Towards the Assessment of Basic Computational Thinking Skills using Syntactic Analysis Techniques

Antonio Gonzalez-Torres
Computer Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica
antonio.gonzalez@tec.ac.cr

Elliot Ramirez-Trejos
Computer Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica
eart22@estudiantec.cr

Lilliana Sancho-Chavarria
Computing Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica
lsancho@tec.ac.cr

Jose Navas-Su
Computing Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica
jnavas@tec.ac.cr

Cesar Garita
Computing Engineering
Costa Rica Institute of Technology
Cartago, Costa Rica
cesar@tec.ac.cr

Jorge Monge-Fallas
School of Mathematics
Costa Rica Institute of Technology
Cartago, Costa Rica
jomonge@tec.ac.cr

***Abstract*—This article introduces an exploratory method for automatically grading programming exam questions using syntactic analysis. The target problem is the lack of a robust, scalable, and automated method to analyze computational thinking skills from source code written by elementary school students. The proposed method uses a variety of techniques to assess student responses, including analyzing the programming structure, programming correctness, and code execution based on certain parameters defined during the exercise specification. Analysis of the source code and evaluation of the answers to the exercises are carried out using high performance computing to improve the response time of the system. This preliminary work will contribute to a robust method for automated exam scoring, which is expected to assess and support the development of computational thinking among students.**

***Keywords*—Automatic evaluation, computational thinking, source code analysis, abstract syntax trees.**

I. Introduction

The fundamental role that technologies play today, as well as advances in automation, artificial intelligence, the Internet of Things, and cloud computing, position computational thinking (CT) as an essential way of thinking for people in everyday life and in different disciplines, particularly in the fields of Science, Technology, Engineering, and Mathematics (STEM) [1], [2]. However, there is no consensus on the definition of CT, it is considered as a process that transforms the way students reason and acquire skills to solve concrete and abstract problems by following a series of detailed steps that ultimately lead to a programmed solution. Therefore, it encourages the use of skills such as abstraction, algorithmic thinking, problem decomposition, pattern recognition, and analysis and evaluation of solutions to problems. This contributes to the development of critical thinking, creativity, communication, and collaboration.

The benefits stated have motivated its introduction in many educational systems, but it is necessary to continuously evaluate the results obtained during the teaching and learning processes of CT to improve it as a new discipline. In general, there are currently no robust and scalable methodologies to assess the level of students' CT based on the source code they produce as responses to exercises, assignments, and projects at schools. This requires recognizing the predominant learning patterns and determining the fulfillment of the objectives, competencies, and expected learning outcomes.

The research described in this paper has been developed as part of a research collaboration between the Costa Rica Institute of Technology (TEC) and the Omar Dengo Foundation (FOD). It aims at proposing methods to determine the CT level of students in problem solving with programming, while seeking to support the improvement of CT development in primary and secondary school students. The research has two main components: the definition of a programming language and a method to automate the classification of solutions to test questions and exercises. However, this work focuses on describing the latter.

The definition and development of the simple programming language was carried out to facilitate the problem solving process for Spanish-speaking students. This programming language is called LIE++[1], it is based on Spanish and is intended for use by elementary and high school students. The syntax of the language was defined by FOD as a simplified version of other programming languages, and its development was carried out by researchers and students at TEC. The method for automatic grading programming exams solved in LIE++ considers that the number of solutions to exam questions and exercises that students produce in a national context is enormous. Therefore, the design and development of this method considered the use of high performance computing using Spark [3].

[1]LIE++ uses keywords in Spanish and it it is written in Python. The output source code of LIE++ after compilation is also Python.

Consequently, the rest of this article describes the preliminary results of this research in the following sections. Section II presents some related work as part of the theoretical framework of the research project. Section III describes the method design, while Section IV describes the development of the method. Section V presents the validation of the method and Section VI summarizes the main conclusions and future work of this research.

II. Related Work

Computational thinking (CT) involves problem solving, system design, and understanding of human behavior coupled with the use of fundamental concepts of computing [4]. Some of the core CT skills are the following:

Abstraction: It is used to present information in a simplified way without losing important aspects.

Algorithm design: It is used to automate the resolution of specific problems.

Decomposition: It is used to divide large problems into smaller ones.

Problem formulation: It is used to formulate a problem in the correct way and transfer it to the computational environment.

CT is applied to STEM branches, but can also be applied to other disciplines such as art, music, and everyday tasks. In general, the trend is to consider CT as a required skill for any profession in the near future. Different approaches that provide automatic feedback to support faculty in the evaluation of CT and programming exercises have been applied to one or several courses taught in a given programming language [5]–[8]. The methods found in the literature for automatic evaluation of programming exercises include static and dynamic code analysis [9], machine learning techniques [10], unit testing [11], and a combination of these. Among the aspects that have been considered are functionality, efficiency, coding style, programming errors, and program design [12].

However, the source code used in previous research on automatic exercise scoring is not always publicly available, so it is not possible to reuse the resulting software. Furthermore, the few available tools do not allow massive analysis of programming exercises to verify compliance with the expected learning outcomes. Therefore, the definition of strategies to strengthen the training curricula offered to students based on evidence is limited.

Consequently, new methods and scalable, reliable, and effective techniques are necessary for the automatic analysis of large databases of programming exercises. These methods should provide information to support actions that help improve educational programs that promote the teaching of CT. In this sense, this work represents a novel contribution to the educational field and to computer science combining static source analysis and the use of high performance computing (HPC) with SPARK

The analysis of CT using as basis the code artifacts produced by students can be performed using static and dynamic analysis:

Static analysis: It is based on the study of the source code without execution.

Dynamic analysis: It consists of execution of the code to capture the events that occur while performing various tasks and observing its behavior.

Source code analysis is performed using compilers and translators that transform code from high level to low level or another syntax model to analyze features and calculate metrics that verify and evaluate various aspects of code quality according to certain criteria [13]–[15]. In this work, static analysis techniques are particularly applied to study the code created by students in the LIE++ language to extract metrics related to CT dimensions.

The number of source code artifacts produced by students participating in the programs led by FOD is huge. Therefore, this research considered the use of HPC to improve the performance of the analysis method [3]. HPC includes several variants based on clusters, networked computers, and cloud computing, and therefore there are many platforms and tools that can be used for HPC and big data applications [16]. In this particular work, we propose the use of Apache Spark to perform the analysis of a large number of exercises using multiprocessing techniques [17].

III. Method description

The definition of the method was made based on the research question **RQ1**.

RQ1 How to define a method to automatically grade a massive number of programming exam solutions in LIE++?

The grading of the questions takes into account three factors test:

- Use of programming structures.
- Code clones analysis.
- Execution verification.

Teachers specify the weight of each of these factors as a percentage when they prepare question exams. The result of each factor test is assigned to its corresponding percentage. Then, the percentages of the three factors are added to obtain the final grade for the question. The exam grading is calculated by adding the results of questions that correspond to it.

The process of answering **RQ1** and automatically grading the answers provided by the students follows the steps shown in Fig. 1:

1) The method receives the list of questions contained in the exam, where each question includes a detailed description of the exercise to be solved.
2) Then, it reads the list of responses provided for each question by the students.

Then, the method runs in parallel using Spark and starts reading the following details provided by the teachers during the question specification stage.

1) The list of the programming structures (e.g. if, for, and while) that are expected to be used by the student in the response.

2) Three or more different correct answers for each exam question.
3) The values of the input parameters for the exercise to be tested and the expected results it should produce for a correct solution.

Thereafter, it uses the above details and executes the following tasks in parallel:

- Perform a syntactic evaluation of the source code created by the students to answer the exam questions in LIE++. The aim is to identify the programming structures they are using (e.g., if, for, and while). The method also supports the evaluation of exercises programmed in Java, C#, and Python.
- Perform a code clone analysis. This analysis considers the sample answers provided by the teachers. Students' responses are compared with teachers' samples through clones analysis. This process results in a percentage of code similarity. If the result of the similarity analysis is low, the answer must be manually evaluated by a teacher.
- Execution of the source code of the answers provided by the students using the input and output values specified by the teachers when preparing the question. Student responses are automatically executed using the input parameters and the results are compared with the expected output value provided during the questions specification.

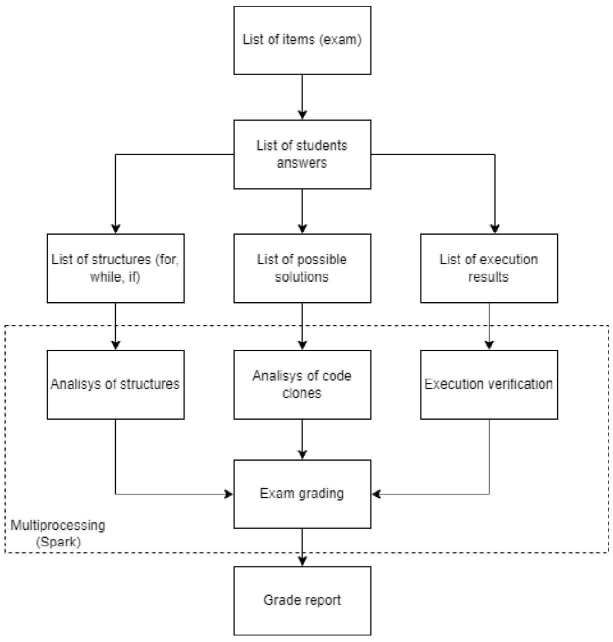

Fig. 1. Flow chart of main method activities.

Later, it takes the results of the above parallelized tasks and performs the weighting of the scores obtained during each type of analysis (i.e. use of programming structures, code clones analysis, and execution verification) to produce the exam grading. Then, it creates a grade report and stores it into a database.

Automatic code evaluation is performed with the support of a Generic Abstract Syntax Tree (GAST) [18], [19]. The GAST is created through the transformation of the abstract syntax tree of a specific language into a generic abstract syntax tree. It allows one to carry out code analysis in the different languages that were mapped to the structure of the GAST. Therefore, the analysis can be performed independently for various programming languages can be executed using high-performance computing when it is needed due to the high volume of information.

Figure 2 shows the steps followed for performing the analysis of structures and clones using the GAST. These include reading from the database *Language Grammar* the syntax and grammar associated with the language in which the answers under analysis were developed (see Figure 2, arrow 1).

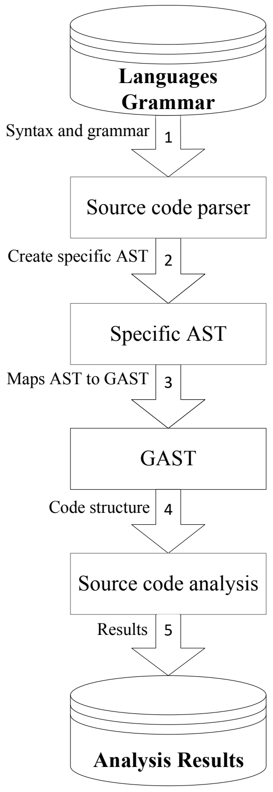

Fig. 2. Source code analysis with GAST.

The processing of the code of each response is performed by the task *Source code parser* using the relevant parser, and invokes the specific process to create the AST according to the associated language (see Figure 2, arrow 2). The next step invokes the GAST mapper to associate the elements of the particular AST with the corresponding ones in the GAST (see Figure 2, arrow 3).

Then, the task *Source code analysis* starts with reading the structure of the code represented by the GAST (see Figure 2, arrow 4) and consists of determining the structural composition of the code of the answers to the exercises (i.e. use of if, if-else, while, for, do-while) and identifying the similarity of student responses with the sample answers provided by teachers through source code analysis. The results produced by the Source Code Analysis process are stored in the database *Analysis Results* (see Figure 2, arrow 5).

Meanwhile, the *Execution verification* takes the source code and executes it with the input parameters and verifies that the output matches the expected output value specified by the teacher during the question preparation stage.

For the purpose of this paper, only responses to exam questions programmed in LIE++ language were considered, although the transformation process to the GAST structure was performed to validate the advantages of a language-independent method.

The following sections describe in more detail each of the phases of the grading process applied by the method to an example question. The percentage weights assigned to each one of the factors could be similar to the following:

- Use of programming structures (40%).
- Code clones analysis (30%).
- Execution verification (30%).

The grading process for code clones analysis and execution verification is straightforward, because the result satisfies or does not satisfy the evaluation criteria. However, the grading of the use of programming structures is more complex, as it considers several programming structures which use can be weighed differently.

A. Analysis of Programming Structure

The analysis of the programming structures contained in the student responses is based on the source code provided by them. This is carried out following the following steps:

1) Student responses are transformed to GAST by mapping the language-specific AST structure in which they were written to the GAST structure.
2) Student responses transformed to the GAST structure are interpreted by the code metric analyzer to determine the use of the required structures.
3) The grade for the specific exercise is calculated and stored in the database.

The weight assigned to the grading example shown in Table I for the use of programming structures represents 40% of the total grade of the item. This 40% represents 10 points for the exam item. Note the *Expected, Analyzed and Value* fields under the If and For structures in I and the values underneath.

The example in Table I shows that students were expected to use 5 *if* conditionals and 6 *for* structures (see *Expected* field). The points assigned to *if* conditionals and *for* structures are 4 and 6, respectively (see Value field). The result of the evaluation of the student's response is reflected by the *Analyzed* field. Consequently, the student used 3 *if* conditionals and 3 *for* structures.

Therefore, the *Partial grade* obtained for the *if* conditionals was!1.2 points and *for* structures 3 points, and the *Analysis grade* 4.2 points. The calculus of the *Partial grade* for each programming structure is *Partial grade* = *Value* ÷ *Analyzed*. Then, the *Analysis grade* result is calculated as the summation of the partial grades for all programming structures under evaluation, as following:

$$\sum_{i=1}^{n} \textit{Partial grade}_i$$

Partial grade is the grade obtained by the student on the exam item for the programming structure under evaluation.

TABLE I: Composition of the question subscores.

Structures					
If			For		
Expected	Analyzed	Value	Expected	Analyzed	Value
5	3	4	6	3	6
Partial grade			Partial grade		
1.2			3		
Analysis grade					
4.2					

B. Clone Analysis

Similarly to structural analysis, clone analysis is based on two inputs: sample code solutions provided by the professors (clones) and student answers. This analysis requires performing the following actions:

1) The identification of possible clones between the sample solutions provided by the professors and the student responses is performed using the GAST structure.
2) Once the degree of similarity between the sample solutions provided by the teachers and the response given by the student has been identified, the grade is computed for the exercise under evaluation.

The degree of similarity between the students' answers and those provided by the teacher is calculated using a GAST-based clone identification method. For example, Type 1 clones are fragments of identical or nearly identical programs, while Type 2 clones are code segments that are syntactically or structurally similar, and Type 3 clones are copied sections with significant changes.

C. Verification of execution

The verification of the execution results receives the input parameters and output values specified for the exercise and the code programmed by the students. The steps of this task are listed below:

1) The execution of the LIE++ compiler is performed, which produces Python code as output.
2) The translated response code to Python is executed using the input parameters.
3) The results obtained from the execution of each answer are compared with those specified by the teachers.

The execution of the LIE++ compiler is performed to produce the corresponding Python. The test score is calculated by comparing the results of the code execution with the output values.

The scores of the three stages (expected use of structures, clone evaluation, and execution results) are weighted and stored in the system database.

IV. Development

During the preparation of an exam item, teachers should provide details on the structures that students should use. Fig. 3 shows an example of a structure requirement provided by a teacher using a graphical interface: the student should use at least three (Cantidad, line 6) while (Nombre, line 4) structures and the grading weight (Peso, line 5) of the requirement is 3[2].

```
1 "Conditions": [
2     {
3       "idCondicion": 3,
4       "Nombre": "While",
5       "Peso": 3,
6       "Cantidad": 3
7     }
8 ]
```

Fig. 3. Conditional structure of the solution to a question.

Based on that condition, after processing a student response, Fig. 4 shows an example of the output of the structure analysis process. It can be seen that the 3 control structures (respuestaXEtapa, line 6) were expected according to the condition specified for the exercise. However, none were found, so the score obtained is zero (PuntajeObtenido, line 11).

```
1   {
2     "idRespuestaEstudianteXExamen": 1,
3     "idItem": 39,
4     "Respuesta": "**LIE++ Code**",
5     "RespuestaMapeada": "**LIE++ Mapped Code**",
6     "respuestasXEtapa": [
7       3.0,
8       0.0,
9       0.0
10    ],
11    "PuntajeObtenido": 0
12  }
```

Fig. 4. Output of the structural analysis process.

Clone analysis includes the use of a set of sample correct answers provided by teachers to check for the occurrence of clones among the code of answers. Fig. 5 presents an example in LIE++ of the code used as one of the possible solutions to implement the fibonacci function.

Fig. 6 shows an example of the result of the clone analysis. The result indicates that at least one of the possible solutions with some degree of similarity to the student's answer was identified. The grade obtained by the student is indicated on line 8.

In the case of execution verification, the Python code that is generated for LIE++ is executed. Fig. 7 shows an example of the result of the verification of the code execution of a response, in which the score obtained is reported on line 8.

[2]The language used in interfaces and coding is Spanish and a JSON format is used for the internal system representation of the inputs and outputs.

```
para fibonacci(pNumero)
    si (pNumero > 0) entonces
        arreglo := [pNumero]
        arreglo[0] := 0
        si (pNumero = 1) entonces
            devolver arreglo
        fin
        arreglo[1] := 1
        si (pNumero = 2) entonces
            devolver arreglo
        fin
        contador := 2
        repetir (pNumero - 2) veces
            arreglo[contador] := arreglo[contador - 2] + arreglo[
contador - 1]
            contador := contador + 1
        fin
        devolver arreglo
    sino
        devolver "Error: Debe ingresar un valor mayor que 0"
    fin
fin
```

Fig. 5. Source code sample in LIE++ provided by a teacher as a possible solution for a question.

```
1  {
2      "idRespuestaEstudianteXExamen": 45,
3      "idItem": 42,
4      "Respuesta": "**LIE++ Code**",
5      "RespuestaMapeada": "**LIE++ Mapped Code**",
6      "respuestasXEtapa": [
7        0.0,
8        213.0,
9        0.0
10     ],
11     "PuntajeObtenido": 0
12   },
```

Fig. 6. Result of clone analysis.

The programming languages used in the implementation of the prototype to validate the method were Java and Python, and also different libraries related to code parsing and text mining associated with the GAST. Spark was also used for high performance computing and Maven for managing software packages in Java.

```
1  {
2      "idRespuestaEstudianteXExamen": 45,
3      "idItem": 42,
4      "Respuesta": "**LIE++ Code**",
5      "respuestasXEtapa": [
6        0.0,
7        0.0,
8        11.428571428571427
9      ],
10     "PuntajeObtenido": 0
11 }
```

Fig. 7. Result of execution verification.

V. Validation

The prototype validation was carried out using functional tests, teacher evaluations, and performance analysis. Functional tests were performed using several cases to verify the operation of the prototype with respect to the analysis of code structures, clones, and execution results. In all cases, it was confirmed that the prototype worked as expected. In addition,

TABLE II: Scores obtained by the prototype.

Instructor	Grade
1	78.00
2	76.00
3	80.00
4	77.00
Average	77.75
Prototype	74.33
Difference	3.42%

a group of teachers was asked to perform a manual scoring of students' responses to an exercise. The result was compared with the grade calculated by the prototype (see Table II). It allows one to observe that the average grade assigned by the teachers (average) is similar to the grade calculated by the prototype (prototype), with a difference of 3.42%.

Finally, Fig. **??** shows a comparison of the time consumed by the prototype as a function of the number of students or exercise responses, considering a serial implementation (without multiprocessing) versus a Spark implementation (with multiprocessing). The test performed to compare the serial and parallel execution of the methods was also successful. Spark produced a performance improvement of 70.72% .

VI. Conclusions

The goal of the preliminary work presented in this paper is to propose an approach to the automatic evaluation of programming exams for elementary and high school students. The method combines different strategies for such assessment using rubrics for the evaluation of use of programming structures (e.g. If, 'Fo, and While), the verification of solution (code clones analysis) and the results produced by student answers (execution verification).

The prototype successfully allowed us to verify the use of programming structures in the solutions to programming questions provided by the students. Furthermore, it also allowed one to determine the degree of similarity between the students' answers and a set of reference solutions provided by the teachers, which enables one to identify, to some extent, the students' programming skills. Furthermore, verifying the execution of student responses by comparing actual results with expected results that were provided for each exercise with the corresponding input values was of great value for testing the method and student skills.

The validation carried out with teachers verified that the grades assigned by the prototype are similar to the average grades assigned by a group of teachers for a particular exam. Additionally, the prototype was implemented using multiprocessing techniques and tools, resulting in a significant improvement in execution time with respect to serial execution of the code.

References

[1] A. Gonzalez-Torres, L. Sancho-Chavarria, M. Zuniga-Cespedes, J. Monge-Fallas, and J. Navas-Su, "A strategy to assess computational thinking," in *8th Annual Conf. on Computational Science & Computational Intelligence (CSCI'21), Las Vegas, USA*, 2021.

[2] S. Pacheco-Portuguez, A. Gonzalez-Torres, L. Sancho-Chavarria, I. Trejos-Zelaya, J. Monge-Fallas, J. Navas-Su, A. J. Cañas, A. Rodríguez, and C. A. Chinchilla, "A method for assessing computational thinking in students using source code analysis," in *2022 International Conference on Advanced Learning Technologies (ICALT)*, 2022, pp. 144–146.

[3] T. A. S. Foundation. (2009) Apache spark. [Online]. Available: https://spark.apache.org/history.html

[4] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, mar 2006.

[5] C. Wilcox, "Testing strategies for the automated grading of student programs," *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016.

[6] M. Poženel, L. Fürst, and V. Mahnič, "Introduction of the automated assessment of homework assignments in a university-level programming course," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015, pp. 761–766.

[7] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *ACM J. Educ. Resour. Comput.*, vol. 5, p. 4, 2005.

[8] Z. Ullah, A. Lajis, M. Jamjoom, A. H. Altalhi, A. S. A.-M. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on novice programming: Strengths and limitations of existing systems," *Computer Applications in Engineering Education*, vol. 26, pp. 2328 – 2341, 2018.

[9] J. Caiza and J. Del Alamo, "Programming assignments automatic grading: Review of tools and implementations," in *INTED2013 Proceedings*, ser. 7th International Technology, Education and Development Conference. IATED, 4-5 March, 2013 2013, pp. 5691–5700.

[10] H. A. C. Morales, "Source code analysis on student assignments using machine learning techniques," Master's thesis, Departamento de Ingeniería de Sistemas e Industrial, Facultad de Ingeniería, Universidad Nacional, Colombia, 2017.

[11] A. B. Dieter Pawelczak and D. Schmudde, "A new testing framework for c-programming exercises and online-assessments," ser. 17th International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS 21), 2015.

[12] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005.

[13] J. Navas-Su and A. Gonzalez-Torres, "A method to extract indirect coupling and measure its complexity," in *2018 International Conference on Information Systems and Computer Science (INCISCOS)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2018, pp. 186–192.

[14] A. González-Torres, J. Navas-Sú, M. Hernández-Vásquez, F. Hernández-Castro, and J. Solano-Cordero, "A visual analytics architecture for the analysis and understanding of software systems," *Enfoque UTE*, vol. 10, no. 1, pp. 218–233, 2019.

[15] A. Bastidas Fuertes, M. Pérez, and J. Meza Hormaza, "Transpilers: A systematic mapping review of their usage in research and industry," *Applied Sciences*, vol. 13, no. 6, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/6/3667

[16] R. Robey and Y. Zamora, *Parallel and High Performance Computing*. Manning.

[17] E. R. Trejos, "Algoritmo de programación paralela para la evaluación masiva y automática del código de items de exámenes,," Master's thesis, Área Académica de Ingeniería en Computadores, Tecnológico de Costa Rica, 2021.

[18] J. Leitón-Jiménez and L. A. Barboza-Artavia, "Método para convertir código fuente escrito en diversos lenguajes de programación a un lenguaje universal," Master's thesis, Costa Rica Institute of Technology, oct 2021.

[19] M. Hirzel and H. Klaeren, "Code coverage for any kind of test in any kind of transcompiled cross-platform applications," in *Proceedings of the 2nd International Workshop on User Interface Test Automation*, ser. INTUITEST 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–10. [Online]. Available: https://doi.org/10.1145/2945404.2945405