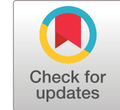
Intelligent Automation
& Soft Computing

Tech Science Press

DOI: 10.32604/iasc.2023.036102
Article



NewBee: Context-Free Grammar (CFG) of a New Programming Language for Novice Programmers

Muhammad Aasim Qureshi[1,*], Muhammad Asif[2] and Saira Anwar[3]

[1]Department of Computer Science, Bahria University Lahore, 54000, Pakistan
[2]Department of Law, Science and Technology, University of Bologna, 40126, Italy
[3]Department of Multidisciplinary Engineering, Texas A & M University, College Station, 77843, USA

*Corresponding Author: Muhammad Aasim Qureshi. Email: maasimq@hotmail.com
Received: 17 September 2022; Accepted: 23 November 2022

Abstract: Learning programming and using programming languages are the essential aspects of computer science education. Students use programming languages to write their programs. These computer programs (students or practitioners written) make computers artificially intelligent and perform the tasks needed by the users. Without these programs, the computer may be visioned as a pointless machine. As the premise of writing programs is situated with specific programming languages, enormous efforts have been made to develop and create programming languages. However, each programming language is domain-specific and has its nuances, syntax and semantics, with specific pros and cons. These language-specific details, including syntax and semantics, are significant hurdles for novice programmers. Also, the instructors of introductory programming courses find these language specificities as the biggest hurdle in students learning, where more focus is on syntax than logic development and actual implementation of the program. Considering the conceptual difficulty of programming languages and novice students' struggles with the language syntax, this paper describes the design and development of a Context-Free Grammar (CFG) of a programming language for the novice, newcomers and students who do not have computer science as their major. Due to its syntax proximity to daily conversations, this paper hypothesizes that this language will be easy to use and understand by novice programmers. This paper systematically designed the language by identifying themes from various existing programming languages (e.g., C, Python). Additionally, this paper surveyed computer science experts from industry and academia, where experts self-reported their satisfaction with the newly designed language. The results indicate that 93% of the experts reported satisfaction with the NewBee for novice, newcomer and non-Computer Science (CS) major students.

Keywords: Programming language; formal language; computer language; language grammar; simple syntax programming language; novice programmer

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1 Introduction

A structured communication system between humans, computers or their intersection (i.e., human to human, computer to computer and human to computer or computer to human) is known as language [1]. In communications and computer science, the role of two types of languages is predominant: natural and formal languages. Natural language is the mode of communication between humans. This communication has different forms in natural languages, e.g., spoken, written and symbolic (i.e., signs, gestures). However, formal languages are inherent languages extracted from the natural languages with a specific set of rules. In computer science, formal languages serve as the basis for defining the grammar and syntactical aspects of computer programming languages.

Computer Programming Languages (CPL) (languages needed to write software programs) are formal languages that provide an interface between computer-to-human, computer-to-computer, or human-to-computer interactions. Besides interface, CPL expresses computing-specific logic according to language rules and syntax [2].

In computing, programming languages are a central component [3]. These languages are designed to bridge the gap between the machine and the human who want to perform some specific tasks on the machine [4]. In today's era, learning programming and the ability to program are essential skills as reflected by various international policy documents for pre-college [5] and undergraduate [6] students. Considering the importance of computing and computational thinking, skills and abilities to understand computer languages are fundamental skills. The knowledge and ability to program allow the students to get the maximum benefit from computers as it helps in problem-solving, computational thinking and simulations. Through these programming languages, one can guide computers to perform some specific tasks. Programming knowledge is the fundamental and central component of computing, computational thinking, and computer science education [7]. Over the time, many programming languages have emerged, highlighting varying aspects of programming mechanisms, including low-level, structured, object-oriented and high-level programming languages. However, choosing which language to use is critical as every language has its specifications, limitations and affordances [8].

The existence of many programming languages raises another critical question; why do we build another language? [9]. Futhermore, what fundamental variation the new language actually provides viable tool to write programs in relation to others. Literature suggests that every new language overcomes the weakness of the existing ones or meets the new specific goals [10]. For example, some languages have exceptionally rigid language structures (syntactic), while others enforce indentations. It becomes a nightmare for new developers, novice programmers and non-CS major students to remember such superficial punctuations and constraints [11], primarily when they work on relatively large and complex projects. Literature suggests that the right way of learning programming is to shift programming students' attention to logic instead of worrying over the sentence structure of the code [12]. For example, a Java developer would be consuming their energies over the brackets and blocks, while a python coder will be stressing over the indentation. Similarly, the developer of C and C++ must stress over the semi-colon for the termination of the instructions [12]. Furthermore, sometimes programmers must consider the data types while writing the code, which leads to fatal errors if handled incorrectly [13].

Given these language nuances, present programming languages can be summed up in two significant classes, i.e., firmly bound and approximately bound dialects [14]. The firmly bound languages are early dialects in which developers must deal with each detail like proper blocks, indentation, semi-colons and different keywords. C, C++ and Java are examples of early dialect languages [15]. Approximately bound languages have relatively less stress over such structural aspects. However,

developers have to follow some sentence structure level constraints in these languages—examples of such languages include PHP and Python [16].

This study aims to introduce a context-free grammar of a programming language for new developers, novice programmers and non-CS major students. This paper hypothesizes that using features, this language "NewBee" will help students in learning the language in a more relaxed way. The premise of NewBee is to shift students' focus from syntax to logic building. Consequently, students will be learning to remove logical errors and will have less focus on syntactical errors.

The rest of the paper is organized into seven sections. Section 2 focuses on the previous related research on the grammar of different programming languages. Section 3 discusses the approach which is adopted to hypothesise the design of a language including the steps involved. Section 4 focuses on the grammar of the language. Section 5 list the limitation of the language. In Section 6 some sample programs are given. Finally, Section 7 concludes the paper.

2 Literature Review

A program is a set of instructions given to any computing machine to perform some specific tasks to achieve a predefined target. Programming languages are characterized into two types—Domain-Specific Languages (DSLs) and General-Purpose Languages (GPLs) [16]. DSL deals with the specific class of problems that belong to a specific domain, including databases, web applications, etc. Examples of these languages are HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and Structured Query Language (SQL) [17].

GPLs are used for building software in various application domains. C, C++, C#, Python and Java are examples of General-Purpose Languages [18].

DSLs can be implemented as External Domain Specific Language (EDSL) and Internal Domain Specific Language (IDSL) [19]. IDSL is the domain-specific language that is implanted into a general-purpose language that is limited with the grammar of the host language [20].

Few practices were carried out in the past in domain-specific languages. For example, Van Deursen et al. [21] listed the practices for the attainment approach at the aspect level only for semantics. Similarly, Oliveira and colleagues [10] surveyed Domain-Specific Language where the attainment details were not given. Furthermore, authors of [22,23] focused on the composability aspects of language and provided details on how they can cover different language workbenches.

Prior research studies have used various strategies to propose new languages. For example, in [23], a new language was presented to evaluate the DSL attainment approaches based on the unified state machine. The authors claimed that no single approach is valid for all scenarios. Similarly, prior literature studies [24–27] have used systematic mapping of existing languages. For example, in [24], authors surveyed to examine the methods and techniques of the existing DSL. They evaluated the domains and tools used in the creation of DSL. The authors concluded that external DSL for various domains were gaining a lot of attraction and described them as invaluable and relevant across domains.

Various literature studies [25,26,28] conducted their examination based on systematic mapping and captured the DSL field's research space and trends over a given period. The authors concluded that DSLs would be the primary programming language for the foreseeable future. Furthermore, one of the unresolved issues in the DSL field is how to make DSL development easier for domain experts. Additionally, the authors examined current approaches to resolve the issue mentioned above through the survey.

Similarly, in the Language Workbench Challenge 2013, studies [29,30] proposed a feature model for language workbenches and categorized them using the model. The authors presented a uniform challenge (i.e., a DSL for surveys) implemented by ten workbenches. The study examined the properties of various workbenches in several ways. It showed that no single language workbench could deliver all the required features. The authors are more concerned with the available features of the workbench than with the methods employed to obtain them.

Motivation: Although these studies used various methods to examine the methods and techniques of existing DSL, they highlighted unresolved issues. It is noteworthy that these unresolved issues and heavy grammatical rules can be very confusing for novice programmers which provides the motivation to conduct this study.

When novice programmers, newcomers to computing majors and non-cs major students start learning logic development, using some programming language, they face many issues and difficulties. Many of these issues are related to the intrinsic hard nature of programming [31] Also, some issues are associated with the complexities of logic building, syntax and sentence structure [32]. The most hazardous thing in writing the code for a complicated program is not always logical, but it is syntax or sentence structure [33]. To our surprise, the lack of even a single semi-colon or bracket in the complicated code is one of the worst nightmares of a programmer, while in some cases, the data type parsing issues are one of the major concerns for programmers [34]. This study presents a context-free grammar for a programming language for novice programmers, newcomers to CS majors and students with non-cs major backgrounds. This language supports these learners in logic building and provides an easy way to comprehend the language's syntax.

3 Language Design Approach

The programming language, i.e., NewBee, addresses the sentence structure level issues raised by the programmers. It makes the syntax simple and allows novice students to concentrate on the logic. The significant goal is to develop a language for which new students don't need to struggle to remember or memorize the syntax. Along with ease, a few language constraints are added to the language. The premise of these constraints is rooted in the need and level of novice programmers. This paper used a systematic approach to accomplish such a degree of facilitation. The methodology of this paper handpicked language rules that concentrate on straightforwardness for the programmers. This paper chose these rules from previous languages and combined them into one language. We believe that the language built from straightforward rules and emphasizing less structure will provide the necessary ease to novice programmers for learning and implementation in the NewBee. The methodology of the proposed language is shown in Fig. 1.

For instance, in the proposed language, novice programmers don't have to end the statement with a semi-colon as is required in C++ or Java. Similarly, the NewBee is not space touchy as Python. Additionally, the proposed language facilitates the programmers on sections stress to begin a block. They need to begin a block with a mark which is an idea preoccupied with the low-level computing construct.

In this article, a new language is being proposed to give syntax-level ease to the novice, newcomers and non-CS major students. By keeping the syntax close to students' daily language, syntax requirements may be least bothersome causing fewer syntax errors. The following section explains our proposed language sentence structure and introduces the syntax of the language.

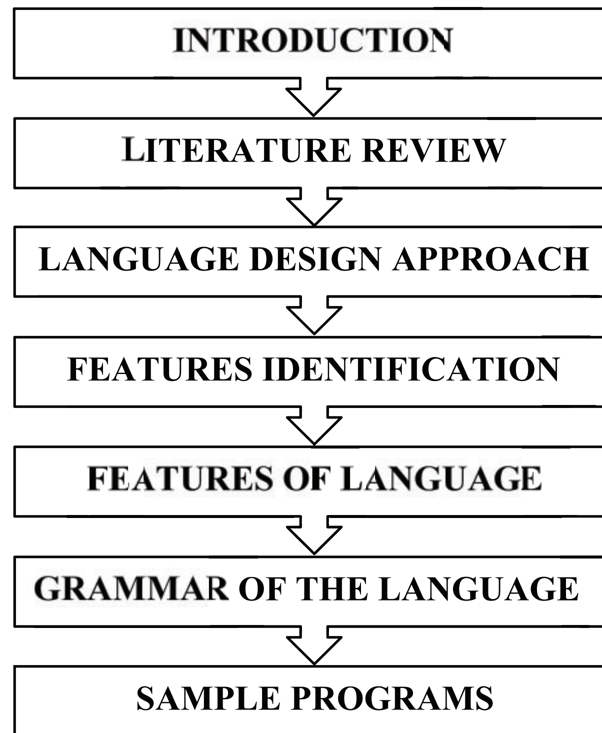


Figure 1: Methodology

3.1 Language Feature Identification

To identify the ideal features, this paper conducted a short survey including a sample of industry developers, students and instructors $N_{Instructor} = 7$; $N_{Students} = 23$; and $N_{Developers} = 11$. The survey results indicate that the following features of the NewBee are a necessary component for the novice, newcomers and students who do not have computer science as their major:

1) Program Execution Sequence
2) Statements Termination
3) Decision-making Structure
4) Loop Structures
5) Functions
6) Type Specifier
7) Operators
8) Comments
9) Reversed Words

3.2 Feature of "NewBee"

Following are the key features of the proposed language.

3.2.1 Program Execution Sequence

The program Execution sequence in NewBee Language begins with the keyword "Start of Program" to let the compiler know that the program has Started. In the same way, the program will

end with "End of Program" Anything that was written after "End of Program" will not execute and a developer can write their notes/explanations here.

Code Example:

```
Start of Program
    < Your program code goes here>
End of Program
```

3.2.2 Statement Termination

The termination of the statement will be without any visible terminator; instead, it will be terminated just with the next line.

Code Example:

```
a = 10
Display a
```

3.2.3 Decision-making Structure

If then structure:

Decision-Making Statement in NewBee will start with the keyword "If" and the block termination will be with "End of If". The language provides braces-free syntax so there will be no brackets for decision-making structure.

Code Example:

```
If a<=10
   Display a
End of If
```

If then Else structure:

The If Else statements are again the block statements. These statements will also be written in the same way. This means starting with a keyword i.e., "If". Followed by the then-part statements. The else part will start with "Else" and will end at "End of Else".

Code Example:

```
If a > b
    Display "a is larger"
Else
   Display "b is larger"
End of Else
```

3.2.4 Loop Structure

The language provides three different types of looping structures:

i. "from till" loop structure
ii. "while" loop structure
iii. "do while" loop structure

From-Till loop

The language provides “from-till” which iterates from the given value till the condition is true. It has two versions one is with an auto-increment of 1 and the other one will increment the loop counter with the value mentioned after the keyword “with”. Termination of the structure is with the key work “End of Loop”.

Code Example:

```
From i = 0 till i<=5
            Display “Hello World”
End of Loop
```

While loop

The language provides the “While” keyword as the start of the loop and the block will terminate with the keyword “End of Loop”.

Code Example:

```
a = 0
          While a<=10
                   Display a
                   a++/
          End of Loop
```

Do While:

```
do
        a= a+1
         Display “a”
        While a < 5
End of Loop
```

3.2.5 Functions

All functions will be written at the end of the main program. It will start with the keyword “Function” and will end at “End of Function”.

Code Example:

```
Start of Program
          add (10 20)
Function add (a b)
            c=a+b
            Display c
End of Function
```

3.2.6 Types of Specifier

This language is free of data type i.e., there is no need to write any data type keyword to declare the identifier. The data type will be automatically cased at the time of assignment means if the value is an integer the identifier will become an integer if it is a float then the identifier will become afloat.

Code Example:

```
a = 1
b = 2.5
c = abc
d = a
```

Code Example:

```
From i = 0 till i<=5 with x
        Display "Hello World"
End of Loop
```

3.2.7 Operators

Operators being used in NewBee can be seen in Table 1.

Table 1: Operators of "NewBee"

Arithmetic	Conditional	Relational
+	>	And
−	<	Or
*	>=	Not
/	<=	
%	<>	
	=	

3.2.8 Comments

The language supports single and multi-line comments.

Single Line Comments:

Single line comments are written with the double slash (//) sign.

Code Example:

```
//this is a single line comment
```

Multiline Comment:

A multiline comment starts with the single forward slash in the beginning (/) and a single forward slash (/) at the end of the comment.

Code Example:

```
/ this is a multiline comment, us to
Start and end with the backslash/
```

3.2.9 Reserved Keywords

Following Table 2. provides the list of reserved words/strings. All words will be case-insensitive so that students need not bother with the case-sensitivity of keywords and do not face such errors. The reflection of making these words case-insensitive is not present (but only "If" is made case-insensitive) in the CFG (due to the page limitations).

Table 2: List of reserved words

Start	End	If
of	Function	Else
Program	Loop	While
From	Till	Do
Input	Display	With
And		Not
Return		

4 Grammar of the Language

The grammar of the language is as bellow: The words written in Cap-Bold are the Terminals and the others are the non-terminals.

START		Start of Program **nl STMT nl** End of program | Start of program **nl STMT nl** Function **FUN_STMT** End of program	
STMT	→	**INPUT_STMT**	**nl STMT** |
		OUTPUT_STMT	**nl STMT** |
		EXP_STMT	**nl STMT** |
		ASS_STMT	**nl STMT** |
		SEL_STMT	**nl STMT** |
		LOOP_ STMT	**nl STMT** |
		FUN_CAL_STMT	**nl STMT** |
		FUN_DEC_STMT	**nl STMT** |
		break |	
		(nl used for the new-line)	
IDENTIFIER	→	**ALPHABET IDENTIFIER** | **ALPHABET**	
ALPHABET	→	**a** | **b** | **c**| **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u**| **v** | **w** | **x** | **y** | **z** | **A** | **B**| **C**| **D**| **E**| | **F** | **G** | **H** | | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** | _	
CONSTANT	→	**DIGIT CONSTANT** | **DIGIT**	

(Continued)

DIGIT → **1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0**
STRING → **CHARACTER STRING | STRING**
CHARACTER *all character*
INPUT_STMT → Input **IDENTIFIER** | Input **STRING**
OUTPUT_STMT → Display **IDENTIFIER** | Display **STRING**
ASS_STMP → **IDENTIFIER = FUN_CALL_STMT |**
EXP-STMT |
IDENTIFIER = EXP-STMT |
IDENTIFIER = ALPHABET |
IDENTIFIER = CONSTANT
EXP-STMT → **EXP-STMT + EXP-STMT |**
EXP-STMT - EXP-STMT |
EP-STMT * EP-STMT |
EP-STMT/EP-STMT |
(EXP-STMT) |
IDENTIFIER |
CONSTANT
SEL_STMT → **IF LOG_STMT nl STMT nl** End of If |
IF LOG_STMT nl STMT nl ELSE_STMT nl End of If
IF → If | IF | if | iF
ELSE_STMT → **ELSE nl STMT**
LOG_STMT → **COND_STMT LOG_OP COND_STMT LOG_STMT |**
COND_STMT LOG_OP COND_STMT |
LOG_STMT
LOOP_STMP → **WHILE_LOOP | DO-WHILE_LOOP | FROM-TILL_LOOP**
COND_STMT → **EXP-STMT COND_OP EXP_STMT |**
EXP-STMT COND_OP EXP_STMT |
EXP-STMT COND_OP EXP_STMT |
EXP-STMT |
IDENTIFIER |
CONSTANT
COND_OP → < | > | >= | <= | = |<>
LOG_OP → And | Or | Not
FROM-TILL_LOOP → From **ASS_STMT** till **LOG_STMT** |
From **ASS_STMT** till **LOG_STMT** with **CONSTANT |**
From **ASS_STMT** till **LOG_STMT** with **IDENTIFIER**
WHILE_LOOP → While **LOG_STMT nl STMT nl** End of Loop |
While **COND_STMT nl STMT nl** End of Loop
DO-WHILE_LOOP → Do **nl LOG_STMT nl** While **EXP_STMT nl** End of Loop |
Do **nl STMT nl While COND_STMT nl** End of Loop
FUN_CALL_STMT → **IDENTIFIER (ARG_LIST) nl |**
IDENTIFIER () nl
ARG_LIST → **ARG , ARG_LIST |**
ARG

(Continued)

ARG	→	**EXP_STMT |** **IDENTIFIER |** **CONSTANT**
FUN_DEC_STMT	→	Function **FUN_DECL** End of Function **FUN_DEC_STMT**
FUN_DECL	→	**IDENTIFIER (PARA_LIST) nl STMT nl** Return **IDENTIFIER |** **IDENTIFIER (PARA_LIST) nl STMT nl** Return **CONSTANT |** **IDENTIFIER () nl STMT nl** Return **IDENTIFIER |** **IDENTIFIER () nl STMT nl** Return **CONSTANT**
PARA_LIST	→	**IDENTIFIER , PARA_LIST |** **IDENTIFIER**

5 Language Limitations

In order to keep language simple and easy to understand for the newbies, few leverages are restricted. Details can be seen as follows:

- It is a procedural language and doesn't support Object-Oriented Programming (OOP) features like Inheritance, Encapsulation, and Polymorphism
- Only alphabets—lower case and upper case and "_" is allowed for identifiers
- An identifier with the name of keywords should not be created (the compiler can apply this restriction)
- Function call passing in function arguments is restricted
- Every building block, like loops and if-else structures, start and end with keywords that are close to common sense and English
- Language does not support Arrays and Switch statements
- Language does not support any external libraries.

6 Sample Program

Some sample programs are given below:

6.1 Program to Add Two Numbers

```
Start of program
   Display "Enter value:"
   Input a
   Display "enter the value:"
   Input b
   c=a+b
   Display "Result": "
   Display c
End of program
```

Output:

```
    Enter value: 5
```

```
Enter the value: 5
Result: 10
```

6.2 *Program to Find Factorial of a Number*

```
Start of program
   fact = 1
   Display "Enter Number: "
   Input num
   From a=1 till a<=num
           fact=fact*a
   End of Loop
   Display "Factorial of Given Number is ="
   Display fact
End of program
```

Output:

```
   Enter Number: 3
   The Factorial of the Given Number is = 6
```

6.3 *Program to Find Prime Numbers*

```
Start of program
   Display "Enter a positive integer: "
   Input n
      flag = 0
      From i = 2 till i<= n And flag = 0 with 1
            If n%i = 0
            flag = 1
            End of If
   End of Loop
   If n = 1
            Display "1 is neither prime nor composite."
   Else
            If flag = 0
            Display "given number is a prime."
    Else
            Display "given number is composite."
   End of If
End of program
```

Output:

Enter a positive integer: 7

the given number is a prime.

7 Results and Conclusion

This study has presented a grammar for a language for beginners, novice programmers, and the non-cs major student called NewBee. The language proposes a simple and close English sentence structure and syntax to provide ease and comprehension to the beginners and novice programmers. The language is proposed using a systematic approach with feature identification, context-free grammar, and sample programs. Additionally, this paper conducted a short survey and examined the satisfaction of the experts. In this survey, 17 experts (with a minimum of 5 years of industrial experience or teaching experience in multiple languages) were given forms containing the programs (elaborating syntax) and their satisfaction on a 10-Likert scale (1 to 10) was recorded, where one indicated lowest satisfaction and ten indicated the highest satisfaction. The average satisfaction appeared to be 93% among different users of the language (Keeping in mind that this language is for the novice, newcomers and students who do not have computer science as their major).

8 Future Work

This study can further be extended by incorporating more new and old language to make things more and more simple for the programmers

Acknowledgement: The authors would like to express their most profound gratitude towards, Mr Rana Muhammad Ijaz and Mr Sikandar Hayyat for their valuable time and efforts in helping us.

Funding Statement: This material is based upon the work supported by the startup fund provided to Dr. Saira Anwar by Texas A&M University, College Station, USA. Any opinions, findings, conclusion, or recommendations expressed in this material do not necessarily reflect those of Texas A&M University.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. L. Guzman, "What is human-machine communication, anyway," in *Human-machine Communication Rethinking: Communication Technology and Ourselves*, Peter Lang, Book, New York, USA, vol. 1, pp. 1–28, 2018.

[2] M. Soeken, T. Haener and M. Roetteler, "Programming quantum computers using design automation," in *2018 Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Dresden, Germany, pp. 137–146, 2018.

[3] D. Johnson and M. Ketel, "IoT: Application protocols and security," *International Journal of Computer Network & Information Security*, vol. 11, no. 4, pp. 1–8, 2019.

[4] K. Vinall and E. A. Hellmich, "Down the rabbit hole: Machine translation, metaphor and instructor identity and agency," *Second Language Research & Practice*, vol. 2, no. 1, pp. 99–118, 2021.

[5] N. G. S. S. L. States, "Next generation science standards: For states, by states," Washington, DC, USA, Book, 2013.

[6] S. Olson, "Grand Challenges for Engineering: Imperatives, Prospects and Priorities: Summary of a Forum," National Academies Press, Washington, DC, USA, 2016.

[7] A. Juškevičiene, G. Stupuriene and T. Jevsikova, “Computational thinking development through physical computing activities in STEAM education,” *Computer Applications in Engineering Education*, vol. 29, no. 1, pp. 175–190, 2021.

[8] D. Proctor, “The social production of internet space: Affordance, programming and virtuality,” *Communication Theory*, vol. 31, no. 4, pp. 593–612, 2021.

[9] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha *et al.,* “Ranking programming languages by energy efficiency,” *Science of Computer Programming*, vol. 205, pp. 102609–102639, 2021.

[10] O. Grljević and Z. Bošnjak, “Sentiment analysis of customer data,” *Strategic Management*, vol. 23, no. 3, pp. 38–49, 2018.

[11] F. Del Bonifro, M. Gabbrielli, A. Lategano and S. Zacchiroli, “Image-based many-language programming language identification,” *PeerJ Computer Science*, vol. 7, pp. e631–655, 2021.

[12] A. M. Abubakar and A. A. Mustapha, “Newton’s method cubic equation of state C++ source code for iterative volume computation,” *International Journal of Recent Engineering Science*, vol. 8, no. 3, pp. 12–22, 2021.

[13] J. -S. Lee, Y. -W. Su and C. -C. Shen, “A comparative study of wireless protocols: Bluetooth, UWB, ZigBee and Wi-Fi,” in *IECON 2007–33rd Annual Conf. of the IEEE Industrial Electronics Society*, Teipei, Taiwan, pp. 46–51, 2007.

[14] J. Peterson, “Speaking ability progress of language learners in online and face-to-face courses,” *Foreign Language Annals*, vol. 54, no. 1, pp. 27–49, 2021.

[15] S. G. Kochan, “Programming in C Third Edition,” Book, Developer’s Library, Indianapolis, Indiana, 2021.

[16] X. Chen, D. Song and Y. Tian, “Latent execution for neural program synthesis beyond domain-specific languages,” *Advance in Neural Information Processing Systems*, vol. 34, pp. 1–13, 2021.

[17] D. Pollak, V. Layka and A. Sacco, “DSL and Parser Combinator,” in *Beginning Scala 3*, Berkeley, California: Springer, pp. 237–245, 2022.

[18] S. Höppner, T. Kehrer and M. Tichy, “Contrasting dedicated model transformation languages versus general purpose languages: A historical perspective on ATL versus java based on complexity and size,” *Software and Systems Modelling*, vol. 21, pp. 1–33, 2021.

[19] K. Faldu, A. Sheth, P. Kikani and H. Akbari, “KI-BERT: Infusing knowledge context for better language and domain understanding,” arXiv Prepr. arXiv2104.08145, vol. 2, pp. 1–10, 2021.

[20] R. Liu, M. Gao, S. Ye and J. Zhang, “IGScript: An interaction grammar for scientific data presentation,” in *Proc. of the 2021 CHI Conf. on Human Factors in Computing Systems*, Yokohama, Japan, pp. 1–13, 2021.

[21] A. Van Deursen, P. Klint and J. Visser, “Domain-specific languages: An annotated bibliography,” *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[22] S. Erdweg, P. G. Giarrusso and T. Rendel, “Language composition untangled,” in *Proc. of the Twelfth Workshop on Language Descriptions, Tools and Applications*, New York, USA, pp. 1–8, 2012.

[23] N. Vasudevan and L. Tratt, “Comparative study of DSL tools,” *Electronic Notes Theoretical Computer Science*, vol. 264, no. 5, pp. 103–121, 2011.

[24] L. M. do Nascimento, D. L. Viana, P. A. S. Neto, D. A. Martins, V. C. Garcia *et al.,* “A systematic mapping study on domain-specific languages,” in *the Seventh Int. Conf. on Software Engineering Advances (ICSEA 2012)*, Lisbon, Portugal, pp. 179–187, 2012.

[25] M. Mernik, “Domain-specific languages: A systematic mapping study,” in *Int. Conf. on Current Trends in Theory and Practice of Informatics*, Limassol, Cyprus, pp. 464–472, 2017.

[26] T. Kosar, S. Bohra and M. Mernik, “Domain-specific languages: A systematic mapping study,” *Information and Software Technology*, vol. 71, pp. 77–91, 2016.

[27] J. Tanha, Y. Abdi, N. Samadi, N. Razzaghi and M. Asadpour, “Boosting methods for multi-class imbalanced data classification: An experimental review,” *Journal of Big Data*, vol. 7, no. 1, pp. 1–47, 2020.

[28] M. Mernik, J. Heering and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[29] S. Erdweg, T. V. D. Storm, M. Volter, R. Bosman, W. R. Cook *et al.,* “The state of the art in language workbenches,” in *Int. Conf. on Software Language Engineering*, Indianapolis, USA, pp. 197–217, 2013.

[30] S. Erdweg, T. V. D. Storm, M. Volter, R. Bosman, W. R. Cook *et al.,* "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structure*, vol. 44, pp. 24–47, 2015.
[31] P. N. Johnson-Laird, M. Bucciarelli, R. Mackiewicz and S. S. Khemlani, "Recursion in programs, thought, and language," *Psychonomic Bulletin & Review*, vol. 29, pp. 430–454, 2022.
[32] S. Olson, "Grand Challenges for Engineering," Washington, D.C.: National Academies Press, Book, 2016.
[33] J. Hartmann, J. Huppertz, C. Schamp and M. Heitmann, "Comparing automated text classification methods," *International Journal of Research in Marketing*, vol. 36, no. 1, pp. 20–38, 2019.
[34] H. M. Gualandi, "*The Pallene Programming Language*," Ph. D. Dissertation. Pontifcia Universidade Católica do Rio de Janeiro, 2020.