UP! Designing a Readability-Centered Programming Language: Principles, Implementation, and Comparative Analysis

Erick José Heiler
Instituto Federal Catarinense
Blumenau, Santa Catarina, Brazil
erick.jose.heiler@gmail.com

Ricardo de la Rocha Ladeira
Instituto Federal Catarinense
Blumenau, Santa Catarina, Brazil
ricardo.ladeira@ifc.edu.br

ABSTRACT

This paper aims to create and evaluate a programming language designed for teaching programming logic. The language UP was developed focusing on readability and writability, seeking to be simple and suitable for beginners. Readability refers to how easy it is to read and understand code, while writability concerns how much the language demands from the developer. These characteristics guided the specification of the language, including its syntax, semantics, and fundamental structures. The language was implemented in Python. Exercises commonly used in programming education or representing the main structures of the language were selected to evaluate its characteristics. The UP language was analyzed using metrics related to readability and writability: total lines of code (LOC), effective lines of code (eLOC), source file size in bytes, and number of built-in functions, enabling comparisons between UP, Python, and C. UP achieved satisfactory performance: in LOC, it remained close to Python and better than C. In eLOC, it achieved better averages than both languages. Regarding file size, it was slightly worse than Python due to more descriptive structures but still better than C. The analysis of built-in functions showed that UP can express solutions with fewer built-in calls, highlighting its conciseness and expressiveness. Overall, the language proved suitable for the proposed objectives. Future work includes expanding the exercises evaluated, exploring new metrics, evaluating reliability, and verifying the use of the language in educational environments.

KEYWORDS

Programming Language, Readability, Writability.

1 INTRODUCTION

Programming has consolidated itself as an essential skill in the twenty-first century, fostering the development of logical thinking, problem-solving abilities, and intellectual autonomy in an increasingly technological society [1-2]. In this context, learning a programming language contributes to the formation of solid foundations in computational reasoning and facilitates future adaptation to new languages [3]. The choice of the first language is fundamental, influencing both the learning process and the quality of the code produced [4-5]. However, many decisions regarding this choice are based more on market demands than on pedagogical criteria, and several modern languages present complex syntaxes that hinder code comprehension and maintenance [6]. These factors highlight the need for a language that prioritizes readability and writability, characteristics that are essential for clearer and more accessible learning.

Given this scenario, this paper describes the development of the UP programming language, designed for teaching programming logic and grounded in the criteria of readability and writability. Its name refers to the idea of growth and evolution, concepts that align with the purpose of promoting learning and continuous improvement in the study of programming. Furthermore, it is a simple and writable name, characteristics that directly reflect the principles of the language.

The specific objectives of this research include specifying the UP language (its syntax, semantics, and fundamental structures) and implementing it according to these definitions. Additionally, this work selects statements commonly used in programming education, implements them in Python, C, and UP, collects metrics related to readability and writability, and finally compares the results among the three languages.

In light of this, the creation and analysis of the UP language is proposed as an alternative focused on introductory programming education, offering a simple, readable, and suitable tool for supporting the learning process and strengthening the development of computational reasoning.

2 METHODOLOGY

The method of this study was structured based on the typology of [7], considering the nature, objectives, and procedures of the research. Regarding its nature, the research is characterized as applied, since it seeks to apply existing knowledge to solve a practical problem, resulting in the development of the UP programming language.

Regarding the objective, it is classified as descriptive, as it aims to describe and analyze the characteristics that promote readability and writability in a programming language, as well as evaluate how these elements impact learning and code development.

Regarding the procedures, two main procedures were used. The bibliographic research, based on books, scientific articles, and other relevant sources, provided the theoretical foundation for the development of the UP language. The experimental research, in

turn, was carried out to evaluate whether the language meets the criteria of readability and writability.

The bibliographic research addressed concepts of readability and writability, in addition to fundamental topics for the development of programming languages, such as interpreter and compiler structures, lexical, syntactic, and semantic analysis, definition of data types, and evaluation metrics.

The study also analyzed the evolution of programming languages, identifying elements to be incorporated or avoided in the language. Additionally, the process of creating programming languages was extensively investigated, providing the theoretical basis to guide the development of UP and enable its comparison with other languages.

The implementation of the UP language was carried out in Python 3.9.13, the most up-to-date version during the specification phase. The process included defining the syntax, semantics, keywords, and creating the interpreter. All source codes are available at `https://github.com/erickheiler00/tcc-up-language`.

With the language completed, the experimental research began. A set of programming exercises was selected and implemented in Python, C, and UP. The C implementations were developed and tested using the GCC 6.3.0 (MinGW) compiler, while the Python implementations used version 3.9.13 of the interpreter.

Finally, a comparative analysis was conducted between the languages based on the metrics of lines of code (LOC), effective lines of code (eLOC), source file size (in bytes), and number of built-in functions, in order to evaluate the readability and writability of the language.

3 RELATED WORKS

In related works, studies on the creation of new programming languages, research comparing existing languages, and works using metrics to evaluate relevant characteristics, especially in the context of programming education, stand out.

In [18], the Julia language is presented as an open-source alternative to MATLAB®, aimed at teaching numerical and symbolic programming and supporting scientific research applications. The authors perform benchmarks with different algorithms to identify best practices and improvements, demonstrating that Julia shows superior performance in several tests.

Santos *et al*. [19] proposed PROGLIB, a language based on Brazilian Sign Language (LIBRAS) for students with hearing impairments, involving the creation of the language, development of an IDE with a visual interpreter, and validation through workshops. PROGLIB focuses on linguistic accessibility, while this work aims to create a more general language, intended for beginners.

In [10], Seabra, Drummond and Gomes compared Python, Java, and C through quantitative and qualitative metrics, analyzing six classic computing problems based on execution time, lines of code, file size, production time, documentation access, and use of built-in functions. Some of these parameters are directly related to readability and writability. The results showed distinct advantages for each language, highlighting the relevance of metrics associated with coding and productivity, which are also applied in the evaluation of the developed language.

The Temple language was developed to facilitate programming in introductory Computer Engineering courses, emphasizing programming logic and language simplicity to reduce learning barriers [3]. Like Temple, the language developed in this work aims to make programming more accessible and easier to understand, with the difference that Temple was created for specific courses, while UP was conceived for broader use, based on readability and writability.

4 DESIGN AND IMPLEMENTATION

This section presents the main ideas that guided both design and development of the UP programming language. The following sections summarize the principles adopted during its construction and provide an overview of how the language was implemented.

4.1 Design

The UP language was designed focusing on readability and writability, adopting the structured procedural imperative paradigm. This paradigm organizes the program as a sequence of instructions that define, step by step, how the processing should occur. In this model, assignment statements allow modifying the values stored in variables throughout execution [6].

Structured procedural programming organizes the code into procedures or functions and uses control structures to specify, clearly and orderly, the steps required for the program to produce the desired result [6].

UP is an interpreted language with strong and static typing, which does not include object-oriented features, maintaining a reduced set of basic constructs and avoiding complex structures. Strong typing prevents operations between incompatible values, while static typing checks variable types before program execution, ensuring that incompatible assignments are identified and blocked and preventing type changes during program execution [6].

The language contains integer types, floating-point numbers, logical values, text, and lists. Additionally, it provides operators, conditional structures, repetition structures, and functions. An illustrative example of UP code is presented in Frame 1, demonstrating the use of these constructs in practice, including functions, variable declarations, list manipulation, loops, conditionals, operators, and input/output routines. The program defines a function `divisores` which computes all divisors of a given integer (`int` type) by iterating from `1` to `num` (the loop runs from `1` to `num` inclusive, as defined by the range `1` to `num+1`), testing divisibility, and storing the valid values in a list

(`lista_divisores`). The main function reads an integer provided by the user, invokes `divisores`, and prints the resulting list of divisors, illustrating how UP handles typed input, function calls, and formatted output.

```
fun divisores(int num) -> list
    list lista_divisores = []
    for i = 1 to num+1 do
        if num % i == 0 then
            lista_divisores + i
        endif
    endfor
    return lista_divisores
endfun

fun main() -> void
    int num = input_int("Digite um número
para saber seus divisores: ")

    list lista_divisores = divisores(num)
    print("Os divisores do numero ", num,
" são: ", lista_divisores)
endfun

main()
```

Frame 1: A code example demonstrating basic constructs of the UP language.

4.1.1 Programming Language Evaluation Criteria

To continue the development of the language, it is necessary to understand the evaluation criteria of a programming language defined by [6] and determine which ones will be prioritized in the project. According to the author, readability and writability are fundamental criteria for evaluating a language, and they are influenced by characteristics that determine how clear and simple a program can be.

Readability is defined as the ease of reading and understanding a program, improving efficiency in both development and code comprehension [6]. Writability refers to the simplicity with which a programmer can develop programs using a given language, allowing the efficient creation of programs [6].

4.1.2 Readability-Oriented Decisions

The language syntax was inspired by natural language, making the understanding of instructions easier. To reinforce this characteristic, the language uses expressive keywords and explicit block endings, such as `endif` and `endwhile`, which make structures more visible and avoid ambiguities in block delimitation.

The data types also follow this principle, being defined in a simple and semantically appropriate way, such as the use of true and false for logical values. Moreover, the language requires the variable type to be specified on its first use, and its constructions have fixed and consistent meaning, avoiding multiple interpretations and contributing to a direct reading of the code.

The overall simplicity is maintained through a reduced set of basic constructions and the standardization of fundamental operations, such as variable increment, which has only one way of being performed, avoiding unnecessary variations. The language also presents a moderate level of orthogonality: operators like + behave uniformly with integers, lists, and literals, allowing additions, concatenations, and repetitions without specific rules for each type, facilitating reasoning about the behavior of the code.

The keywords have self-explanatory names in English and a single meaning, avoiding semantic overload, something that, according to [6], compromises understanding. The language also prevents keywords from being used as identifiers, reinforcing readability.

Finally, the expressions were designed to balance clarity and practicality. They are not short enough to harm readability, nor so long as to compromise writability. In some cases, the language uses a larger number of keywords to make certain structures more descriptive, prioritizing code clarity.

4.1.3 Writability-Oriented Decisions

The UP language was developed seeking a balance between simplicity and orthogonality, in order to limit the multiplicity of constructions and avoid redundancies that could hinder writing. The standardization of structures and the selection of resources ensure that the programmer produces clear code with minimal effort.

The language provides support for abstraction, hiding implementation details and simplifying development. The built-in functions form a reduced and expressive set, covering only the essential operations and avoiding extensive constructions, which reduces the amount of code and makes the writing flow more efficient.

Operators, functions, and other constructions were designed to provide expressiveness without requiring excessive detail. Thus, UP favors the writing of direct programs without unnecessary constructions, with a compact syntax that reflects human reasoning in a natural way.

4.2 Implementation

This section presents the implementation aspects of the UP language. It introduces the overall process involved in creating programming languages and describes the architecture of the UP interpreter, detailing the main stages responsible for analyzing and executing UP programs.

4.2.1 Process of Creating Programming Languages

According to [8], the process of creating a programming language involves defining its syntax, establishing the meaning of each construct, and implementing the structures responsible for

program analysis and execution. Even in interpreted languages, the stages follow the model presented by the authors, composed of lexical analysis, syntactic analysis, and semantic analysis, differing only by the absence of intermediate or machine code generation, since the interpreter executes the source program directly.

In lexical analysis, characters are transformed into tokens recognized as structural elements of the language, such as identifiers, keywords, operators, and punctuation marks. In syntactic analysis, tokens are organized according to the structural rules of the language, while in semantic analysis, it is verified whether the constructs have valid meaning, including type and consistency checks. Development also involves using a symbol table to record identifiers and internal structures that represent the program throughout the analysis phases [8].

4.2.2 Interpreter Architecture

To implement the interpreter, the fundamental characteristics of the language were defined, such as syntax, supported types, and behavior of basic structures. Based on this, the UP language interpreter was developed in Python and structured in four main stages:

- **Lexical Analysis:** Converts the source code into tokens, representing basic units such as keywords, identifiers, and operators.
- **Syntactic Analysis:** Organizes the tokens into an Abstract Syntax Tree (AST), representing the hierarchical structure of the program and detecting structural errors.
- **Semantic Analysis:** Checks type compatibility, expression coherence, and whether operations are allowed by the language.
- **Interpretation:** Traverses the AST, executing each node according to the language rules, producing the results.

Together, these stages describe how the interpreter processes and executes programs.

5 RESULTS AND DISCUSSION

This section presents the language evaluation process, including the metrics used (Section 5.1), the applied exercises (Section 5.2), and the comparative analysis of the results (Section 5.3), focusing on readability and writability.

5.1 Metrics for Evaluating the Language

To analyze the readability and writability of the language, metrics from studies on code quality were selected, and UP was compared with Python and C. [9] highlight that no single metric adequately represents code quality. Therefore, it is appropriate to use a set of metrics to analyze different aspects of readability and writability. The metrics in this study include the number of lines of code (LOC), effective lines of code (eLOC), source file size in bytes, and the number of built-in functions, which are presented and discussed below.

5.1.1 Lines of Code (LOC)

In accordance with [10], the number of lines of code (LOC) represents the total number of lines in a source program. This metric has some limitations, since different ways of writing code can produce different numbers of lines to solve the same problem. Even so, it is still widely used in comparative analyses [10]. In this work, the LOC metric is used to evaluate writability, considering that fewer lines indicate more direct solutions, aligning with the relationship between writability and the effort required to program.

5.1.2 Effective Lines of Code (eLOC)

The number of effective lines of code (eLOC) considers only lines containing relevant instructions, disregarding blank lines, comments, imports, and structural terminators (such as `endif` or blocks delimited by braces) [10]. The use of eLOC is important because, as per [11], LOC may not accurately represent the programmer's effort, including lines that do not contribute to the implemented logic. Bán and Ferenc [12] reinforce that effective lines more accurately show what actually composes the program's logic. Thus, eLOC was used to evaluate writability more precisely, directly reflecting the logical effort required to implement a solution.

5.1.3 Source File Size

As reported by [10], the source file size corresponds to the disk space occupied by the program. Although the storage cost is currently low, [10] highlight that code size remains relevant in analyses of complexity, maintainability, and structural comparison between languages. Research shows that readability depends on multiple factors. [9] emphasize that isolated readability metrics are not sufficient to understand how code is perceived by developers, and [13] reinforce that it mainly depends on the characteristics of each line.

[14] indicate that larger files tend to be perceived as more descriptive, which can contribute to readability in certain contexts. However, this perception may vary, as they point out that less experienced programmers tend to consider longer segments more understandable, while more experienced programmers prefer shorter code. Thus, the source file size was used as a readability metric, assuming that larger files indicate more detailed descriptions and smaller files reflect more concise syntax.

5.1.4 Built-in Functions Usage

The use of built-in functions was adopted as a complementary metric to evaluate writability and readability. In personal communication with Seabra, it was mentioned that languages with more built-in functions tend to speed up programming, since many operations are already available, reducing the effort required to implement basic functionalities [15].

However, the analysis showed that this relationship is not direct. In some exercises, certain languages required multiple calls

to built-in functions to solve part of a problem, while another language performed the same task with only one. This resulted not only in less code, but also in a more objective solution. In this way, writability does not depend only on the number of built-in functions used, but on how expressive and adequate they are for each situation.

According to [6], languages with many basic constructs may compromise readability. The larger the set of resources the programmer needs to know, the greater the chance that different developers will learn distinct subsets, which makes code comprehension more difficult.

Given this, it is understood that using fewer built-in functions to solve the same problem indicates greater expressiveness of the language, in addition to contributing to readability and writability. Even so, this result does not invalidate the idea that using more built-in functions can speed up development. It only reinforces that balance is necessary, since the impact depends on the type of problem being solved.

5.1.5 Selection of Languages for Comparison

The developed language was compared with Python and C considering the paradigm and the educational purpose, since both are widely used in introductory programming education [16-17]. Python presents a simple, intuitive, and beginner-friendly syntax, in addition to its wide adoption in educational and professional contexts [16]. C, even with its more complex syntax and greater structural rigor, remains an initial language in many courses due to its robustness and extensive use in computational-base applications [17].

Another determining factor was the contrast perceived between the two languages. As reported by [17], C usually requires greater structuring and generates more extensive and verbose solutions, while Python stands out for its simplicity, readability, and ease of use. The new language adopts an intermediate point between C and Python, making the comparison in terms of readability and writability more evident.

5.2 Selected Exercises

The exercise statements presented in this section were defined with the support of a professor experienced in teaching programming to beginners. The selection was based on exercises traditionally used in introductory algorithm courses. The goal was to cover problems whose solutions rely on fundamental programming structures and concepts, providing a basis for comparing the analyzed languages. All exercise statements and their corresponding solutions in each language are available in `https://github.com/erickheiler00/tcc-up-language/blob/master/exercicios/listasDeExercicios/lista_de_exercicios_impl.pdf`.

The same logic and structures were used in solving the exercises in different languages, although some have their own features that result in shorter code. In addition, considering that whitespace and the use of the TAB key influence the file size metric, all exercises were properly standardized to avoid undue variations in this measurement.

5.3 Comparison of Languages

This section presents the results of the metrics used to compare UP, Python, and C regarding readability and writability. The analysis considers the values obtained for each metric, and the data used in this evaluation are presented in Table 1.

All measurements presented in Table 1 were performed by the authors during the experimental phase of this research. The metrics were obtained directly from the source files generated for each implementation of the exercises. LOC and eLOC were counted according to the criteria defined in Section 5.1. The file size was measured in bytes from the source files, and the number of built-in functions was identified by analyzing the functions used in each solution.

5.3.1 Lines of Code (LOC)

The result showed that Python presented the lowest average (23.33 lines), followed by UP (26.83 lines) and C (40.17 lines), as shown in Table 1. The difference between UP and Python occurs mainly due to UP's mandatory structural terminators (`endif`, `endfor`, `endwhile`, and `endfun`), which make block delimitation clearer but increase the number of lines.

Even so, UP remained close to Python in most exercises, with only a few lines of difference. The only exception was exercise 1, in which UP had an advantage for not needing to import any module to calculate the logarithm. In exercises 8 and 9, the absence of native set and dictionary structures in UP and C led to the use of lists and arrays to simulate these types, while Python was able to use its native resources directly, resulting in a more straightforward solution.

When compared to C, UP showed a significant advantage, using fewer lines of code in all exercises. This is because it was necessary to use multiple `#include` statements in C, declare all variables before use, separate between `printf()` and `scanf()` for data input, use more extensive syntax to create the `main()`, and deal with greater complexity in handling strings and arrays. In exercise 7, which involves lists and extensive string manipulation, the C implementation required many more lines than UP. In longer exercises like this, the difference becomes even more evident.

5.3.2 Effective Lines of Code (eLOC)

The behavior of eLOC showed a relevant difference compared to LOC. UP obtained the lowest average (16.92), followed by Python (17.33) and C (24.17), as shown in Table 1. There was a change of positions between UP and Python, with C remaining with the worst average. UP's structural terminators increase LOC but are not counted as effective lines, which explains this difference. These results indicate that, for the exercises analyzed, the amount of effective lines in UP was similar to or slightly lower than in

Exercises	Ex1			Ex2			Ex3			Ex4			Ex5-v1			Ex5-v2		
Metrics/Language	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**
LOC	18	20	31	22	21	40	20	17	27	10	7	19	62	53	77	30	26	44
eLOC	16	16	22	15	15	21	13	13	16	6	6	10	36	36	46	18	18	27
File size (in bytes)	534	517	834	697	566	930	469	387	578	198	159	325	1176	1114	1630	649	601	831
Number of built-in functions	10	12	13	9	12	12	3	4	4	2	3	2	18	24	18	8	12	10
Exercises	**Ex6**			**Ex7**			**Ex8**			**Ex9**			**Ex10-v1**			**Ex10-v2**		
Metrics/Language	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**	**Up**	**Py**	**C**
LOC	18	15	31	47	39	83	21	18	31	38	32	47	18	16	28	18	16	24
eLOC	11	12	19	31	32	53	12	13	17	23	23	29	11	12	16	11	12	14
File size (in bytes)	444	417	687	1380	1230	2511	429	377	628	1042	919	1426	363	356	569	366	350	425
Number of built-in functions	2	4	5	4	4	18	2	3	3	6	6	15	5	9	9	4	6	5

Table 1: Summary of language performance for each metric across the evaluated exercises.

Python, without providing a basis for broader conclusions about logical complexity.

In six of the ten exercises analyzed, UP and Python presented exactly the same eLOC, suggesting that both followed the same logical structure. In the remaining ones, UP had an eLOC one line lower, because execution starts directly through the call to the `main()` function, while Python uses the block `if __name__ == "main":`, adding an extra line. This pattern remained even in the more complex exercises, such as exercise 5 version 1 (36 effective lines in both languages) and exercise 7 (31 in UP versus 32 in Python), reinforcing that logical equivalence is maintained even when the difficulty level increases.

Compared to C, UP also presented a better average. Implementing the same exercises in C requires more effective lines due to the mandatory declaration of variables, manual control of array indices, string manipulation with auxiliary variables, and explicit memory management.

With the eLOC results, it is observed that UP is not as verbose compared to Python. In terms of the programmer's actual logical effort, UP proved to be as expressive as Python. The difference in LOC arises only from the structure with explicit delimiters aimed at reinforcing readability.

5.3.3 Source File Size

The source file size metric showed that UP presented an average of 645.58 bytes, positioning itself between Python (582.75 bytes) and C (947.83 bytes), as shown in Table 1. The difference in relation to Python occurs, mainly, because UP uses delimiters, more detailed control structures, and functions with descriptive names.

In addition, function declarations in UP require the specification of parameter types and return types, contributing to readability. Its repetition structures are also more detailed than those of Python, which slightly increases the file size, but this difference does not grow proportionally to the complexity of the program, keeping values close between the two languages, even in larger exercises.

Compared to C, UP required less space in all exercises. This shows that its code is shorter and more direct, while C requires more lines and additional resources to perform the same tasks.

Overall, considering the set of exercises used, the results indicate that the byte difference between UP and Python is small. Its more explicit syntax promotes readability and makes the code structure clearer, without significantly increasing the programmer's effort.

5.3.4 Built-in Functions Usage

Regarding the metric of built-in functions, UP showed the lowest average (6.08), followed by Python (8.25) and C (9.50), as shown in Table 1. Even using fewer built-in functions, UP expresses solutions in a more direct and concise way, showing greater expressiveness.

In the exercises that request a numerical input, UP obtains the value using only `input_int()` or `input_float()`, while Python requires two calls (`int(input())` and `float(input())`), and C separates the process into two distinct functions (`printf()` and `scanf()`).

Iteration in UP does not require built-in functions, while Python depends on `range()`. In C, some tasks require multiple calls to auxiliary functions, especially for manipulating strings and arrays. Normally, the more calls to built-in functions a

language required, the more lines of code were generated compared to the others.

As reported by [6], many basic constructions, including built-in functions, make the language more difficult to learn and can compromise readability. Thus, UP stands out by offering fewer functions, yet more expressive ones. Python presents an intermediate quantity, while C illustrates the problem pointed out by the author, requiring many functions for simple tasks. In this way, the language ends up increasing complexity without bringing proportional gains in expressiveness.

Therefore, the analysis of the built-in functions metric highlights the importance of the expressiveness that the language offers. Thus, UP manages to express more with fewer functions, which reinforces its efficiency and aligns with the results obtained in eLOC.

6 CONCLUSION AND FUTURE WORK

The present work involved the creation, evaluation, and comparison of the UP programming language with Python and C. The development of the language ranged from its specification, based on the principles of readability and writability, to the implementation of its interpreter. For evaluation and comparison, ten programming exercises were developed in each of the three languages. The analysis considered specific metrics, including number of lines of code (LOC), number of effective lines of code (eLOC), source file size in bytes, and quantity of built-in functions.

The results indicate that, in general, UP remained close to Python in all metrics, in addition to presenting better results than C. In LOC, UP was slightly behind Python due to the delimiters. In eLOC, it presented values equal to or better than those of Python, showing similar logical behavior. Regarding source file size, UP remained close to Python, although slightly worse due to its more descriptive syntax. It is worth noting that the C language presented significantly worse results in the first three metrics. The metric related to the use of built-in functions showed the greatest variation, since in some exercises C performed worse than Python and in others it did not. However, in all evaluated exercises, UP used an equal or smaller quantity of built-in functions compared to the other languages, achieving a better result.

Therefore, the experiments demonstrate that UP achieved the proposed objectives, standing out for its readability and writability. Although these conclusions are based only on the sample of exercises analyzed, they provide a consistent basis for future studies, which may expand the set of evaluated exercises, explore new analysis metrics, assess the language reliability, and investigate its use in educational contexts.

REFERENCES

[1] C. M. C. França Reis, J. P. S. Santana, and E. M. Pedreira, "A importância do ensino da lógica de programação para os estudantes do ensino médio," in *Anais do Congresso Internacional de Educação e Geotecnologias (CINTERGEO)*, 2023, pp. 22–27.

[2] R. Narciso *et al.*, "Importância da programação na educação fundamental: preparando alunos para o futuro digital," *Revista Ibero-Americana de Humanidades, Ciências e Educação*, vol. 10, no. 3, pp. 268–282, 2024.

[3] J. M. M. Pimenta, "Temple – uma linguagem de programação para o ensino de programação," M.Sc. dissertation, Universidade de Évora, 2019.

[4] C. J. Burgess, "Software quality issues when choosing a programming language," WIT Transactions on Information and Communication Technologies, vol. 14, WIT Press, 1970.

[5] C. Britton, "Choosing a programming language," Microsoft MSDN Library, 2008. Available at: https://msdn.microsoft.com/en-us/library/cc168615.aspx

[6] R. W. Sebesta, *Conceitos de Linguagens de Programação*, 9ª ed., Bookman, 2011.

[7] R. S. Wazlawick, *Metodologia de Pesquisa para Ciência da Computação*, 3ª ed., GEN LTC, Rio de Janeiro, 2020, p. 152.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Pearson Educación, 1990.

[9] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?," in *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 417–427.

[10] R. D. Seabra, I. N. Drummond, and F. C. Gomes, "Análise comparativa de linguagens de programação a partir de problemas clássicos da computação," Revista de Sistemas e Computação-RSC, vol. 8, no. 1, 2018.

[11] E. F. Miglioranza, "Estudo comparativo de linguagens de programação para manipulação de vídeo em telefones móveis," 2009.

[12] D. Bán and R. Ferenc, "Recognizing antipatterns and analyzing their effects on software maintainability," in International Conference on Computational Science and Its Applications, Springer, 2014, pp. 337–352.

[13] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546–558, 2009.

[14] T. V. Ribeiro and G. H. Travassos, "Attributes influencing the reading and comprehension of source code – discussing contradictory evidence," CLEI Electronic Journal, vol. 21, no. 1, pp. 5–1, 2018.

[15] R. D. Seabra, "Comentário sobre funções nativas em linguagens de programação," mensagem pessoal, 29 out. 2025.

[16] G. A. da C. Pavan, L. S. Cardoso, A. de S. Conter, and K. Pereira, "Um estudo para aplicação do Python como linguagem de iniciação à programação no ensino fundamental II," in Congresso Latino-Americano de Software Livre e Tecnologias Abertas (Latinoware), SBC, 2024, pp. 504–507.

[17] A. A. Barbosa, D. Í. S. Ferreira, and E. B. Costa, "Influência da linguagem no ensino introdutório de programação," in Brazilian Symposium on Computers in Education (SBIE), 2014, vol. 25, p. 612.

[18] J. M. M. Pereira and M. B. B. Siqueira, "Linguagem de programação JULIA: uma alternativa open source e de alto desempenho ao MATLAB," Revista Principia, no. 34, pp. 132–140, 2017.

[19] R. E. S. Santos, C. V. C. Magalhães, J. S. C. Neto, and S. S. L. Paiva Júnior, "Proglib: Uma linguagem de programação baseada na escrita de LIBRAS," in Workshop de Informática na Escola (WIE), SBC, 2011, pp. 1533–1542.