When the grammar has a derivation $S \Rightarrow^* \alpha S \beta$, this feature creates a new non-terminal start symbol S' and inserts a new rule $S' \rightarrow S$, which removes the recursion in the start symbol. Thus, start symbol is limited to initiate derivations.

3.3.2 | Elimination of lambda rules [G]

This feature removes empty productions in a CFG. To accomplish this conversion, we implemented the λ -rules algorithm, which makes grammar essentially non-contractile (or even non-contractile).

3.3.3 | Elimination of chain rules [G]

Rules like $A \rightarrow B$, which are called chain rules, do not directly contribute to form a string and must be removed by an algorithm that locates and replaces them with the productions referring to the chain destination variable.

3.3.4 | Remove variables that do not generate terminal strings [G]

This feature removes usefulness variables that do not contribute to the generation of strings. It calculates the TERM set that contains the variables that generate terminal strings directly or indirectly, and removes the variables that are not in TERM.

3.3.5 | Removal of unreachable symbols [G]

This feature removes usefulness variables that are not reached from the start symbol, that is, those variables that never appear in a derivation $S \Rightarrow^* \alpha A \beta$. It calculates the REACH set that contains the reachable variables directly or indirectly from the start symbol, and removes the variables that are not in REACH.

3.3.6 | Chomsky normal form [G]

This feature standardizes the rules of a grammar in one of the CNF standards ($A \rightarrow BC$, $A \rightarrow a$, and $A \rightarrow \lambda$). Basically, it replaces rules that are not in the CNF standards with new rules. It is worth noting that this feature requires the input grammar to have no recursive start symbol, empty productions, chain rules, and useless symbols (variables that do not generate terminal strings and unreachable symbols); otherwise FLApp applies those algorithms in advance.

Figure 2 provides an illustrative toy example from the grammar input and feature selection (Figure 2a,b), through the required pre-transformations (Figure 2c–g), until the resulted grammar in the CNF (Figure 2h). Although FLApp provides the detailed step-by-step of every feature, we did not omit only the one for the CNF feature, as can be observed in Figure 2h.

3.3.7 | Greibach normal form [G]

This feature standardizes the rules of a grammar in one format of the GNF standards ($A \rightarrow aA_1A_2 \dots A_n$, $A \rightarrow a$, and $A \rightarrow \lambda$) to ensure that there will be no direct or indirect left recursions. Basically, it replaces rules that are not in the GNF standards with new rules. It is worth noting that this feature requires the input grammar to already be in the CNF, otherwise FLApp automatically performs the previous described feature in advance.

3.3.8 | CYK recognition algorithm [G]

This feature checks if a given CFG can generate a given string. The CYK algorithm uses a bottom-up approach to analyze the sentence by associating it with the rules of the grammar. If the start symbol is at the top of the CYK matrix, the string belongs to the grammar; otherwise it does not.

3.3.9 | CFG to PDA conversion [G/M]

This feature converts a CFG G into a PDA M , such that $L(G) = L(M)$. Since this feature requires the CFG G to be in the

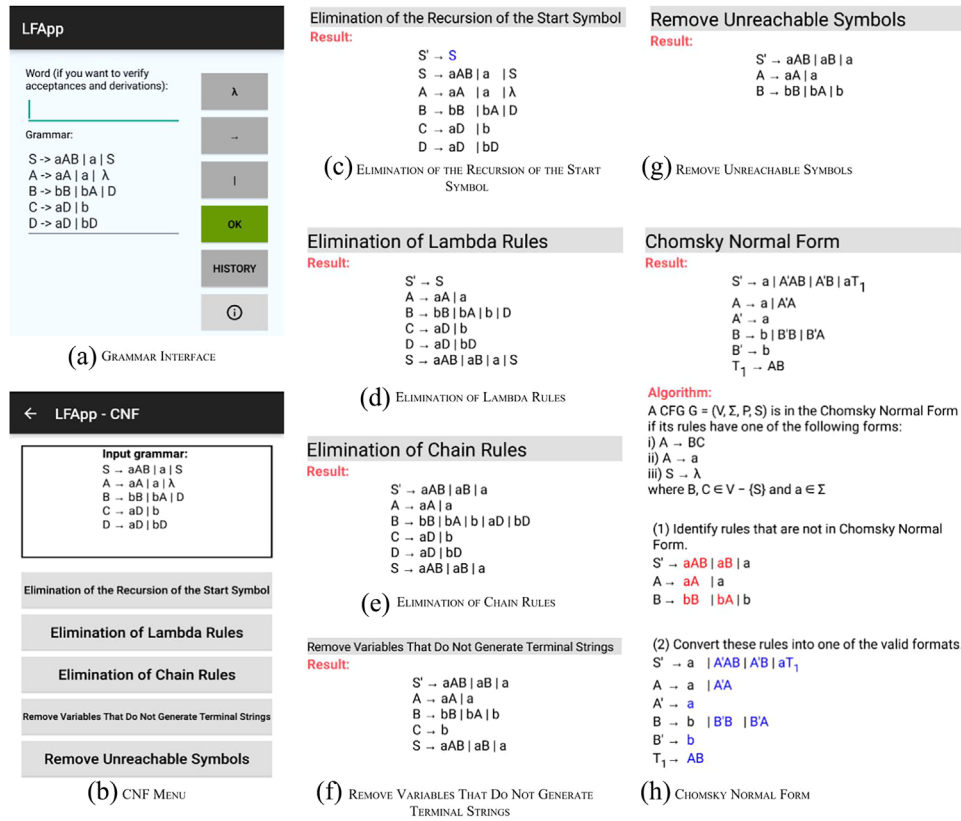


FIGURE 2 Chomsky normal form feature. User inputs the grammar (a), he/she chooses CNF feature from the menu (b), and FLApp executes five required features sequentially (c-g) until it provides the resulted grammar in the CNF form (h)

GNF, FLApp may automatically perform the GNF feature in advance. Next, it creates an extended PDA with only two states q_0 and q_1 , where q_0 is the start and q_1 is the accepting states. For each rule of G , we must create a transition in the *extended* PDA in which it reads the terminal rule, pops the variable rule, and pushes the remainder on the right side of the rule. All transitions are from and to state q_1 , except for those obtained by rules of the start symbol that do not pop and leave from the state q_0 .

3.3.10 | Simulation of PDA [M]

The user creates a PDA using the editor and simulates it for an input string. The simulation is demonstrated by the PDA configurations until the acceptance or rejection of the input string. PDA configurations are demonstrated by the input string, its stack, a state, and the position of read head. In case of non-determinism, we try to find a stack of configurations that leads to an acceptance, which is limited by a stack size of 100 configurations.

3.3.11 | PDA to CFG conversion [G/M]

This feature converts a PDA M into a CFG G , such that $L(M) = L(G)$. First, it creates an *extended* PDA M' where all transitions that do not pop, now stack and pop the same symbol. Next, the variables of G are defined by tuples of the

formula $\langle q_i, A, q_j \rangle$, and the rules are generated from the PDA transitions using one of the four following strategies:

- $S \rightarrow \langle q_0, \lambda, q_j \rangle, \forall q_j \in F$;
- If $[q_j, B] \in \delta'(q_i, x, A)$, then $\langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_k \rangle, \forall q_k \in Q$;
- If $[q_j, BA] \in \delta'(q_i, x, A)$, then $\langle q_i, A, q_k \rangle \rightarrow x \langle q_j, B, q_n \rangle \langle q_n, A, q_k \rangle, \forall q_n, q_k \in Q$;
- $\langle q_k, \lambda, q_k \rangle \rightarrow \lambda, \forall q_k \in Q$

3.4 | Features of context-sensitive languages

The unique feature exclusive to Context-Sensitive Languages is the simulation of Linear Bounded Automaton (LBA).

3.4.1 | Simulation of LBA [M]

The user creates a LBA using the editor and simulates it for an input string. The simulation is demonstrated by the LBA configurations until the acceptance or rejection of the input string. LBA configurations are demonstrated by its tape with its memory delimiters, a state, and the position of read/write head. Figure 3 illustrates an example where the user draws a LBA that recognizes language $L = \{a^i b^j c^i \mid i > 0\}$ and simulates it with the input “abc”.

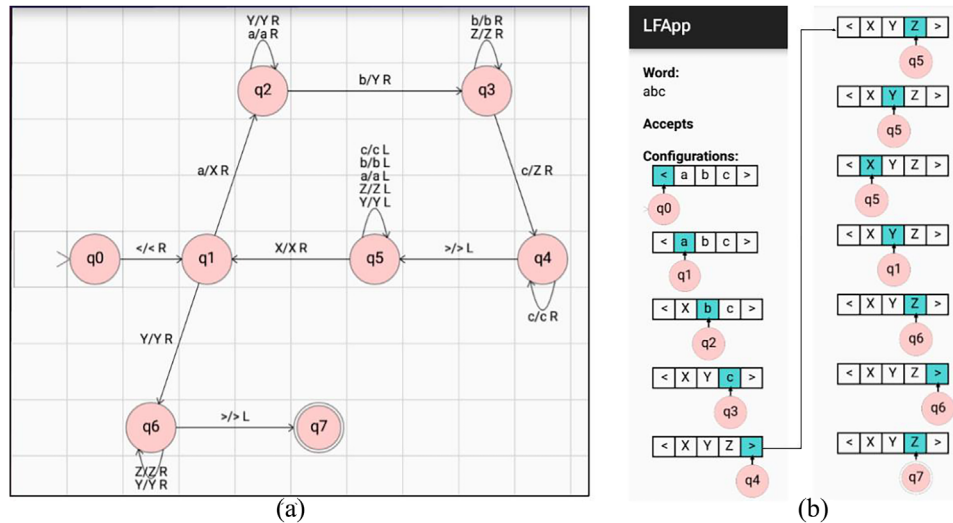


FIGURE 3 LBA simulation feature. User inputs the LBA to be simulated (a) and FLApp outputs the configurations until the acceptance or rejection of a given string (b)

3.5 | Features of recursively enumerable languages

Features of Recursively Enumerable Languages include: simulation of Turing Machine (TM), Multitrack TM, Multitape TM, and TM as Language Enumerators.

3.5.1 | Simulation of turing machine [M]

The user creates a TM using the editor and simulates it for an input string. The simulation is demonstrated by the TM configurations until the acceptance or rejection of the input string. TM configurations are demonstrated by its infinite tape, a state, and the position of read/write head.

3.5.2 | Simulation of multitrack TM [M]

It is similar to the simulation of TM feature, but the Multitrack TM configurations are demonstrated by n infinite tracks, a state, and the position of read/write head.

3.5.3 | Simulation of multitape TM [M]

It is also similar to the simulation of TM feature, but the Multitape TM configurations are demonstrated by n infinite tapes, a state, and the positions of n read/write heads, one in each tape.

3.5.4 | Simulation of TM as language enumerators [M]

The user creates a Multitape TM as Language Enumerator using the editor and this feature provides the configuration

of TM afterwards a new enumerated string. Figure 4a illustrates a TM as Language Enumerator that enumerates language $L = \{a^i b^i \mid i \geq 0\}$ and Figure 4b illustrates the output of FLApp after each new enumerated string.

4 | FLApp DESIGN AND IMPLEMENTATION

FLApp is an educational resource, being publicly available since the beginning of its development in a repository in GitHub.¹ Basically, the app is divided in three modules: `core`, where are the entities and classes that implement the underlying algorithms; `grammar`, where are classes specific to grammar features, and `machine`, where are classes specified to machine features.

4.1 | Core

It contains classes that parse grammars (e.g., class `TextualParser`), verify ambiguity (class `AmbiguityVerification`), and implement general purposes functionalities.

4.2 | Grammar

The main classes are `Grammar` and `Rule`. `Grammar` represents a grammar of any level of the Chomsky Hierarchy, which is composed of a set of `Rules`, which represents a rule of type $\mu \rightarrow \nu$.

¹<https://github.com/rterrabh/FLApp>

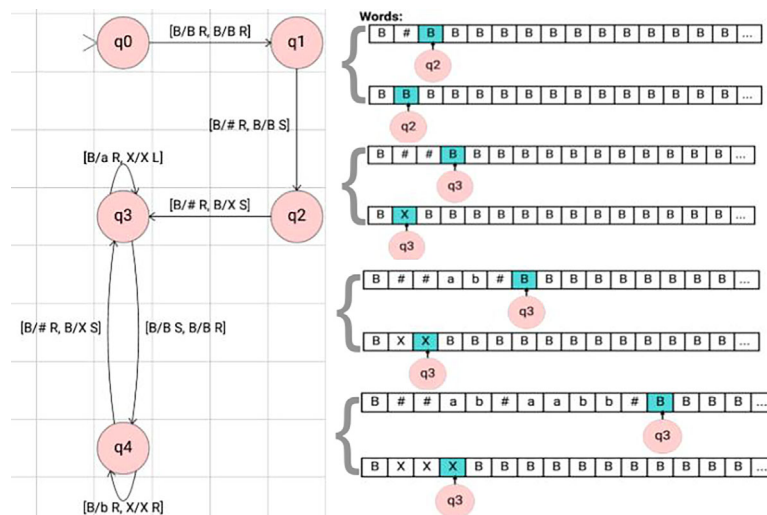


FIGURE 4 Simulation of TM as language enumerator feature

4.3 | Machine

There are a bunch of classes, but we can highlight the following interfaces: *Machine*, which represents an abstract machine; *State*, which defines a state in an abstract machine; *TransitionFunction*, which represents a transition function; and *Configuration*, which represents a configuration of an abstract machine. The concrete machine properly realizes these interfaces, for example, Finite State Automaton has a concrete implementation of these interfaces inside package `fsa`. It is worth noting that, for extensibility and flexibility purposes, there is a `dotlang` package since FLApp stores and reads machines in a well-known and widely used dot graph language.

5 | THE TOOL EVALUATION

To evaluate FLApp, we conducted a correctness and didactical evaluation. The former guarantees the correctness of the implemented functionalities and the latter guarantees the Educational environment of the functionalities.

5.1 | Correctness evaluation

This evaluation verifies—through unit tests [11]—whether the results are exactly what users expect. Our goal is to achieve the correctness that FLApp requires to be widely employed in universities. We simulate 595 different inputs where 398 tests are related to grammars and 197 to machines. These tests cover 90% of the source code, which implicates in a high level of correctness of the application we are proposing.

5.2 | Didactical evaluation

This evaluation aims to demonstrate the adequacy of the Educational environment, which describes and illustrates each step of the algorithms to support students in the learning process. Thus, we rely on a well-known course book [14], which is widely employed in Formal Languages and Automata courses. As reported in Table 2, for every example the book provides, we verified whether the steps FLApp presents to the user are equivalent to those in the book. As result, FLApp provides the same output for the 37 examples of the book besides providing equivalent step-by-step solution.

6 | THE APP AND ITS EDUCATIONAL GAINS: A CASE STUDY

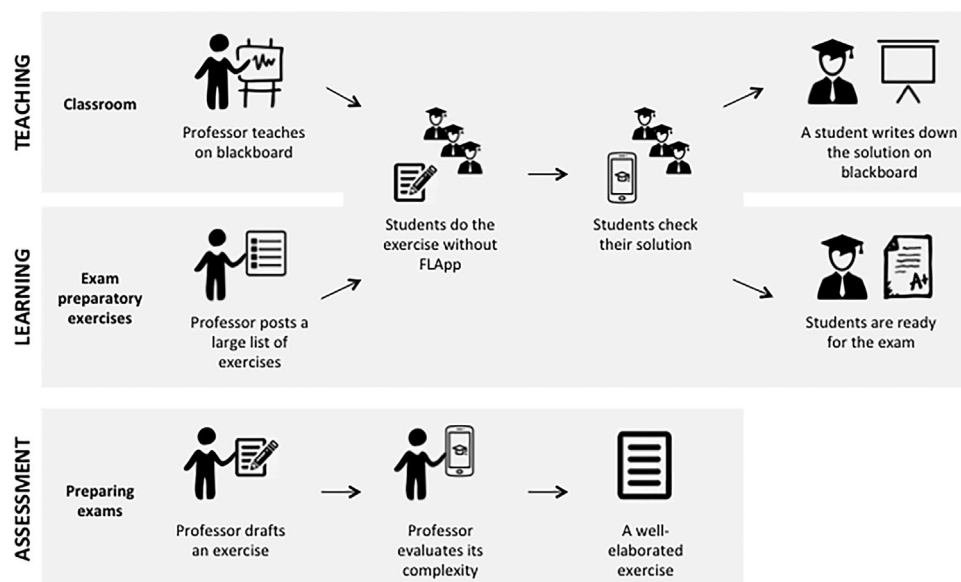
This section reports how FLApp has been practically integrated in the FLA classes of a Brazilian university and what was the outcome. Since 2016, the professor (and also the second author of this paper) has been employing FLApp as part of his teaching classes. Since the algorithms and techniques covered in the course are complex and difficult to assimilate, he promotes dynamism on his classes to keep the students motivated. As illustrated in Figure 5, FLApp has been properly integrated into three main educational activities, namely teaching, learning, and assessment, as follows:

6.1 | Teaching

The professor teaches the algorithms and techniques, giving one or two examples. Next, he writes down up to three exercises and instructs the students to check the solution on FLApp *only* after they finish the exercise.

TABLE 2 Result of the didactical evaluation

Feature	Page	Unit test	Result	Step-by-step	Feature	Page	Unit test	Result	Step-by-step
Leftmost derivation and ambiguity	91	Sudkamp351Test	✓	✓	NFA- λ simulator	166	Sudkamp551Test	✓	✓
	92	Sudkamp352Test	✓	✓	Removing nondeterminism	171	Sudkamp561Test	✓	✓
Elimination recursion of start symbol	105	Sudkamp411Test	✓	✓		173	SudkampF54Test	✓	✓
Elimination of λ -rules	109	Sudkamp421Test	✓	✓		174	Sudkamp562Test	✓	✓
	113	Sudkamp423Test	✓	✓		175	Sudkamp563Test	✓	✓
Elimination of chain rules	115	Sudkamp431Test	✓	✓		176	Sudkamp564Test	✓	✓
Useless symbols—TERM	118	Sudkamp441Test	✓	✓	DFA minimization	181	Sudkamp571Test	✓	✓
Useless symbols—REACH	120	Sudkamp442Test	✓	✓		182	Sudkamp572Test	✓	✓
Useless symbols	121	Sudkamp443Test	✓	✓	Create FSA from Regex	192	Sudkamp611Test	✓	✓
Chomsky normal form	122	Sudkamp451Test	✓	✓	PDA simulator	225	Sudkamp711Test	✓	✓
	124	Sudkamp452Test	✓	✓		231	Sudkamp722Test	✓	✓
CYK	126	SudkampT41Test	✓	✓	CFG to PDA	232	Sudkamp730Test	✓	✓
Removal of direct left recursion	130	Sudkamp471Test	✓	✓	PDA to CFG	235	Sudkamp731Test	✓	✓
Greibach normal form	132	Sudkamp480Test	✓	✓	TM simulator	257	Sudkamp811Test	✓	✓
	136	Sudkamp481Test	✓	✓		260	Sudkamp812Test	✓	✓
FSA simulator	149	SudkampT52Test	✓	✓	TM multitape simulator	269	Sudkamp861Test	✓	✓
	150	Sudkamp521Test	✓	✓	TM nondeterministic simulator	275	Sudkamp871Test	✓	✓
Complete DFA	158	Sudkamp539Test	✓	✓	TM as language enumerator simulator	283	Sudkamp881Test	✓	✓
NFA simulator	164	Sudkamp541Test	✓	✓					

**FIGURE 5** Educational application of FLApp

Meanwhile, he assigns each exercise to a randomly selected student who should write his/her solution on the blackboard.

In this context, we noted two benefits of FLApp: (i) since students can check not only their answers but also their steps towards it, they try to understand by themselves where they made a mistake and their doubts are much more specific; and (ii) FLApp was extremely helpful mainly to those students who are afraid to publicly write down incorrect solutions, that is, FLApp ensures that their solution is correct; otherwise they can see where they fail and correct it before.

6.2 | Learning

The course has three exams (RL; CFL; and CSL and REL). The professor provides three long lists of preparatory exercises for each exam. He again instructs the students to check a solution on FLApp *only* after they finish that exercise. Here, we claim the major benefit of FLApp for students because they can—anywhere—check not only their answers but also their steps towards it. According to the Teacher's Assistant (TA) of this course, FLApp reduced the number of students who demand for tutoring because students try even harder to understand by themselves where and why they made a mistake.

6.3 | Assessment

FLApp supports the task of preparing exercises. The professor writes a draft of the exercise and FLApp provides its solution. While the professor finds the solution too easy or too hard, he modifies the exercise and FLApp again provides the solution. For an actual example, the transformation of an arbitrary grammar to the Chomsky Normal Form requires up to five pre-transformations. The professor wants an exercise that requires the students to make these five transformations. The professor can modify the exercise up to reach the complexity he wants.

7 | CONCLUSION

Formal Languages and Automata (FLA) is an important area of Computer Science that approaches mathematical models able to specify and recognize languages, their properties and characteristics. However, the algorithms and techniques covered in FLA are complex and difficult to assimilate, complicating the learning process.

To facilitate and make the learning more interesting, our study proposes FLApp, a mobile application—what we consider the new way to reach students—with an Educational environment that describes and illustrates each step of the execution of algorithms. FLApp


implements features involving all levels of Chomsky hierarchy, and have an environment for creating/editing abstract models that represents languages for all levels.

As the main contributions of FLApp, we (i) provide an interactive learning environment for students to assist in the FLA teaching and learning process and (ii) promote its usage through the mobile technology since, as reported in Section 6, several students do not carry their laptops and could benefit of the app during the classes.

ACKNOWLEDGMENT

Our research is supported by FAPEMIG.

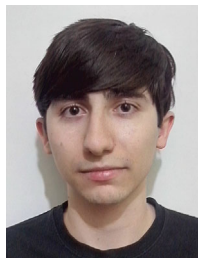
ORCID

Ricardo Terra  <http://orcid.org/0000-0002-5824-7087>

REFERENCES

1. A. V. Aho et al., *Compilers: Principles, techniques, and tools*, 2nd ed., Addison-Wesley, 2006.
2. P. Chakraborty, P. C. Saxena, and C. P. Katti, *Automata simulators: Classic tools for computer science education*, Brit. J. Educ. Technol. **43** (2012), E11–E13.
3. P. Chakraborty, P. C. Saxena, and C. P. Katti, *A compiler-based toolkit to teach and learn finite automata*, Comput. Appl. Eng. Educ. **21** (2013), 467–474.
4. N. Chomsky, *Three models for the description of language*, IRE Trans. Inf. Theory **2** (1956), 113–124.
5. Y. M. e G. da Costa, R. C. de Meneses, and F. R. Uber, *Uma ferramenta para auxílio didático no ensino de Teoria da Computação*. In XVI Workshop sobre Educação em Computação (WEI), pages 208–217, 2008.
6. S. Gorn, *Detection of generative ambiguities in context-free mechanical languages*, J. ACM **10**, (1963), 196–208.
7. E. Gramond and S. H. Rodger, *Using JFLAP to interact with theorems in automata theory*, ACM SIGCSE Bulletin **31** (1999), 336–340.
8. M. T. Grinder et al., *Loving to learn theory: Active learning modules for the theory of computing*, ACM SIGCSE Bulletin **34** (2002), 371–375.
9. A. S. Jukemura, H. A. D. do Nascimento, and J. Q. Uchôa, *Uchôa. GAM: Um simulador para auxiliar o ensino de Linguagens Formais e de Autômatos*. XIII Workshop sobre Educação em Computação (WEI), pages 2432–2443, 2005.
10. F. Manjoo, *A murky road ahead for Android, despite market dominance*. The New York Times, 2015.
11. R. S. Pressman, *Software engineering: A practitioner's approach*, 7th ed., McGraw-Hill, 2009.
12. M. B. Robinson et al., *Java-based tool for reasoning about models of computation through simulating finite automata and turing machines*, ACM SIGCSE Bulletin **31** (1999), 105–109.
13. R. Simpson. *Mobile and tablet Internet usage exceeds desktop for first time worldwide*. Statcounter, 2016.

14. T. A. Sudkamp, *Languages and machines: An introduction to the theory of Computer Science*, 3rd ed., Addison-Wesley, 2005.
15. R. Terra, Data of the discipline Formal Languages and Automata. Technical report, Universidade Federal de Lavras (UFLA), 2016.
16. L. F. M. Vieira, M. A. M. Vieira, and N. J. Vieira, Language emulator, a helpful toolkit in the learning process of computer theory. In 35th Technical Symposium on Computer Science Education (SIGCSE), pages 135–139, 2004.



C. H. PEREIRA is an undergraduate student of Computer Science at Federal University of Lavras, Brazil. He worked on an undergraduate research project supported by FAPESP, working on FLApp, a mobile application for teaching Formal Languages, and Automata. Currently, he

works on another undergraduate research project supported by CNPq, researching about techniques to recover architecture of a software system.



R. TERRA received his PhD degree in Computer Science from Federal University of Minas Gerais, Brazil (2013) with a 1-year internship at the University of Waterloo, Canada. Since 2014, he is an assistant professor in the Department of Computer Science at Federal University of Lavras, Brazil. His re-

search interests include software engineering, and computer science education.

How to cite this article: Pereira CH, Terra R. A mobile app for teaching formal languages and automata. *Comput Appl Eng Educ*. 2018;26: 1742–1752. <https://doi.org/10.1002/cae.21944>