UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Ingeniería

https://revistas.udistrital.edu.co/index.php/reving/issue/view/1185

DOI: https://doi.org/10.14483/23448393.20626

Research

Application of Regular Grammar in the Syntactic Analysis of Email Addresses

Aplicación de gramática regular en el análisis sintáctico de direcciones de correo electrónico

Cristian Alejandro Fandiño-Mesa[1], Marco Javier Suárez-Barón[2]*, and César Augusto Jaramillo-Acevedo[3]

[1]Universidad Pedagógica y Tecnológica de Colombia (Sogamoso, Colombia)
[2]Universidad Pedagógica y Tecnológica de Colombia (Sogamoso, Colombia)
[3]Universidad Tecnológica de Pereira (Pereira, Colombia)

Abstract

Context: This article proposes the use of regular grammar as a strategy to validate the textual structures of emails. It focuses on the RFC 5321 standard and its syntax, formalizing regular grammars to apply production rules with the aim of validating the syntactic context of each structure of an email address.
Method: This article presents a literature review and the development of an email validation model. Related texts focus on the Internet Protocol, along with building automata that apply IPV4 protocol. There are three phases: the development of the model from syntax and regular grammar rules and its construction and application.
Results: The result is a functional application that validates email addresses based on regular grammars and existing regulations. When running efficiency tests, our application obtained a higher email validation margin in comparison with JFLAP. The library can work as a great analyzer of grammatical or lexical structures.
Conclusions: The email validation tool based on GR regular grammars contributes to the practical use of specialized algorithms in the field of computer science, since it is possible to apply it to the recognition of search patterns such as the analysis of lexical structures (e.g., NITs, alphanumeric codes, and valid URLs).

Keywords: email validation, formal grammar, regular expressions

Article history

Received:
23rd/Mar/2022

Modified:
23rd/Nov/2023

Accepted:
11th/Aug/2023

Ing, vol. 28, no. 3, 2023. e20626

*Correspondence: marco.suarez@uptc.edu.co

Resumen

Contexto: En este artículo se propone el uso de la gramática regular como estrategia para validar las estructuras textuales de los correos electrónicos. El estudio se enfoca en el estándar RFC 5321 y su sintaxis, formalizando gramáticas regulares para aplicar reglas de producción en aras de validar el contexto sintáctico de cada estructura de una dirección de correo electrónico.
Método: Este artículo presenta una revisión de la literatura y el desarrollo de un modelo para la validación de correos electrónicos. Los textos relacionados se enfocan en el Protocolo de Internet junto con la construcción de autómatas que utilizan direccionamiento IPV4. El estudio tiene tres fases: el desarrollo del modelo a partir de la sintaxis y reglas gramaticales regulares y la construcción y aplicación del mismo.
Resultados: El resultado es una aplicación funcional que válida direcciones de correo electrónico con base en gramáticas regulares y la reglamentación existente. Al ejecutar pruebas de eficiencia, nuestra aplicación obtuvo un margen de validación de correos mayor en comparación con JFLAP. La librería puede funcionar como un gran analizador de estructuras gramaticales o léxicas.
Conclusiones: La herramienta de validación de correos electrónicos basada en gramáticas regulares GR contribuye al uso práctico de algoritmos especializados en la rama de la computación, dado que es posible aplicarla en el reconocimiento de patrones de búsqueda como el análisis de estructuras léxicas (*e.g.*, NITs, códigos alfanuméricos, y URL válidas).

Palabras clave: validación de correos electrónicos, gramática formal, expresiones regulares

Table of contents

1. Introduction

The Finite State Machine (FSM), the stack automaton, and Regular Grammar (RG) have undergone extensive study in the context of computation theory and application in formal verification, formal analysis, software design, and natural language processing (1). The group of theory-based computing methods for the automatic analysis and representation of human languages is known as Natural Language Processing (NLP). However, in order for text to be automatically analyzed at the same pace as humans, there is still a long way to go, especially knowledge of natural language process PLN (2).

In the 1950s, Noam Chomsky began his study of natural languages, whose objective was to formally define their syntax (3). To this effect, Chomsky introduced generative grammar. Later, it was discovered that the syntax of programming languages can be described Chomsky's grammatical models (context-free grammars). Formal grammar is defined as the set of rules by which the chains of a language are constructed, that is, a grammar is a set of rules that describe a language. These rules are called *production rules* (3), and they describe how to form valid strings from the alphabet of the language.

Language recognition and the establishment of the rules for regular grammar enabled the advent of the stack automaton, oriented towards applications in the validation of emails. This automaton analyzes both basic address structures and those containing special characters. It works in line with the RFC 5321 and 5322 norms (4), as per the recommendations of the Internet Engineering Task Force (IETF). As some issues are currently solved thanks to the validation and detection of spam fraud, a list of valid and invalid addresses ensures functionalities for companies that handle mailing.

Most of the texts related to FSM focus on Internet Protocol (IP) resources to deal with identity theft and network operators. In (5) is proposed a finite automaton scheme that accepts the IPV4 class of addresses, providing the descriptions of the languages together with the IPs and their proper construction for the automaton. The states and transitions that comprise it are shown in said work, which discusses the design of an automaton that accepts the class of IP addresses so that they accept the class in IPV4. In (6), inconsistencies are shown which can allow attackers to bypass email authentication to impersonate arbitrary senders and forge emails signed by Domain Keys Identified Mail (DKIM) with the signature of a legitimate site.

The process includes verifying the structure of an email address based on currently valid regulations, identifying the indicated syntax and explaining the operation of DKIM. By performing various tests with bad addresses that are used for spamming and injection attacks, including cross-site scripting (XSS), SQL injection, and Remote Code Execution (RCE) or spoofing, the possible security flaws of organization when acquiring IT services are evidenced.

The problem with analyzing the structures of emails has to do with syntactic validation, which includes an analysis of the syntax that ensures the validity of an email, while semantic validation could allow only analyzing the addresses of specific domains.

This work is carried out through a review of specialized scientific literature and the construction of RG focused on the syntactic rules of email addresses. With the help of the JFLAP tool, the response of the automaton is validated. After several tests, both valid and invalid addresses are entered, verifying that each of these chains is fulfilled. Then, the algorithm is built, which consists of two parts: the logic of the class to perform validations and the Graphical User Interface (GUI). This algorithm is once again verified with several email addresses. Finally, the distribution of the library is prepared in a repository for third-party applications (Pypi) for later use. Using this library and platforms such as JFLAP, the response time when entering a number of emails from certain addresses or their validity can be checked and compared with accuracy, covariance and precision.

This work is organized as follows. It begins with a description of the materials, methods and methodology used in the design and validation of the proposed algorithm. This is followed by a review of the specialized literature on regular grammar along with its formal definition. Tables of RG rules are shown, along with the result obtained with JFLAP, ending with the tests conducted on the algorithm in specific scenarios, together with the application and subsequent installation of the designed library. Finally, the conclusions and proposals for future work are presented.

2. Subject development

2.1. Methodology

For email validation from RG, stack automata and Python programming were used. The theory involved is based on (1, 2, 7). In addition, the RFC 5321 and 5322 norms were reviewed, which propose a structure of mail forms. These regulations are the basis of the validation of the rules regarding regular grammar, and their objective is to ensure not only that the most common emails work but also those that possess the characteristics outlined in the regulations. During this process, the JFLAP platform and the PyCharm Community IDE were used to display a sketch of the automaton to be programmed.

The methodology applied in this work is depicted in Fig. **1**. It is observed that formal grammar reads the input string one symbol at a time (7), verifying that the chain complies with RFC standards, thus allowing to identify the entry of the mail in its two divisions (i.e., the local part and the domain part) since the grammar rules of each one have already been defined.

2.1.1. Formal definition of stack automata

A *stack automaton* is a finite automaton equipped with stack-based memory. Each transition is based on the current input symbol and on top of the stack (3). It optionally removes the top of the stack and optionally inserts new symbols in it. Initially, the stack contains a special symbol $Z0$ that indicates the bottom of the stack.

It is defined as $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where:

- Q is a set of states
- Σ is the alphabet of input symbols
- Γ is an alphabet of stack symbols
- δ is a transition function used to move from the current state to the next one

The transition function is described in Eq. (1), understanding that the element $(p, a, A, q, \alpha) \in \delta$ is a transition from M. This means that M in the state $p \in Q$ at the entrance a $\in \Sigma \cup \{\varepsilon\}$ and with $A \in \Gamma$. As a top battery symbol, it can be read a, change the state q, delete A, replacing it with the thrust a $\in \Gamma^*$. The component of the transitional relationship is given by $\Sigma \cup \{\varepsilon\}$. The stack automaton allows access and operations in deeper elements. Stack automata can recognize a strictly larger set of languages than insert automata (1). A stack automaton allows full access and makes it easy for stacked values to be

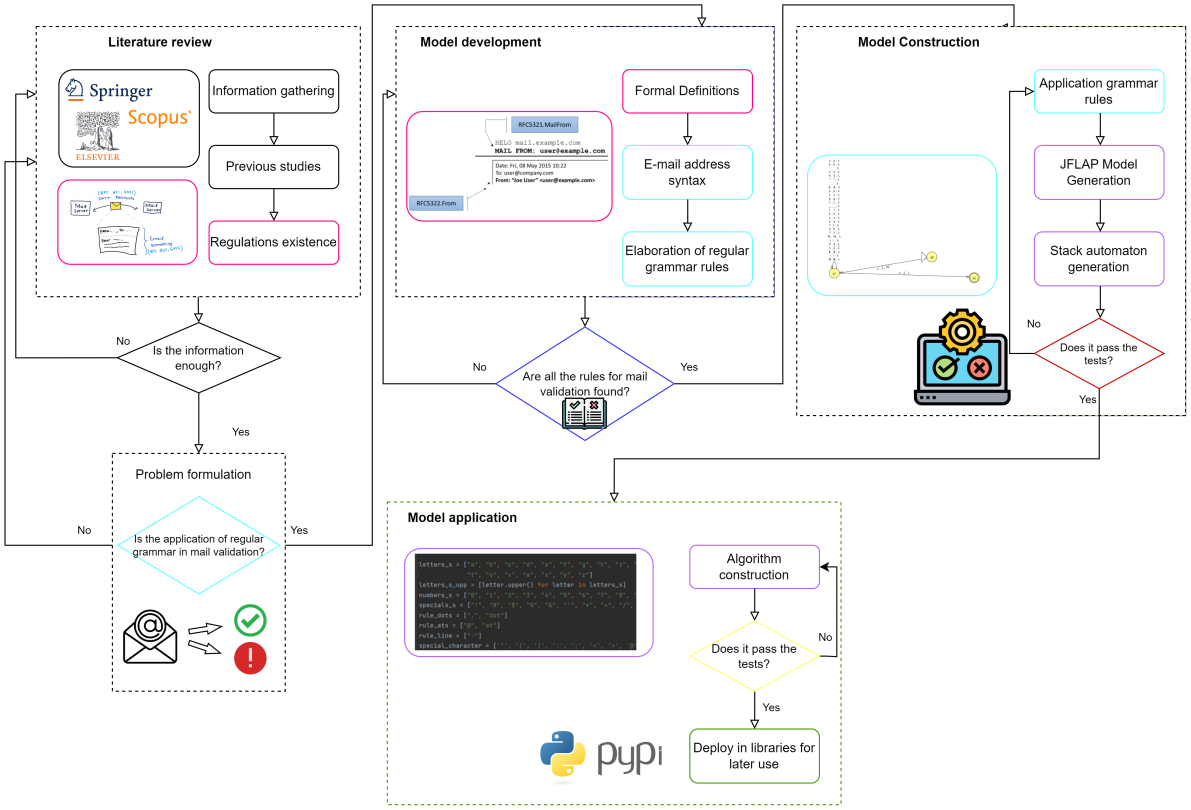


Figure 1. Diagram of the grammar analysis process

complete sub-stacks rather than a set of unique finite symbols. These automata (related to pushdown automata) can step left or right in the input string (surrounded by special end symbols to prevent them from escaping) and move up or down the string in read-only mode (2). A stack automaton is said not to be erased if it never leaves the stack.

The turnstile notation is represented by the symbol $\vdash$, which represents a movement. For example, $(p, b, T) \vdash (q, w, a)$. This implies that, while a transition is made from state p to state q, the input symbol b is consumed, and the top of the stack T is replaced by a new string a.

The sign $\vdash^*$ represents a sequence of movements.

The language accepted by a stack automaton can be defined in two different and equivalent ways (2):

- *Acceptance by final state.* The AP is said to accept the input by the final state if it enters any final state at zero or more moves after reading the entire input.
- *Acceptance by empty stack.* When reading the input string of the initial configuration for some APs, the AP stack is emptied (7).

2.1.2. Formal definition of RGs

Regular grammar generates regular language. RGs have a single nonterminal on the left side and a right side consisting of a single terminal or a single terminal followed by a nonterminal. Formally, a grammar consists of a set of nonterminal (or variables) V, a set of terminals Σ (the language alphabet), a beginning symbol S, which is a nonterminal, and a set of rewriting rules (productions) P (7).

A right-hand regular grammar (also called a *right-hand linear grammar*) is a formal grammar (N, σ, P, S) in which all production rules in P have one of the following forms:

$$A \to aB \tag{1}$$

$$A \to a \tag{2}$$

$$A \to \varepsilon \tag{3}$$

where A, B, and $S \in N$ are nonterminal symbols, $a \in \Sigma$is a terminal symbol, and ε denotes the empty string, *i.e.*, the string of length 0. S is called the *start symbol* (8).

In a regular left grammar (also called a *left linear grammar*), all rules obey the following forms:

$$A \to Ba \tag{4}$$

The language described by a given grammar is the set of all strings that contain only terminal symbols and can be derived from the start symbol via the repeated application of production rules. These two types of grammars have the same generative power, *i.e.*, given a linear grammar on the left, there is always a linear grammar on the right that is equivalent to it, and *vice versa*. In addition, given a regular language, there is always (at least) a linear grammar on the left and a linear grammar on the right that generate it. There is no profound difference between left and right linear grammar.

A regular grammar is a quadruple (V, Σ, R, S), where:

- V is an alphabet of variables
- Σ is an alphabet of constants
- R, the set of rules, is a finite subset of $V \times (\Sigma V \cup \Sigma)$
- S, the initial symbol, is an element of V

This grammar, together with the set of rules, is determined under the elements of $V - \Sigma$, which are called *nonterminals* and are analogous to parts of speech. In the example, $G = (V, \Sigma, R, S)$ where $V = \{S, a, b\}, \Sigma = \{a, b\}$, and R has the rules $S \to aSb$ and $S \to \varepsilon$.

Nonterminals are generally represented by capital letters and terminals by lowercase letters. Therefore, a context-free grammar can be given simply by providing the rules and the starting symbol without a tuple of 4. The above-mentioned grammar can be represented as:

$$S \to aSb \tag{5}$$

$$S \rightarrow \varepsilon \tag{6}$$

Here is a derivation of this grammar.

$$S \implies aSb \implies aaSbb \implies aabb$$

Such a derivation has to eliminate the nonterminal. It is also possible to write this derivation as

$$S \implies {}^{*}aabb \tag{7}$$

Given a context-free grammar G, the language generated by $G, L(G)$, is the set of terminal strings that can be derived from the initial symbol for G.

2.1.3. Syntax

- Email syntax
  Each email address consists of three elements: the local part, the @ symbol (pronounced as ”at”), and the domain name. The local part is placed before the @ symbol, and the domain name is placed after it (9). An example of this is the email johndoe@company.com, where ” johndoe ” is the local part and ” company.com ” is the domain. Emails are not valid without these items.

  parte_local@dominio

- Local part
  The local part is the username that indicates a unique name for the mailbox. It can contain
  - Uppercase and lowercase Latin letters (AZ, az)
  - Numerical values (from 0 to 9)
  - Special characters, such as #! % $ ’& + * -/=? ^_‘. { | }

  It should be noted that the period character (.) Is valid for the local part unless it is placed at the beginning or the end of an email address (9). Moreover, there cannot be two or more periods in a row in the email address (*e.g.*, John.. Doe @ company.com is not allowed).

  The local postmaster part is treated in a special way: it is not case-sensitive and should be forwarded to the email manager of the domain (10). Technically, all other local parts are case-sensitive. Therefore, jsmith@example.com specifies different mailboxes, but many organizations treat uppercase and lowercase letters as equivalent. In fact, RFC 5321 warns that ”a host waiting to receive mail should avoid defining mailboxes where the local part is case sensitive"[p. 405] (11).

- @ symbol
  The @ symbol connects the domain and the person who owns the email address.

- Domain name
  A domain name consists of one or more parts separated by a period, such as example.com (9). Each part must not be longer than 63 characters and can contain the following:

- Uppercase and lowercase Latin letters (AZ, az)
- Numerical values (from 0 to 9), with the condition that the entirety if the domain cannot be numerical
- A dash (-), as long as it is not placed at the beginning or end of the domain name

The domain name indicates the name of the organization. It is an address that leads to the organization's website. When an email is sent, the sending mail server looks for another mail server that matches the domain name of the recipient's address (11). If someone sends a message to a user at company.com, the mail server first checks to see if there is a replying mail server at company.com. If so, it will contact the mail server to see if the username is valid. If the user is real, the email will be delivered.

- Standardization of local parts
  The interpretation of the local part of an email address depends on the conventions and policies implemented in the mail server. For example, case sensitivity can distinguish mailboxes that differ only in the uppercase of the characters in the local part, although this is not very common (11). Gmail ignores all dots in the local part of an @ gmail.com address to determine the identity of the account.

- Particularity
  There is a particularity that is added to the RG: the words dot and at are accepted *verbatim*, making reference to the signs used in the English language. These must be found within the square brackets to be accepted as valid email addresses. In the case of parentheses, they are allowed at either end of the email address in the local part, and they will be identified as comments. With the above, the emails (comment)test@mail.com and test(comment)@mail.com are valid. Table I shows some examples of valid and invalid emails.

Table I. Valid and invalid emails

Valid emails	Invalid emails
simple@example.com	Abc.example.com
very.common@example.com	A @ b @ c @ example.com
disposable.style.email.with+symbol@example.com	a "b (c) d, e: f; g<h>i [j \k] l@example.com
other.email-with-hyphen@example.com	just "notright@example.com
fully qualified-domain@example.com	this is "not \allowed@example.com
user.name+tag+sorting@example.com	this \still \"not \\allowed@example.com
x@example.com	i_like_underscore@but_its_not_allowed_in_this_part.example.com
example-indeed@strange-example.com	QA [icon] CHOCOLATE [icon] @ test.com
test/test@test.com	Testing.fail [at] failed@mail.com
mailhost!username@example.org	@ example.com
user %example.com@example.org	@ example.com
marc [at] mail [dot] is	Joe smith<email@example.com>
karmany [at] email [dot] net	.email @ example.com
test [at] email [dot] net [dot] co	email. @ example.com

2.2. Results and discussion

To verify that the application complied with the RFC 5321 and 5322 standards, it was subjected to a series of tests in different scenarios, where the operation of the main classes was evaluated along with their functions, as well as the results obtained, either in the console or within the application itself. In these tests, the entry of different email addresses and their operation was carried out together with the explanation of the class that performs the ratification. Finally, the algorithm was described and constructed, with all the grammatical rules applied in the validation of emails with the implemented library.

2.2.1. Construction of regular grammar

The RG for email addresses is as follows:

Chains W, where $W \in L(M)$ and $L(G)$. It is stated that $L(M)$ or $L(G)$ defines the language of grammar for $W \subseteq$ "All valid (not just common) email address structures".

RG is formally represented as:

$$(\{S, B, T, U, A, V, X, Z, T, E\}, \{a, b \ldots y, z - 0, 1 \ldots 8, 9\}, \{(S, aU), (A, .S), (U, @V), \ldots (R, a)\}, S)$$

The rules established for the construction of the RG were elaborated based on the syntax of the RFC 5321 and 5322 regulations. Tables II and III present the rules described for the local part, the @ sign, and the domain name.

Table II presents all possible characters (already explained) that can be generated, indicating that, when a period is detected, there must be continuity with at least one character, and, when the local part is finished, it must be followed by the @ sign. This is described in the states S, U, and A. For state X, this works in the same way as in the previous states, except for special characters. In this rule, a period may be needed, followed by other characters that accompany the domain, together with the opening of the square bracket (if that option is to be selected). After the period or without it, it is possible to have another combination of characters that includes the hyphen (valid by regulation). The rule can allow to continue writing after the script, moving on to the next one, which will no longer contain hyphens.

Table III corresponds to the period and the @ sign. The rule for accepting the expression dot (period) is added. It was also possible to determine the special case for accepting the word *at* (at).

After writing the characters comes a final character, since it is not possible for the domain part to contain a hyphen. The domain part ends with any character that is contained in the above-presented Tables, with the exception of the hyphen, as well as without continuing to any variable. The stack is made based on the construction of the RG, where, for each alphabet of the input symbols, the stack is filled or emptied depending on the states to which the automata goes. It can be seen, in Fig. 2, that there are more stack automata rules, which can be interpreted with the already explained grammar, considering that the stack will be eliminated as the conditions are met.

Table II. Characters valid for the local part and domain of the email

State S	Transition	State X	Transition	State S	Transition	State X	Transition
S →	aU aA aS	X →	aY aX	S →	0U 0A 0S	X →	0Y 0X
S →	bU bA bS	X →	bY bX	S →	1U 1A 1S	X →	1Y 1X
S →	cU cA cS	X →	cY cX	S →	2U 2A 2S	X →	2Y 2X
S →	dU dA dS	X →	dY dX	S →	3U 3A 3S	X →	3Y 3X
S →	eU eA eS	X →	eY eX	S →	4U 4A 4S	X →	4Y 4X
S →	fU fA fS	X →	fY fX	S →	5U 5A 5S	X →	5Y 5X
S →	gU gA gS	X →	gY gX	S →	6U 6A 6S	X →	6Y 6X
S →	hU hA hS	X →	hY hX	S →	7U 7A 7S	X →	7Y 7X
S →	iU iA iS	X →	iY iX	S →	8U 8A 8S	X →	8Y 8X
S →	jU jA jS	X →	jY jX	S →	9U 9A 9S	X →	9Y 9X
S →	kU kA kS	X →	kY kX	S →	!U !A !S	X →	-Y -X
S →	lU lA lS	X →	lY lX	S →	#U #A #S		
S →	mU mA mS	X →	mY mX	S →	$U $A $S		
S →	nU nA nS	X →	nY nX	S →	%U %A %S		
S →	oU oA oS	X →	oY oX	S →	&U &A &S		
S →	pU pA pS	X →	pY pX	S →	’U ’A ’S		
S →	qU qA qS	X →	qY qX	S →	*U *A *S		
S →	rU rA rS	X →	rY rX	S →	+U +A +S		
S →	sU sA sS	X →	sY sX	S →	/U /A /S		
S →	tU tA tS	X →	tY tX	S →	=U =A =S		
S →	uU uA uS	X →	uY uX	S →	?U ?A ?S		
S →	vU vA vS	X →	vY vX	S →	^ U ^ A ^ S		
S →	xU xA xS	X →	xY xX	S →	_U _A _S		
S →	yU yA yS	X →	yY yX	S →	‘U ‘A ‘S		
S →	zU zA zS	X →	zY zX	S →	{U {A {S		
S →	[U			S →	| U | A | S		
				S →	}U }A }S		
				S →	-U -A -S		

The construction of the algorithm is carried out in two files, which contain the graphical user interface and the logical part of the email validation.

2.2.2. Application

This GUI was developed with the Tkinter package, which is a binding of the Tcl/Tk graphic library. The above is not a single library but consists of a few different modules, each with a separate functionality and its own official documentation for easy interface construction. This was done under the

Table III. Special mail rules

States	Transition
A	→ .S
U	→ @V
U	→ atC
B	→ .T
B	→ .V
B	→ dotD

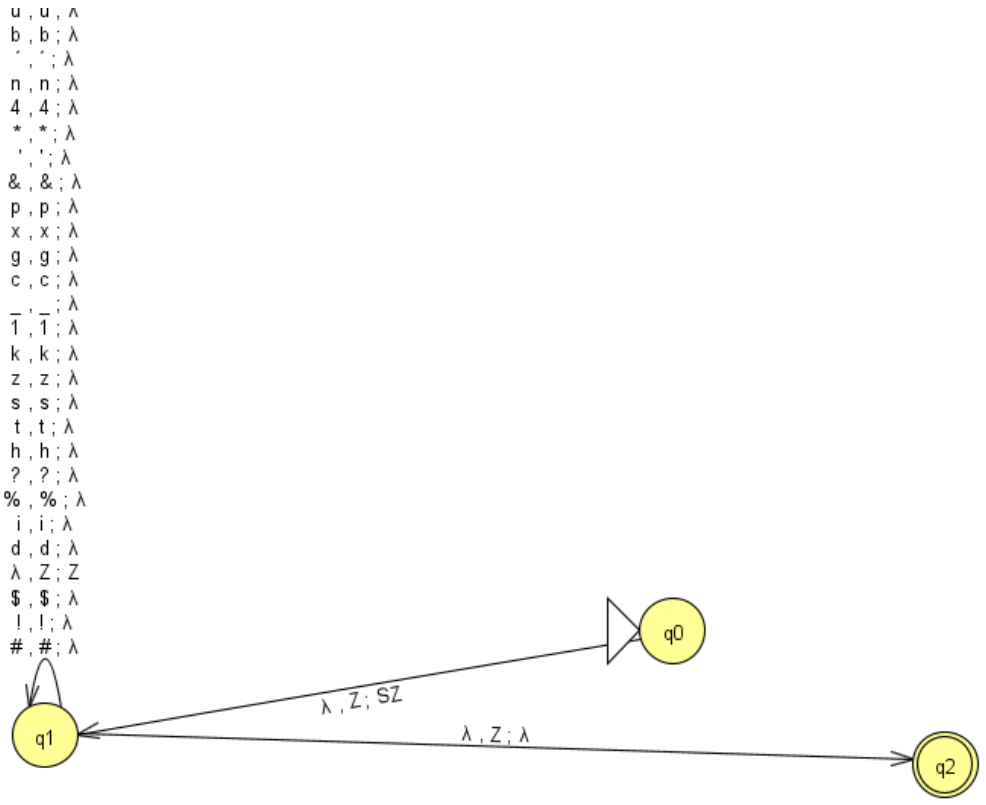


Figure 2. Generated stack automaton

Model-View-Controller design pattern, in which the program is initialized along with its corresponding screen. In the model, the validateMail class is called to validate an email. If everything is correct, it will return the value with the email; otherwise, it will show an error message with the email.

a. *Algorithm implementation*
In the logical part, the definition of global variables is proposed, describing the grammatical rules that must be fulfilled. These include lowercase and uppercase letters (using the upper function), numbers, special characters, the acceptance of symbols such as @ and the period, and adding square brackets.

When entering the validate function, a series of validations begins, such as the number of @ in the email and whether it contains atoms. Then, the mail is divided into two parts for verification, both the local part and the domain part, performing the corresponding iterations character by character. All the messages that are obtained in the results are stored in a list, from which it will be verified if the last two messages indicate that the email complies with the regulations. Next, test scenarios aimed at validating the application are presented, with the help of some of the classes that contain the main algorithm. This is done in different scenarios and under the input of several email addresses.

- *Scenario 1*
  When the program starts, it executes and creates the model instances with an example mail, together with the window with its size and the controller. The model carries out the instance by calling the main class, in which the email is verified. In this instance, the validation of the messages in the main class is performed by the stack. For an email to be valid, the last two messages are stored in an auxiliary variable, which will contain the last two positions, which would indicate the last two messages, as shown in Fig. 3a. For an email to be valid, it must have the message .ºK.ºr .ºK 2"; otherwise, it can indicate a message, such as the repetition of the @ character, two periods in a row, that the @ sign exists, or that it does not comply with any of the rules. This can be seen in Fig. 3b, for which the result will be shown in the console with both a valid email (first line) and an invalid email (second line).

```
    def email(self, value):            vm.validate(value)
        aux = vm.validate_mail()
        if aux[0] == 'OK 2' and aux[1] == 'OK 2' or aux[0]
== 'OK' and aux[1] == 'OK' or aux[0] == 'OK' and aux[1] ==
\
                'OK 2':
            self.__email = value
        else:
            raise ValueError(f'Invalid email address:
{value}')
    def save(self):
        pass
```

```
['OK 2', 'OK 2']
['Error. Ats is upper to 1', 'OK 2']
```

Figure 3. Instance of the *validateMail* class with results in the console

- *Scenario 2*
  The algorithm contains the *validateMail* class, for which the *validate* function is entered and the string of characters from the mail is received. In the first case, the validation of the number of @ or *[at]* expressions is carried out. Assigning the message to the end of the queue indicates that the rule of @ is not being fulfilled. It will be given by finishing the execution of the class; otherwise, it will go to another function in which validations are performed for the local part and the domain. The validation of the @ sign can be seen in Fig. 4a along with the result obtained when executing it (Fig. 4b).

For the case where the number of characters is verified, which cannot exceed 64, the message is added to the final part of the already created queue. At this point, two messages are returned: the first would indicate the error, and the second the entry to a cycle. For the mail to be valid, it must have two messages with validation at the end of the queue.

```
    def validate(word):      pos_at = word.find('@')
    pos_at_special = word.find('[at]')
    count_at = word.count('@')
    count_at_special = word.count('[at]')
    if count_at_special != 1:
        if count_at > 1 or word.find('@') == -1:
            last_message.append("Error. Ats is upper to
1")
        elif word[0] == '.' or word[-1] == '.' or word[-1]
== '-':
            last_message.append("Error. First or last
character is dot")
        elif word[pos_at - 1] == '.' or
word[pos_at_special - 1] == '.' or word[pos_at + 1] == '.'
or word[pos_at_special + 1] == '.' or word[pos_at - 1] ==
'-' or word[pos_at_special - 1] == '-' or word[pos_at + 1]
== '-' or word[pos_at_special + 1] == '-':
            last_message.append("Error. First or last
character with at")
        else:
            validation(word)
    else:
        validation(word)
```

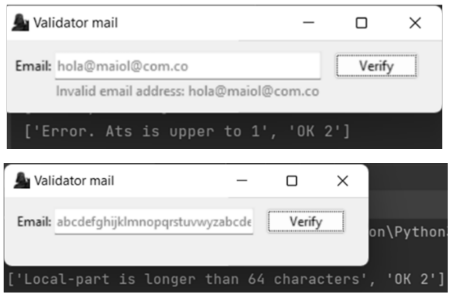


Figure 4. Validation of the number of @ and of characters (a) with their corresponding results for each case (b)

If the number of @ in the email is met, then it will move on to the validation function, which receives the string of characters for the email. In this case, two things can occur: if it contains @, another series of validations are performed, and another function of the algorithm changes on; otherwise, it is indicated that the @ symbol is not present, since it was not detected in the path of characters. Unlike the previous function, by regulation, this one checks the location of the symbol at the beginning or at the end of the chain. If the email does not contain an @ sign, the message indicating the error is added to the bottom of the list. If everything is correct in the previous validation, an *enumerate* loop is entered, which iterates the chain, given that, in Python, the chain is an array and, therefore, it can be traversed. If a string goes to *enumerate*, the output will show the index and value of each character in the string.

It should be noted that the algorithm only uses a library, which complements the function in which it receives double quote or parenthesis characters, indicating whether all its elements are the same in a list and working in the case that comments are contained between parentheses and one desires to validate that only letters have been written in the string.

- *Scenario 3*
  If the local part of the email has less than 64 characters, the *validate_character_local* function will be called, which will verify the local part of the string and evaluate its characters one by one. With

this function, several results can be returned: if the characters are letters, numbers, and special characters, the returned value will be *True*, whereas, for periods and @, 3 or 4 will be returned. This can be seen in Fig. 5a. The above is done with lists that contain all the valid letters, numbers, symbols, and characters that comply with the grammar rules. In some cases, validation is carried out with the *zip* function, which can go through several lists in the same iteration, working for the lists of uppercase and lowercase letters, along with the *dot-at-sign* list. The characters are entered one by one, validating whether they are in any of the lists and returning the corresponding value; otherwise, *False* will be returned, *i.e.*, the result shown in Fig. 5b.

```
if validate_character_local(v) == 3:
                    if word[i - 1] == word[i]:

last_message.append("repeat character dot")
                        break
                if word[i - 1] == '.' and
word[i] == '@':
                    last_message.append("repeat
character continue -at")
                    break
```

Figure 5. Validation of rules according to regulations and their results

If 3 is returned in the previous function, *validation* ensures that there are no repeated periods regarding their specific position with a loop, which allows the chain to be within a collection. Thus, if the route of the chain is found with two periods in a row, it will add the corresponding message to the list, iterating both in the previous position and in the current one. The validation also works for a period followed by an @ sign; the error is added to the last message, and the execution of the class ends. In this case, the validation of the local part is carried out, and the iteration is carried out with @.

Now, if the function returns 4, the message that an @ sign was detected is added to the list. If the email ends with @, an error will be displayed. After adding the message, the *validate_domain* function is entered, which receives the mail along with a number, which will indicate the cycle it must enter, be it 1, identifying the @ character, or 2, identifying the string "[at]. The above will depend on the result given in the *validate_character_domain* function. Then, in the same way, it is verified whether the periods are repeated or there are consecutive periods and @. If the rules are met, the message " OK 2" is added to the list, indicating that the domain part was successfully validated.

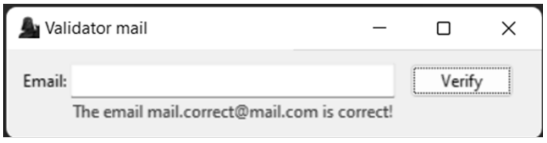
If 2 is returned in the *validate_domain* function, the string has been found to contain "[at", which is a special handling rule. From the above, the chain enters the functions 3 and 4, where the first function, which separates the local part of the domain with *split*, divides each time it finds a square bracket – if it contains more than one ([dot]), several separations of the string have to be performed. The first one contains the domain part, together with "at]", and is stored in *separate_one*, resulting in "at] domain.com". Now, a new division is carried out, but the separating character

is the square bracket "]", in order to obtain only the part of the domain. This is stored in a variable to be able to return it.

Upon receiving the character 1, the string is separated with the identifier of the @ sign using the *split* function, which receives the character in which a string is going to be separated, storing it in a list, and, if it only contains one @, then the separation is saved in two positions in the list. The first is for the local part, and the second is for the domain part, given that the variable will apply the second position to perform the domain validation. Again, as the local part is in a variable, it is verified whether it has no more than 64 characters; if it does not contain them, the function *validates_character_domain* receives the part of the domain and performs the same validations of letters, numbers, and periods with @, adding the possibility that the domain may have an underscore. Finally, the last two positions of the list are returned to indicate their result in the corresponding iterations and validations of each function, so that, in the *App* class, it can be verified whether the valid characters are contained in the last two validations.

According to the aforementioned validation, the GUI can show the result for the typed email (correct or incorrect). Here, there are two functions for each of these two messages, which are identified with colors so that the user can easily see if the result was satisfactory (text in green) or not (text in red), obtaining in a variable the email entered from the function knows from the controller. Note that *self* represents instances of the classes and is used to obtain what is stored in each of the variables. In this way, it is possible to carry out the validation of an, as shown in Fig. 6b.

```
def save(self, email):          try:
            self.model.email = email
            self.model.save()

            self.view.show_success(f'The email
{email} is correct!')

        except ValueError as error:
            self.view.show_error(error)

    def show_error(self, message):
        self.message_label['text'] = message
        self.message_label['foreground'] = 'red'
        self.message_label.after(3000,
self.hide_message)
        self.email_entry['foreground'] = 'red'

    def show_success(self, message):
        self.message_label['text'] = message
        self.message_label['foreground'] =
'green'
        self.message_label.after(3000,
self.hide_message)

        self.email_entry['foreground'] = 'black'
        self.email_var.set('')
```



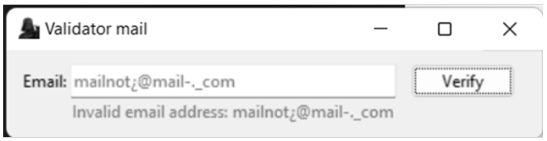


Figure 6. Functions to show the result in the graphical interface

The grammar rules were established in global lists at the beginning of the code, so that they could be used in all the functions, allowing them to be used throughout the code without having to redefine them, as well as to work with the established values. The rules for the types of letters, numbers, special characters, @ and period rules, and lines were applied to each list (Fig. 7).

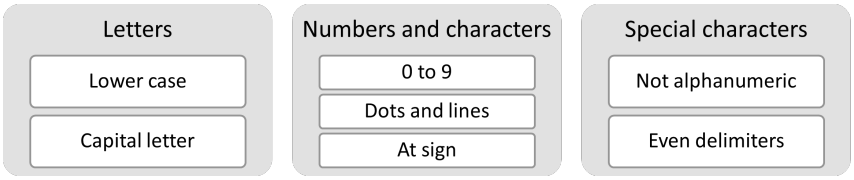


Figure 7. Grammar rules implemented for the algorithm

b Installing the generated library

To install the library, some characteristics are necessary, including the creation of the virtual environment with the corresponding command, within the folder for which the library will be used.

- *Scenario 1*

  The package is installed via a terminal command through the *Pypi* repository, which searches its repositories for the indicated package and downloads it. After this operation, it is possible to use the package (Fig. 8a). If the package was installed correctly, no error will be found in IDE (Fig. 8b). The terminal will display the information corresponding to the package from which the query is being made.

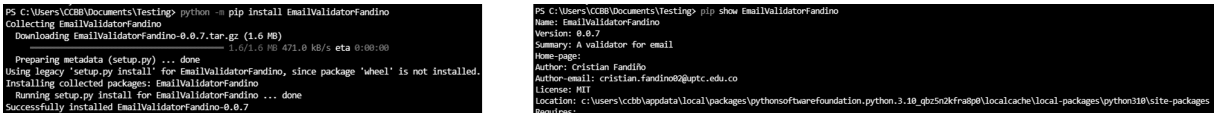


Figure 8. Library installation and package validation

- *Scenario 2*

  In this scenario, the utility of the library is accessed from the GUI. In it, the instance of the App class must be defined together with the call for the *mainloop ()* function. After the above, the window is displayed, in which it is possible to write email addresses for validation (Fig. 9).

- *Scenario 3*

  In this scenario, the library is used via the *validateMail* class, which uses its corresponding functions to enter the email address and show the result in the console: if the entry was valid, the result is *Mail is correct*; if not, it is *Mail is wrong*. This can be seen in Fig. 10.

  As seen in the previous scenarios, it was possible to use the library on another computer, either with the GUI or with the corresponding class, to be able to validate email addresses. In either case, the library complies with the RFC 5321 standard for the construction of an algorithm with regular grammars.

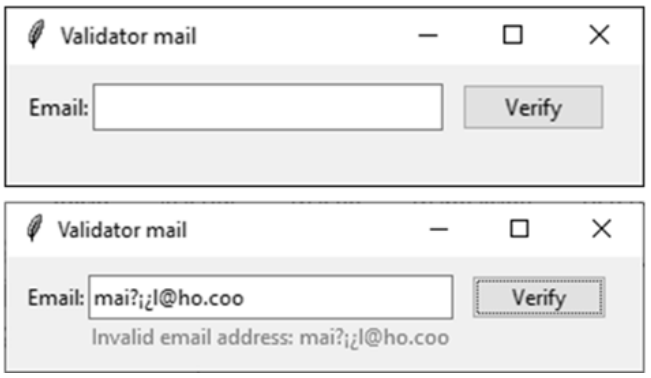


Figure 9. Using the library from the GUI

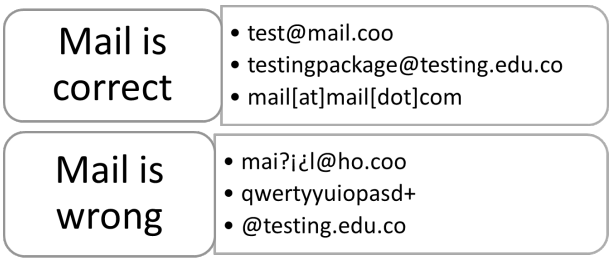


Figure 10. Valid and invalid emails

c. *Algorithm effectiveness*

When performing the validation of the library in real cases, its performance against the JFLAP program was verified. For this comparison of the number of executions *vs.* the effectiveness of results, tests were executed in both scenarios with the indicated number of 100 valid and 100 invalid addresses, with a certain increasing number of executions for them the results are presented in Table **IV**. This Table contains the information related to the valid and invalid emails applied in both parts. Note that the results should be the same.

These results corroborate the performance of the algorithm. According to the standard deviation found for both the algorithm and JFLAP, when executing valid emails, the former has a lower dispersion than the latter. Fig. **11** depicts the results with their corresponding linear regression, showing that the algorithm is more effective; on average, it reaches 99 emails evaluated in up to 150 executions, unlike the 97 obtained with JFLAP. In addition, when evaluating the 100 valid emails between 30, 50, 70, 90, and 110 executions, the algorithm has the highest accuracy, unlike the program with its grammar rules, although the former does not reach the full number of emails.

For the case with only 100 valid emails, it can be stated that the algorithm tends to be more efficient JFLAP; as the number of executions increases, it shows some differences with respect to the latter, whose trend line is not so steep. By comparing the equations generated by both cases, for every 20 executions, it can be expected that, on average, the truly valid emails tend to decrease, albeit not so periodically. This is because, for the algorithm, their number is greater. The variability percentage

Table IV. Execution results (library *vs.* JFLAP)

Executions	Email addresses		Result			
			JFLAP		Library - Algorithm	
	Valid	Invalid	Valid	Invalid	Valid	Invalid
5	100	100	100	100	100	100
10	100	100	100	100	100	100
30	100	100	99	100	100	97
50	100	100	95	97	99	96
70	100	100	95	97	99	97
90	100	100	97	97	99	98
110	100	100	96	95	98	97
130	100	100	96	95	98	96
150	100	100	99	92	96	95
Average			97	97	98,77	97,33
Precision			2,12	2,74	1,30	1,73
Accuracy			-3	-3	-1,22	-2,66

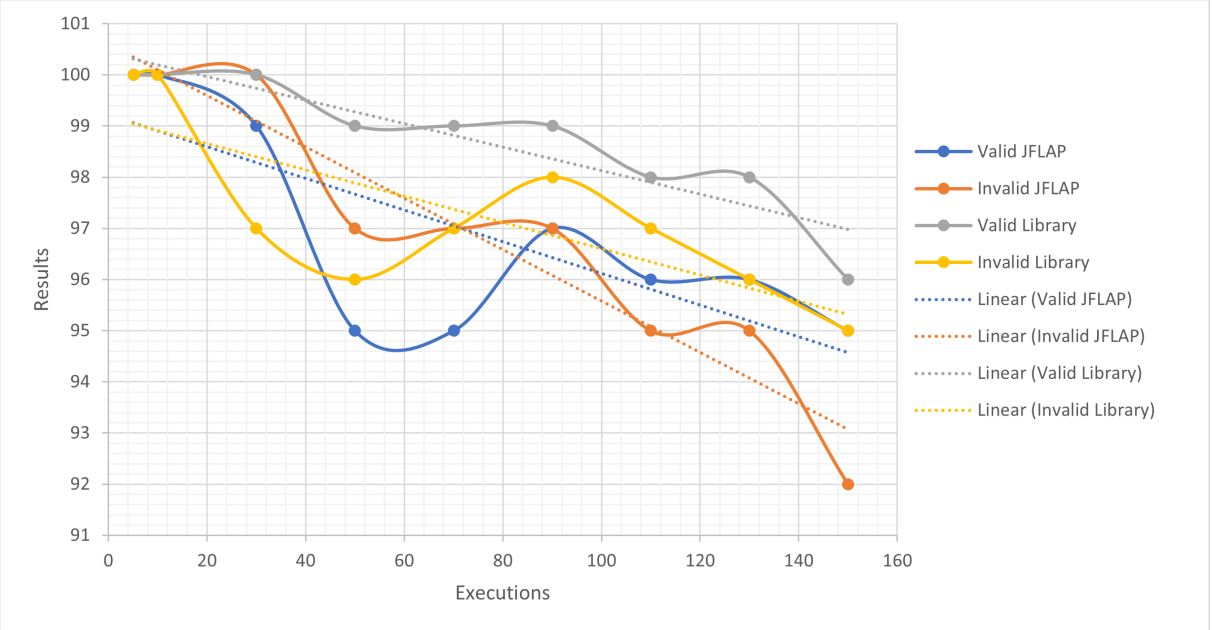


Figure 11. Results of (number of executions *vs.* validated emails)

is 85 % for the algorithm and 58 % for JFLAP. These results could have been generated by the resources of the test device or with the load that represents the multiple executions. The behavior of JFLAP shows a slight increase towards the end, unlike the library. Fig. 12 depicts the results and the equations. The opposite occurs with invalid emails: both had a different behavior because the algorithm tends to improve its results, as seen from the linear trend line. The accuracy of the

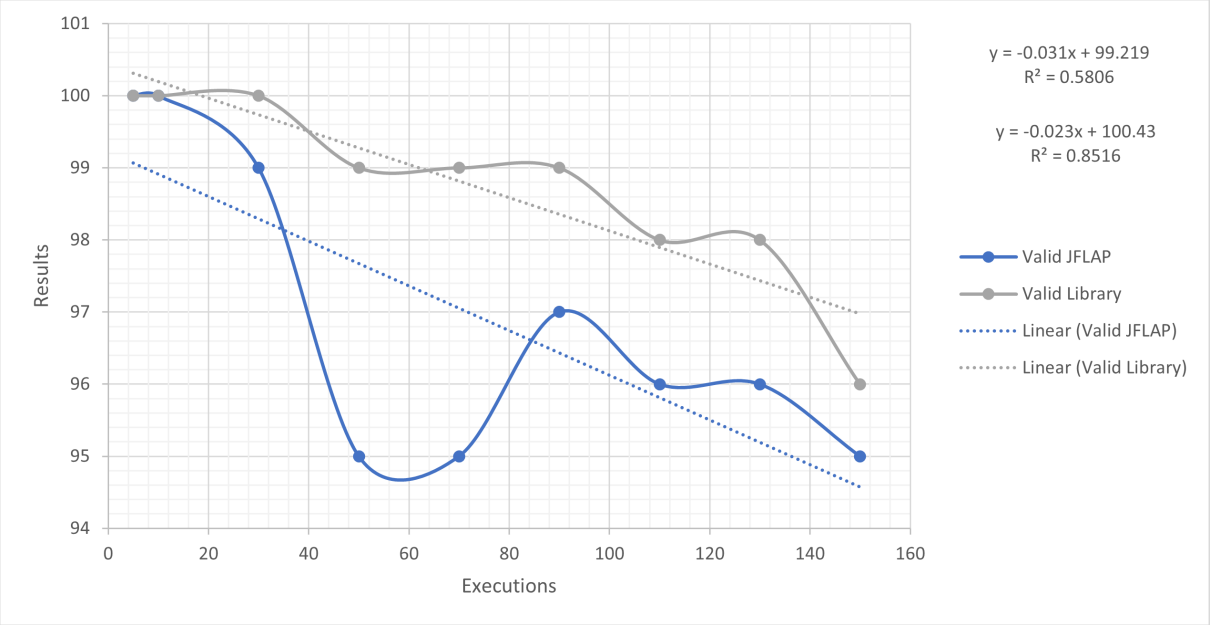


Figure 12. Valid emails for JFLAP and the library

algorithm is closer to zero, unlike JFLAP. The percentage of variability is 91 % for JFLAP and 59 % for the algorithm, since the data are less dispersed in the former than in the latter, which shows changes in the results between 50 and 90 executions. In this case, the library has better detection capabilities than JFLAP (Fig. 13).

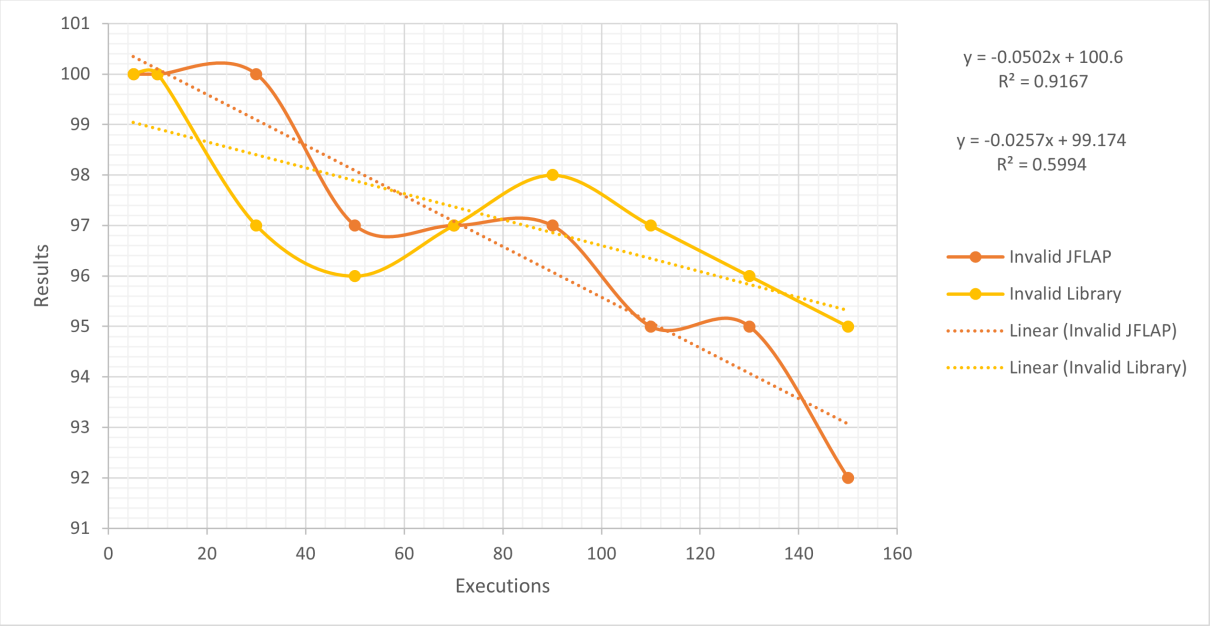


Figure 13. Invalid emails for JFLAP and the library

3. Conclusions

This study constructed an email validation algorithm based on regular grammars and provides notions for its practical use in the study of the branch of computing. Due to its characteristics, the Python language is an excellent tool for working with verification algorithms, to the point that its effectiveness works for corporations that require email validation. Likewise, the effectiveness of the library could be verified when comparing the results obtained by the RG algorithm and JFLAP, obtaining a greater margin for the validation of emails.

Email validation was performed by means of regular grammars, providing a unique pattern that works for all email formats, along with the definition of rules according to RFC regulations, which follows the format and accordingly builds a pattern. Each new rule reduces the degree of freedom in the accepted addresses. The algorithm is improved in terms of the limitation of cycles, the construction of several lists so as not to have to go through a single one and improve effectiveness, and validation procedures, which makes it work optimally with the released library.

Currently, semantic and syntactic analysis for emails can be performed using APIs such as SendGrid Validation API, Abstract API, or Mailgun Email Validation, which accept formats such as JSON and CSV. Regex, which matches text strings, *e.g.*, characters, words or character patterns through regular expressions, is one of the most widely tested to perform mail validation. Unlike the APIs, the library created is for free use, and it is possible to adapt it according to specific needs. The difference with regex is that it parses the string from the regular expression, not from the regular grammar, since a regular expression is much less readable than the original grammar – it lacks the nonterminal names that document the meaning of each subexpression.

Due to its precise matching of search patterns and its compact nature, regular grammar can be applied to validate the email addresses entered by the user in most day-to-day scenarios, paving the way for future APIs based on the shared library. Likewise, there is potential for future work in this area, with a focus on enhancing the algorithm's capability to analyze various lexical structures. These include NITs, alphanumeric codes, valid URLs (to combat *Web Spoofing and phishing*), and the validation of both urban and rural addresses. The library can serve as a great analyzer of grammatical or lexical structures, which are needed in the information technology industries, and it can even be complemented with machine learning for higher precision.

4. CRediT author statement

Marco-Javier Suárez-Barón: conceptualization, formal analysis, research, methodology, software, validation, writing-original draft.
Cristian Alejandro Fandiño: conceptualization, formal analysis, research, programming.
Cesar Jaramillo: software validation, writing, methodology.

References

[1] K. Dokter, F. Gadducci, B. Lion, and F. Santini, "Soft constraint automata with memory," *J. Logical Alg. Meth. Programm.*, vol. 118, art. 100615, 2021. https://doi.org/10.1016/j.jlamp.2020.100615 ↑2, 4

[2] T. Yamakami, "Between SC and LOGDCFL: Families of languages accepted by polynomial-time logarithmic-space deterministic auxiliary depth-k storage automata," in *Computing and Combinatorics*, C. Y. Chen, W. K. Hon, L. J. Hung, and C. W. Lee, Eds., Berlin, Germany: Springer, 2021, pp. 164-175. https://doi.org/10.1007/978-3-030-89543-3_14 ↑2, 4, 5

[3] K. Shuang, Y. Tan, Z. Cai, and Y. Sun, "Natural language modeling with syntactic structure dependency," *IJ Math. Sci. Comp.*, vol. 523, pp. 220-233, 2020. https://doi.org/10.1016/j.ins.2020.03.022 ↑3, 4

[4] J. Schwenk, *Guide to internet Cryptography*, Cham, Germany: Springer, 2022. https://doi.org/10.1007/978-3-031-19439-9 ↑3

[5] P. R. Chandra, K. Sravan, and M. S. Chakravarthy, "A new approach to the design of a finite automaton that accepts class of IPV4 addresses," *IJ Math. Sci. Comp.*, vol. 5, no. 1, pp. 65-79, 2019. https://doi.org/10.5815/ijmsc.2019.01.06 ↑3

[6] J. Chen, V. Paxson, and J. Jiang, "Composition kills: A case study of email sender authentication," in *29th USENIX Security Symposium*, 2020, pp. 2183-2199. ↑3

[7] E. G. Vázquez and T. G. Saiz, *Introduction to the theory of automata, grammars and languages*, Madrid, Spain: Editorial Universitaria Ramón Areces, 2022. ↑4, 5, 6

[8] A. Sharma and R. Kumar, "Imbalanced learning of regular grammar for DFA extraction from LSTM architecture," in *Soft Computing for Problem Solving*, M. Thakur, S. Agnihotri, B. S. Rajpurohit, M. Pant, K. Deep, and A. K. Nagar, Eds., Berlin, Germany: Springer, 2023, pp. 85-95. https://doi.org/10.1007/978-981-19-6525-8_8 ↑6

[9] M. Novo-Lourés,. D. Ruano-Ordás, R. Pavón, R. Laza, S. Gómez-Meire, and J. R. Méndez, "Enhancing representation in the context of multiple-channel spam filtering," *Inf. Processing Management*, vol. 59, no. 2, art. 6, 2022. https://doi.org/10.1016/j.ipm.2021.102812 ↑7

[10] G. Howser, *Computer networks and the Internet. A hands-on approach*, Berlin, Germany: Springer, 2020. https://doi.org/10.1007/978-3-030-34496-2 ↑7

[11] H. Tschabitscher, "LifeWire," 2021. [Online]. Available: https://www.lifewire.com/are-email-addresses-case-sensitive-1171111 ↑7, 8

Cristian Alejandro Fandiño Mesa
Systems and computing engineer, UPTC, Sogamoso, Colombia. Research assistant for the GALASH-UPTC group.
Email: cristian.fandino02@uptc.edu.co

Marco Javier Suárez Barón

Systems engineer, PhD in Strategic Planning and Technology Management, UPAEP, Mexico, Master of Information Management, Escuela Colombiana de Ingeniería. Associate professor at UPTC. Director of the GALASH research group.

Email: marco.suarez@uptc.edu.co

César Augusto Jaramillo Acevedo

Systems and computing engineer, MSc in Systems and Computing Engineering from Universidad Tecnológica de Pereira. Director and researcher in projects related to the Industry 4.0, precision agriculture, education, and business development. He has been a professor-researcher of Universidad Tecnológica de Pereira for more than 12 years. He has been active in research groups whose areas of interest and teaching are related to software engineering, compilers, AI, IoT systems, the cloud, distributed systems, and the Industry 4.0.

Email: swokosky@utp.edu.co