© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712881

Developing Tindar: A Native Arabic Programming Language

ABD ALROOF S. ALMOGADMI[1], SUMAIA A. ELTOMI [2], MURAD S. BAGHNI [3]
[1,2,3] *Department, of Electronic Engineering, College of Industrial Technology, Misurata, Libya*

Abstract- ***This paper introduces Tindar, a new Arabic programming language designed to ease programming education and computing interaction for Arabic speakers. Tindar features a simple, extensible structure supporting fundamental programming concepts, object-oriented programming, and functional programming features. Its architecture comprises a parser, syntactic analyzer, code generator, and virtual machine. Implemented in Rust for speed and security, the Tindar compiler outperforms Python in performance tests and is cross-platform compatible. This work emphasizes the significance of culturally and linguistically relevant computing interfaces, addressing the linguistic and technical challenges of integrating Tindar with modern development environments and offering practical solutions.***

Index Terms- ***Arabic programming language, Compiler design, Extensibility, Language integration.***

I. INTRODUCTION

Programming languages constitute a fundamental pillar of modern technological infrastructure, serving as the primary tool through which developers create software and applications that permeate nearly every aspect of daily life. Since the advent of computing, programming languages have evolved rapidly in response to growing user demands. However, this evolution has been largely confined to English-based programming languages, leaving other linguistic communities marginalized in this critical domain [1]. Despite advancements in computing speed and storage, the fundamental principles of language design remain consistent. Although language design is a broad field, the methodologies and resulting compiler structures, from early COBOL translators to modern JavaScript compilers, share common traits [2][3]. Despite continuous technical advancements, the persistent dominance of English in most programming languages has created a significant linguistic and cognitive gap for non-English-speaking communities. This growing disparity has negatively impacted learning accessibility and inclusivity, particularly in the Arab world, where Arabic-language educational resources and localized programming tools remain relatively scarce and underdeveloped [3]. Developing an Arabic programming language is more than translation—it's a strategic step to empower Arabic speakers to engage with computing in their native language. It reinforces linguistic identity in tech, promotes Arabic digital content, and supports programming education in Arabic-speaking institutions, especially at foundational levels [4].

II. LANGUAGE EVOLUTION

Since the early days of computing, programmers have developed numerous programming languages. Despite exponential improvements in computer speed and storage, the core principles of language design remain largely consistent. While language design is broad, its methodologies are relatively limited, and compilers, from early COBOL to modern JavaScript compilers, share many similarities.

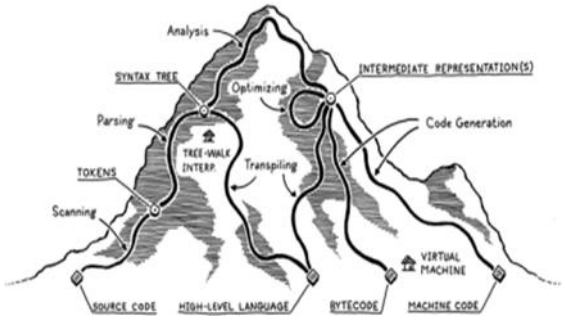

Fig 1. Guide to Creating Languages.

Figure 1 depicts source code transforming into machine code, illustrating compiler evolution and the lasting principles of modern compiler architecture [5].

1. Arabic Programming Gap

The established translation from source code to machine code reveals a structural bias: English's prevalence in programming hinders Arabic speakers.

© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712881

- Language Barrier: Many Arabic-speaking beginners face challenges with programming languages rooted in English, hindering their entry into the field [6].
- Structural Complexity: Some existing Arabic programming languages suffer from complex syntax, which makes it hard for beginners to learn and grasp fundamental concepts.
- Limited Integration: Current Arabic-based languages face difficulties integrating with other tools and languages, complicating the development process and increasing challenges for programmers.
- Low Efficiency: Some languages lack performance and usability, negatively affecting program execution speed and consuming system resources excessively.

2. Compiler Workflow Stages

The Arabic programming gap extends beyond language to encompass the process of transforming source code into executable instructions via a compiler's sequential stages [5].

A. Pre-processing

Pre-processing modifies source code prior to compilation, managing directives like C/C++ #include and #define for inclusion or macros. Rust employs advanced declarative macros, while Python and JavaScript limit pre-processing to maintain simplicity and interactivity.

B. Lexical Analysis

Lexical analysis scans source code, converting character streams into tokens (keywords, identifiers, numbers, symbols) while ignoring spaces and comments. These tokens form the basis for syntactic analysis [5][7]. Figure 2 shows the process.

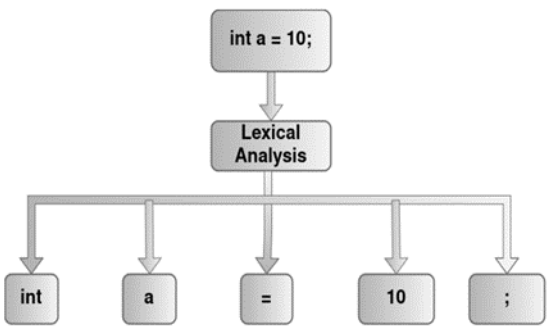

Fig 2. Lexical analysis process

C. Syntax Analysis (Parser)

Syntax analysis, or parsing, checks if token sequences from lexical analysis conform to context-free grammar. If syntactically correct, the parser produces an Abstract Syntax Tree (AST) that represents the program's logical structure, omitting elements like parentheses and semicolons that are irrelevant for subsequent stages such as code generation. The syntax tree is shown in figure 3 [8].

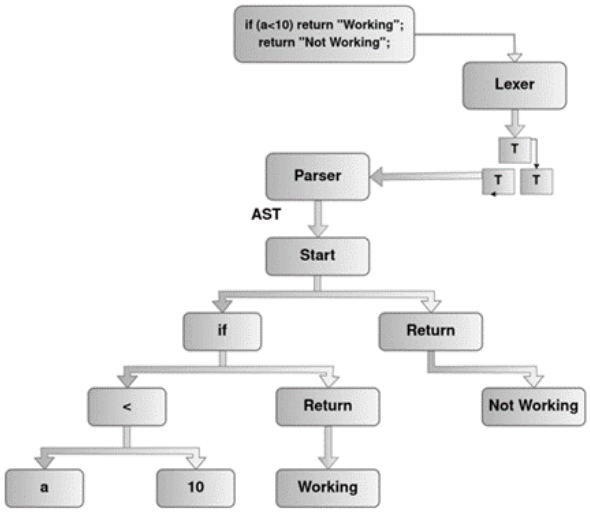

Fig 3. Syntax tree

D. Code Generation

In the final stage, code generation converts the Abstract Syntax Tree (AST) into executable instructions. Designers may target native code for speed or virtual code for portability. Bytecode, introduced in the 1960s, ensures cross-platform execution without recompilation, addressing portability concerns.

E. Run Time

After compilation, the program enters the run-time phase, executing either natively through the OS or within a virtual machine. This phase provides services like garbage collection and error handling. Compiled languages like Go embed run-time code, while interpreted languages like Java and Python depend on virtual machine run-time environments.

3. From Source to Execution

Programming languages employ diverse methods to convert source code into executable programs. Virtual Machines offer portability and flexibility, but sacrifice performance. Languages like C rely on compatible compilers for portability. Tree-walk

interpreters facilitate static analysis and early error detection, though their slow speed restricts them to smaller projects. Transpilers enable cross-platform development by converting code between languages, as demonstrated by TypeScript's conversion to JavaScript [5].

III. RELATED WORK

This section reviews key academic studies on programming language development that inform our approach and provide foundational insights.

A. Arabic Languages Development

The study in [9] identified the lack of Arabic programming languages as a key barrier for Arabic-speaking students. To address this, the researchers developed Alf, an Arabic programming language that translates into C++ and includes essential programming concepts. Their findings indicate that using Arabic-based programming languages enhances comprehension, facilitates native-language learning, and promotes technical education within Arabic-speaking communities.

Phoenix, an object-oriented Arabic programming language using Arabic syntax and vocabulary, was detailed in [10]. Its compiler includes a pre-processor, scanner, parser, semantic analyzer, code generator, and linker. Comparative experiments showed Phoenix's strengths over C# in functions, loops, and arithmetic, but its portability is limited to Windows.

APL, an Arabic educational programming language, uses GPT-based Large Language Models to translate Arabic code into Python via prompt engineering [11]. The system—comprising a Python library, learning interface, and runtime environment—demonstrates that LLMs can function as natural-language compilers. By reducing language barriers, APL enhances self-learning and makes programming concepts more accessible to Arabic speakers.

B. Other Languages Development

The "Hela" language, designed to resemble Sinhala, aimed to simplify programming education [12]. Its Java compiler comprised command-line tools, a preprocessor, lexer, and parser. A user survey indicated that Hela was more beginner-friendly and had greater natural language resemblance but was less effective for general programming and had less clear loop structures.

In [13] a recent study introduced BASH-A, a customizable educational platform that uses native languages, such as Bengali, to teach programming. The study found that BASH-A improved conceptual understanding and reduced coding errors, demonstrating the benefits of linguistic personalization for beginner programmers from non-English-speaking backgrounds.

TPD is an experimental programming language, inspired by Turkish and supporting both imperative and functional paradigms, designed for Turkish-speaking high school students and novice programmers [14]. Its integrated development environment compiles TPD code into Java, leveraging users' native language proficiency to ease the learning of fundamental programming concepts in education.

IV. PROPOSED APPROACH

Tindar, initially designed for a simple Arabic terminal emulator, was later generalized to expand its use. Its syntax, inspired by C and JavaScript, facilitates ease of transition for users familiar with those languages. Features were prioritized and implemented based on this broader applicability. Tindar's core architecture, aligned with its development stages, is illustrated in fig 4.

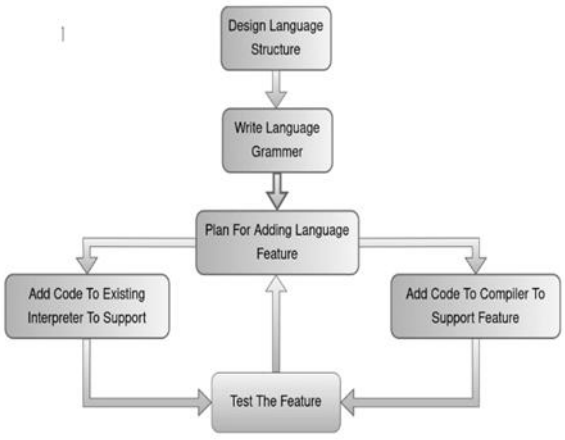

Fig 4. System architecture

Tindar execution starts from the main function ()ابدا which organizes program flow and calls data processing functions. The function contains the

© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712881

اطبعس statement that prints "السلام عليكم" with a new line.

```
دالة ابدا () { اطبعس("السلام عليكم")؛ }
```

1. Variables and Data Classification

For maintainable programs, understanding variables and data types is key. Tindar supports both dynamic and static typing, classifying data types into basic types in table 1 and compound types in table 2.

Table 1. Primitive Data

Type	Description	Example
Number	Integer and float	عرف العدد: رقم = 10؛
Boolean	True or False	عرف الحالة: منطقي = صح؛
Text	Strings	عرف الرسالة: نص = "السلام عليكم!"؛
General	Accepts any value	عرف القيمة: عام = 10؛

Table 2. Composite data

Type	Description	Example
Array	Contains mixed data types	عرف الأعداد: مصفوفة<رقم> = [1, 2, 3]؛
Entity	Combines data and functions	كيان موظف {اسم، عمر}؛
Set	Mixed data types	عرف م :(رقم، نص، رقم) = (10، "مرحبا"، 10)؛
List	Key-Value pairs	عرف ل= قائمة{ "محمد": 20، "علي": 30 }؛

2. Variable Declaration Methods

Variable declaration in Tindar can be performed in two ways, as demonstrated in table 3.

Table 3. Variable Declarations

method	Description	Example
Explicit Declaration	Data type is stated directly.	عرف ع: رقم = 10؛
Implicit Declaration	Compiler infers type as general.	عرف ع = 10؛
method	Description	Example
Explicit Declaration	Data type is stated directly.	عرف ع: رقم = 10؛
Implicit Declaration	Compiler infers type as general.	عرف ع = 10؛

3. Key Concepts in Tindar

Tindar introduces a set of core principles that constitute the structural foundation of modern programming languages. It emphasizes variable declaration in multiple forms and offers diverse arithmetic, logical, and bitwise operators to ensure computational efficiency. The language supports dynamic control flow through conditional constructs (لو, والا) and iterative loops (بينما). Moreover, it employs reusable functional units, defined using the keyword (دالة) and returning values via (أعد). Finally, Tindar provides a flexible architecture for creating data structures across entities, defined by the keyword (كيان), followed by the object name and a list of its properties, allowing data structures to be organized in a systematic and extensible manner.

V. TINDAR IMPLEMENTATION

Tindar was developed in Rust due to its speed, memory safety, and modern compiler. Rust's combination of low-level efficiency and high-level safety, enforced by its ownership model, type system, and tools like Cargo, provides a robust foundation for a reliable and efficient compiler. The implementation process is detailed in fig 5.

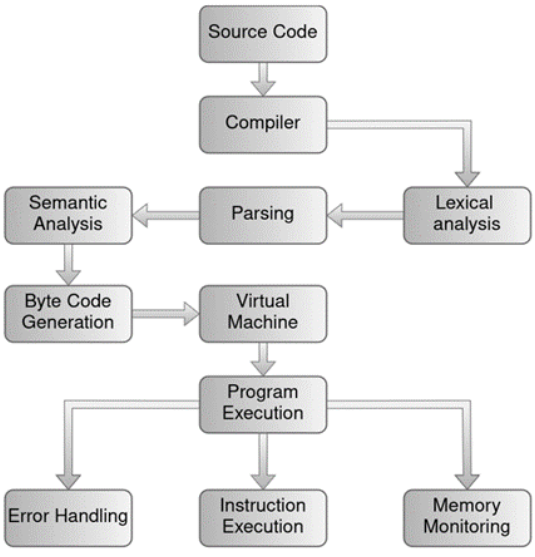

Fig 5. Execution strategy

A. Compiler

The compiler processes Tindar source code in the following stages:

© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712881

- Lexical Analysis

The lexical analyzer scans source code, verifying each character (except those in quotations) conforms to Arabic. Valid characters produce lexical units; otherwise, an error is reported.

```
دالة ابدا(){ عرف ز؛ }
```

```
pln_compiler/examples/and.pln:2:5:[j]حرف غير مدعوم
```

- Syntactic Analysis

The syntactic parser checks the grammatical correctness of Tindar sentences and displays an error message when a syntactic error is detected.

```
دالة ابدا(){ اطبع(لو)؛ }
```

```
pln_compiler/examples/and.pln:3:8: expected:primary expr found:If
```

Tindar parser uses top-down parsing with Pratt parsing for efficient expression handling and to avoid left recursion issues. Fig 6 demonstrates the Pratt parsing algorithm's application to mathematical precedence [5][15].

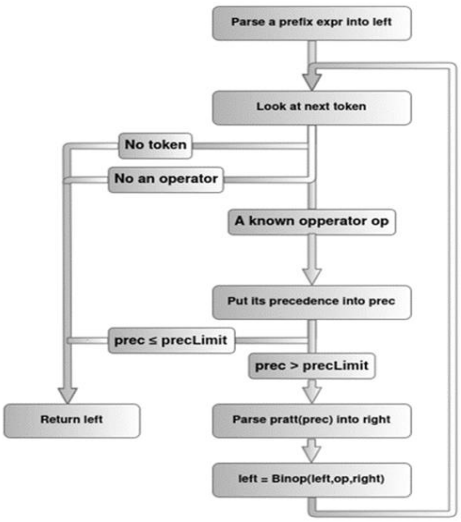

Fig 6. Pratt parsing algorithm

- Semantic Analysis

During semantic analysis, the compiler checks type consistency to ensure valid operations, such as preventing the addition of numbers to strings.

```
دالة ابدا(){ اطبع("سلام" + 10)؛ }
```

```
pln_compiler/examples/and.pln:3:7:lhs and rhs of expression should have type of int when using + [for now] found(lhs:str[7],rhs:int)
```

It verifies that variables are declared before use and communicates with the virtual machine to generate instructions. Rust code defines operations like addition in this phase. Finally, a virtual entry function invokes the main function and terminates execution.

B. Virtual Machine

The virtual machine is a critical component for performance and efficiency, executing instructions step-by-step. Rust code handles arithmetic operations and manages runtime issues.

C. Runtime

The virtual machine's operation hinges on a set of runtime services:

- Data Representation

Tindar uses NaN-boxing for efficient data representation on 64-bit systems, a common strategy in dynamic languages like JavaScript Fig 7 [5]. For other word sizes, it employs tagged counters, which utilize extra bits to distinguish data types within a unified memory space.

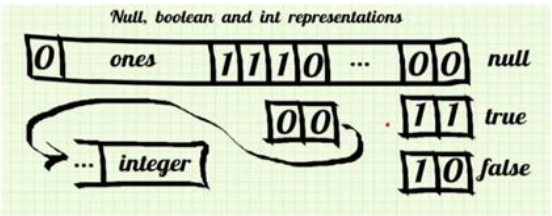

Fig 7: Nan-boxing method

- Memory Management

Mark-and-Sweep garbage collector is used to manage memory efficiently.

- Rust Integration

To enable interoperability, Rust code is compiled into dynamic libraries that the virtual machine loads at runtime.

VI. RESULTS AND DISCUSSION

This section presents and discusses the experimental results evaluating Tindar's performance.

1. Code testing

Tindar's versatility is demonstrated through shape-drawing, GUI graphics, and object-oriented programming examples. A simple shape-drawing program, as seen in Figure 8, uses textual characters

© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712881

to render a left-aligned triangle, illustrating basic programming concepts. Figure 9 showcases Tindar's GUI capabilities, rendering geometric shapes on a colored canvas. Finally, Figure 10 confirms a successful object-oriented linked list implementation with dynamically constructed node relationships. These examples highlight Tindar's suitability for teaching and application development, supporting structured, graphical, and object-oriented paradigms.

Fig 8. Drawing a triangle

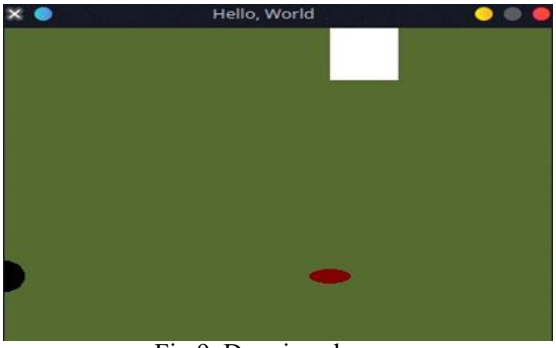

Fig 9. Drawing shapes

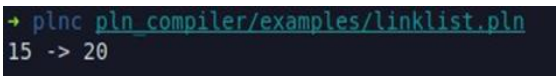

Fig 10. Linked list execution

2. Portability Testing

Tindar's portability was tested using a Fibonacci function to measure performance across operating systems. Table 4 shows the execution speeds on different platforms

Table 4. Run-Time by Device and OS

Device	Processor	OS	Run Time (s)
Toshipa	i3 G3	Linux	20 – 26
Toshipa	i3 G3	Win 10	33
Hp	i5 G3	Win10	34 – 32
Lenovo	i3 G2	Linux	23- 20

3. Syntax Comparison

Tindar's syntax resembles that of C-family languages Table 5, potentially easing adoption and learning.

Table 5. Tindar vs General Languages: Syntax

Feature	Tindar	Python / Rust / G0
Declaration	عرف س = 6؛	x := 6 Let x = 6; x = 6
Conditions	لو س > 0{..}	if x > 0 {..} if x > 0 {..} if x > 0:
Loops	بينما س > 5{..}	for i:=0;i<5;i++{} for i in 0..5{..} for i in Range (5) {}
Arrays	عرف م = [1,2]؛	arr := []int{1, 2} let arr = [1,2]; arr = [1,2]
Comments	#تعليق	//comment //comment #comment

4. Performance

Tindar's performance was benchmarked against Python using a Fibonacci sequence [16], with execution times presented in table 6.

Table 7. Performance: Tindar vs Python

Device	Processor	OS	Run Time (s)	
			Tindar	Python
Toshipa	i3 G3	Linux	20 – 26	45
Toshipa	i3 G3	Win10	33	51
Hp	i5 G3	Win10	32 – 34	52 – 47

VII. CONCLUSION

The Tindar programming language showed that usability, scalability, and platform compatibility are as crucial to success as performance. Its features, like automatic memory management and cross-platform support, improved accessibility for new programmers. Effective design demands not only efficient syntax and algorithms but also structural elements that enhance the developer experience. Sustaining Tindar's growth requires developing robust libraries, improving tool integration, and promoting educational adoption to foster a strong user community.

REFERENCES

[1] M. M. Khalatia and T. A. H. Al-Romany, "Artificial Intelligence Development and Challenges (Arabic Language as a Model)," International Journal of Innovation, Creativity and Change, vol. 13, no. 5, pp. ,2020. [Online]. Available: www.ijicc.net

[2] Thain, "Introduction to Compilers and Language Design", 2nd ed. Notre Dame, IN, USA: Douglas Thain, 2023. ISBN: 979-8-655-18026-0.[Online].Available: http://compilerbook.org

[3] Z. Alyafeai and M. S. Al-Shaibani, "ARBML: Democratizing Arabic Natural Language Processing Tools," in *Proc. 2nd Workshop on NLP Open Source Software (NLP-OSS)*, pp. 8–13, Virtual Conference, Nov. 19, 2020. [Online].Available: https://aclanthology.org/2020.nlposs-1.2/

[4] M. A. Mohammed, S. H. M. Zeebaree, S. Mostafa, and M. K. A. Ghani, "Designing and Implementing an Arabic Programming Language for Teaching Pupils," J. Southwest Jiaotong Univ., vol. 54, no. 3, pp, Jun. 2019. [Online]Available: https://www.researchgate.net/publication/333858707

[5] R. Nystrom, *Crafting Interpreters*, 1st ed. [Online]. Available: https://craftinginterpreters.com. © 2015–2020.

[6] A. ElSabagh, S. S. Azab, and H. A. Hefny, "A comprehensive survey on Arabic text augmentation: approaches, challenges, and applications," *Neural Comput. Appl.*, vol. 37, pp. 7015–7048, 2025, doi: 10.1007/s00521-025-11002-z.[Online]Available: https://doi.org/10.1007/s00521-025-11020-z

[7] W. Kania and R. Wajman, "CKRIPT: A new scripting language for web applications," *IAPGOŚ*, vol. 2, pp. 2022.[Online]Available: (PDF) CKRIPT: A NEW SCRIPTING LANGUAGE FOR WEB APPLICATIONS

[8] GeeksforGeeks, “Introduction to Syntax Analysis in Compiler Design,” GeeksforGeeks, 27-Aug-2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/introduction-to-syntax-analysis-in-compiler-design/ [Accessed: 29-Oct-2025].

[9] H. H. A. Razaq, A. S. Gaser, M. A. Mohammed, E. T. Yassen, S. A. Soltana, S. R. M. Zeebaree, D. A. Mahmood, M. K. A. Ghani, and R. N. Farhan, "Designing and implementing an Arabic programming language for teaching pupils," *J. Southwest Jiaotong Univ.*, vol. 54, no. 3, pp2019, [Online]Available: https://jsju.org/index.php/journal/article/view/289

[10] Y. Bassil, “Phoenix – The Arabic Object-Oriented Programming Language,” International Journal of Computer Trends and Technology (IJCTT), vol. 67, no. 2, pp. 1–7, Feb. 2019. [Online]. Available: https://arxiv.org/pdf/1907.05871

[11] S. Sibaee, O. Najar, L. Ghouti, and A. Koubaa, “LLMs as Scribe for Arabic Programming Language,” *arXiv preprint arXiv:2403.10876v1 [cs.SE]*, Prince Sultan University, KSA, Mar. 2024. [Online]. Available: https://arxiv.org/abs/2403.16087

[12] R. Yasasri and D. Karunarathna, “Helaa: A Sinhala Language–Based Programming Language,” Proc. of the International Conference, University of Colombo School of Computing, Sri Lanka, Aug. 2023. [Online]. Available:https://www.researchgate.net/publication/372946071

[13] VIJAYGANESH, S. Nandwana, and R. Kumar, “Breaking the Language Barriers of Programming: An All-Inclusive and Personalizable Programming Platform — BASH-A,” International Journal of Recent Research and Review, vol. XVII, no. 2, pp. xx–xx, Jun. 2024. [Online]. Available: https://www.ijrrr.com/papers/June2024/Breaking-the-Language-Barriers-of-Programming.pdf

[14] S. Tutar, C. Bozsahin, and H. Oguztuzun, “TPD: An Educational Programming Language Based on Turkish Syntax,” Dept. of Computer Engineering, Middle East Technical University, Ankara, Turkey. [Online]. Available: https://www.researchgate.net/publication/301687920

[15] M. Janiczek, “Demystifying Pratt Parsers,” Blog post, Jul. 3, 2023. [Online]. Available: https://martin.janiczek.cz/2023/07/03/demystifying-pratt-parsers.html

[16] S. M. Farooq and S. H. Shabbeer Basha, “A Study on Fibonacci Series Generation Algorithms,” in Proc. 3rd Int. Conf. on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, Jan. 2016. [Online]. Available: https://www.academia.edu/96152241