

Article

Ring: A Lightweight and Versatile Cross-Platform Dynamic Programming Language Developed Using Visual Programming

Mahmoud Samir Fayed *  and Yousef A. Alohalı 

College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia

* Correspondence: msfayed@ksu.edu.sa

Abstract: New programming languages are often designed to keep up with technological advancements and project requirements while also learning from previous attempts and introducing more powerful expression mechanisms. However, most existing dynamic programming languages rely on English keywords and lack features that facilitate easy translation of language syntax. Additionally, maintaining multiple implementations of the same language for different platforms, such as desktops and microcontrollers, can lead to inconsistencies and fragmented features. Furthermore, they usually do not use visual programming to fully implement the compiler and virtual machine. In this research paper, we introduce Ring—a dynamically-typed language with a lightweight implementation. However, it boasts several advantages, including a rich and versatile standard library and direct support for classes and object-oriented programming. The Ring language offers customization features. For instance, it allows easy modification of the language syntax multiple times, enabling programming by writing code using Arabic, English, or other keywords. Additionally, the language permits the creation of domain-specific languages through new features that extend object-oriented programming, allowing for specialized languages resembling CSS or Supernova. In the era of the Internet of Things, instead of creating another language implementation to support microcontrollers, the same Ring implementation allows us to create projects and applications for desktops, the web, WebAssembly, Android, or Raspberry Pi Pico. The Ring Compiler and Virtual Machine are designed using the PWCT Visual Programming language based on ANSI C. The visual implementation is composed of 18,945 components that generate 24,743 lines of code, which increases the abstraction level by approximately 23.5% and hides unnecessary details.



Citation: Fayed, M.S.; Alohalı, Y.A. Ring: A Lightweight and Versatile Cross-Platform Dynamic Programming Language Developed Using Visual Programming. *Electronics* **2024**, *13*, 4627. <https://doi.org/10.3390/electronics13234627>

Academic Editor: Porfirio Tramontana

Received: 15 October 2024

Revised: 21 November 2024

Accepted: 21 November 2024

Published: 23 November 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: programming language; visual programming; desktop; WebAssembly; IoT

1. Introduction

Programming languages play a crucial role in producing systems and applications. They serve as the means of communication between us and the computer, enabling control and the creation of software and applications. Initially, there was machine language, which allowed us to program by directly controlling the operations provided by the hardware. Soon, many programming languages evolved, each with different goals—such as ease of learning, specific domain usability, using new programming paradigms, performance improvement, security, portability, or achieving flexibility [1–4].

During the evolution of programming languages, a category known as dynamic programming languages emerged. Examples of such languages include Lisp, Smalltalk, Erlang, Python, Lua, and Julia, as demonstrated in Figure 1. These languages exhibit several features that defer determination and execution to runtime rather than compiling time. Notable characteristics include dynamic typing, flexible data structures, reflection, metaprogramming, and the ability to evaluate code from strings using functions like eval().

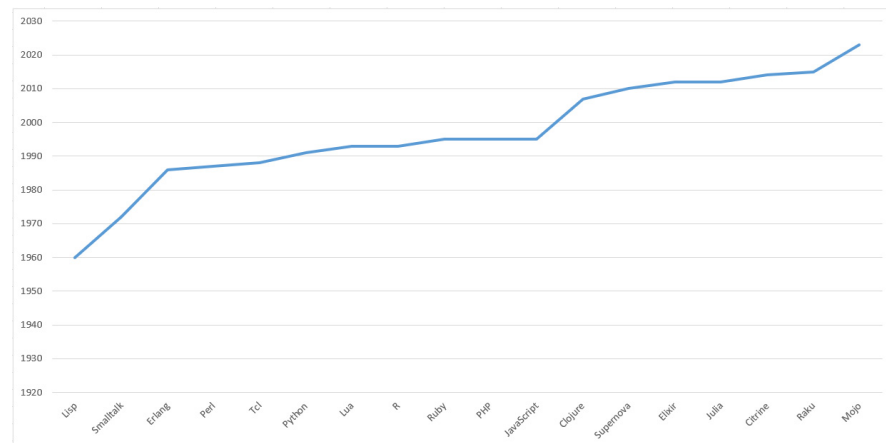


Figure 1. Some of the dynamic programming languages that were developed, starting in 1960.

Additionally, dynamic languages often provide a Read-Eval-Print-Loop (REPL) for interactive development. The overarching goal of these languages is to achieve simplicity, flexibility and reduced compile time. Ultimately, this speeds up the development cycle and facilitates the creation of project prototypes in less time [5–12].

Another category of programming languages is visual programming languages (VPLs). These languages use more than one dimension to create computer programs graphically through text, shapes, colors, etc. They have achieved notable usage in education through projects like Scratch [13–15]. Unlike the Scratch visual programming language, which enables children to create multimedia applications using a user interface in their native language, most dynamic programming languages rely on English keywords. Unfortunately, these dynamic programming languages lack features that facilitate easy translation of language syntax and libraries into other human languages [16–21].

While most visual programming languages are domain-specific, there are projects classified as general-purpose and applicable to a wide range of programming tasks. One such project is the Programming Without Coding Technology (PWCT) software, a visual programming language that supports code generation in multiple textual programming languages, including the C programming language [13]. Most of the popular dynamic programming language implementations are based on using textual programming languages like C, C++, etc. We assume that using visual programming to create the dynamic programming language compiler and virtual machine is possible and provides a more user-friendly implementation by avoiding syntax errors and increasing the abstraction level.

Dynamic programming languages as software products differ from one another in terms of design, syntax, semantics, paradigms, features, implementation, execution methods, libraries, tools, and supported platforms, resulting in variations in the domains where they are most suitable for use [22–31]. Some dynamic programming languages are specifically tailored for domains like R, MATLAB, and dBase [32–34]. On the other hand, some dynamic programming languages serve as general-purpose tools suitable for a wide range of tasks like Python [35–40]. Domain-specific languages are designed for a specific domain, and they can be classified into two main types. The first type is Internal/Embedded DSLs, which are embedded inside general-purpose languages and use their constructs, while the second type is External DSLs (Like CSS, SQL, Supernova, etc.), which use its syntax and semantics. Dynamic programming languages like Ruby could be used to create internal DSLs. However, these internal DSLs will not resemble external DSLs [41–44].

The Supernova dynamic programming language is a domain-specific language distributed with the PWCT Visual Programming language [45]. This language was developed over nine months to explore creating simple GUI applications using command-based syntax that looks natural, as demonstrated in Figure 2.

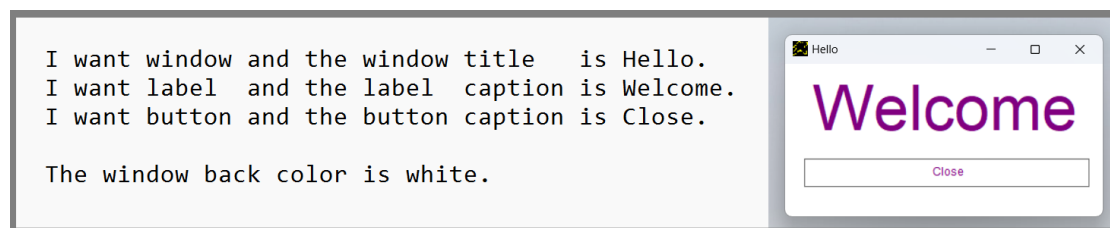


Figure 2. Using commands in the Supernova language to describe the application user interface.

After developing Supernova, we considered whether we could develop a new programming language that supports object-oriented programming and extend it with novel features to enable the development of embedded domain-specific languages resembling CSS and Supernova [13,45].

With the rise of popularity of the Internet of Things (IoT) [46–48], numerous projects—such as MicroPython and mRuby—have endeavored to leverage popular dynamic programming languages for embedded systems and microcontroller development. This requires developing a lightweight implementation and has led to the challenge of maintaining different implementations for the same programming language, where one implementation could miss features that exist in another implementation [49–52].

Also, while we find many dynamic languages used for web application development on the server side, JavaScript has dominated the scene as the language used at the front-end inside web browsers. With the emergence of WebAssembly (binary instruction format that can be executed by modern web browsers), it has become more practical to use several other languages within the browser. However, this led to the development of different language implementations to allow dynamic languages to fully benefit from this leap. Creating a new dynamic programming language in this era may necessitate considering this evolution to maximize its advantages [53–55].

Therefore, the primary aim of this work, as demonstrated in Figure 3 (all characteristics have the same weights), is to use visual programming to develop a lightweight and embeddable dynamic programming language and environment that facilitates easy and rapid translation of language syntax. Additionally, the language will empower developers to create embedded domain-specific languages (DSLs) resembling external DSLs like CSS and Supernova. The language will be multi-paradigm, providing direct support for object-oriented concepts such as classes, objects, encapsulation, and inheritance. Furthermore, our language will offer cross-platform support for desktop, web, WebAssembly, and 32-bit microcontrollers—all using a unified implementation. This implementation will be based on a visual programming design that generates ANSI C code for the bytecode compiler and the virtual machine. As a “batteries-included” language, it will come with rich libraries and tools, including an integrated IDE with a form designer.

The main contributions of this study are as follows:

- According to our knowledge, this is the first study to use visual programming to develop and maintain a compiler and virtual machine for a dynamic programming language over many years. We started development in September 2013, and the first version of the Ring language was released in January 2016. We have provided multiple releases each year to improve the design and respond to community feedback. Ring 1.21.2 was released in September 2024;
- Novel features that can extend the object-oriented programming paradigm, enabling the development of domain-specific languages that resemble CSS and Supernova. Additionally, Ring’s customization features, such as syntax modification, could support multiple languages (e.g., Arabic, English);
- The design and implementation of a dynamic programming language with broad cross-platform compatibility, featuring a lightweight implementation that still provides rich features.

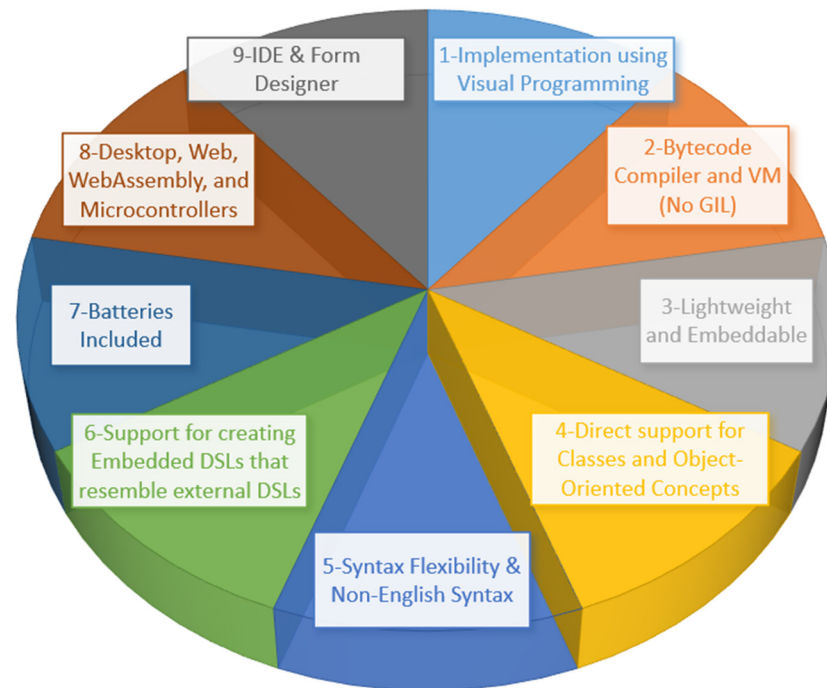


Figure 3. The key features of the proposed dynamic language and environment.

The rest of this paper is organized as follows: Section 2 discusses related works. Section 3 presents the materials, methods, and the proposed dynamic programming language. In Section 4, we present the results and evaluation of the visual implementation. Section 5 contains the discussion. Finally, concluding remarks and future work are provided in Section 6.

2. Related Work

The design of the proposed programming language is associated with various categories of dynamic programming languages. In Table 1, we present some of the different categories along with examples of dynamic programming languages that could fit within them. It is worth noting that some programming languages can be classified in more than one category. For instance, Ruby could be classified in both categories two and three, while a language like Tcl could fall into the first three categories.

Table 1. Some categories of dynamic programming languages and examples.

Refs.	Category	Examples
[22–25]	Lightweight and Embeddable	Lua, Squirrel, Wren, etc.
[29,36]	Comes with Ready-to-Use Libraries	Tcl, Perl, Python, etc.
[12,43]	Support creating Embedded DSLs	Lisp, Ruby, etc.
[11,34]	Comes with Powerful IDEs	Smalltalk, Visual FoxPro, etc.
[19,20]	Supporting Non-English Syntax	Supernova, Citrine, etc.
[32–34]	Domain-specific dynamic languages	R, xBase, etc.
[9,56,57]	Concurrency-oriented design	Erlang, Elixir, etc.
[6,58,59]	Comes with a focus on Performance	Julia, Mojo, etc.
[49–52]	Other implementations	MicroPython, mRuby, etc.

The first category is Lightweight and Embeddable languages [22–25], designed for ease of integration and minimal resource consumption, making them suitable for various environments. The second category includes Languages that Come with Ready-to-Use Libraries, offering a rich set of pre-built functionalities to accelerate development [29,36].

Support for Creating Embedded DSLs is the third category, providing flexibility for niche applications [12,43]. The fourth category is Languages that Come with Powerful

IDEs, which enhance the development experience through robust tools and features [11,34]. The fifth category encompasses Languages Supporting Non-English Syntax, broadening accessibility for developers worldwide [19,20]. Domain-Specific Dynamic Languages form the sixth category, tailored for fields to optimize efficiency and effectiveness [32–34].

The seventh category focuses on Concurrency-Oriented Design, managing simultaneous tasks crucial for high-performance applications [9,56,57]. Languages with a Focus on Performance make up the eighth category, ensuring rapid execution and responsiveness [6,58,59]. The ninth category, Other Implementations, includes different implementations of popular dynamic languages that focus on supporting microcontrollers or embedded systems. These implementations could be developed by the same team that created the original language (like mRuby) or by another team of developers who take the original language implementation and modify it by adding or removing features or changing the implementation (like MicroPython) [49–52].

In Table 2, we present the key features of our proposed language and its connections to some other dynamic programming languages. Deliberately, we chose at least one programming language from each relevant category (the first five categories) to provide a broader context for our language design. As we compare these languages, if a feature is absent in the basic distribution but available through external libraries, tools, or ongoing projects, we denote it with a (star) in the corresponding cell.

Table 2. The main features of the proposed dynamic programming language.

Criteria	Lua [24]	Python [60]	Ruby [61]	VFP [34]	Supernova [20]	Proposed Language (Ring)
Open Source	✓	✓	✓	X	✓	✓
Portable	✓	✓	✓	*	*	✓
Lightweight	✓	*	*	X	✓	✓
Embeddable	✓	✓	✓	X	X	✓
Dynamic Typing	✓	✓	✓	✓	✓	✓
Function like Eval()	✓	✓	✓	✓	X	✓
Classes Concept	*	✓	✓	✓	X	✓
Inheritance Concept	*	✓	✓	✓	X	✓
Private Attributes	*	*	✓	✓	X	✓
Batteries Included	*	✓	✓	✓	*	✓
IDE	*	✓	*	✓	*	✓
Form Designer	*	*	*	✓	*	✓
Non-English Syntax	*	*	*	*	✓	✓
Case insensitive	X	X	X	✓	✓	✓
1-based indexing	✓	X	X	✓	✓	✓
Change Keywords	X	X	X	X	X	✓
Internal DSL	✓	✓	✓	✓	X	✓
IDSL (Custom Syntax)	X	X	X	X	X	✓
Visual Implementation	X	X	X	X	✓	✓
VI Based on CPWCT	X	X	X	X	X	✓
Desktop	✓	✓	✓	✓	✓	✓
Web	✓	✓	✓	✓	X	✓
WebAssembly	*	✓	✓	X	X	✓
Microcontroller	*	*	*	X	X	✓
No-GIL	✓	*	*	X	X	✓
Register based VM	✓	X	X	X	X	X
Off-side rule	X	✓	X	X	X	X
xBase (Database DSL)	X	X	X	✓	X	X

The Lua programming language stands out due to its compact language features and efficient implementation, which are written in ANSI C. It serves as an embeddable language, making it suitable for integration into projects that require scripting capabilities through a relatively fast scripting language. Lua is commonly used for scripting in game development. Notably, Lua does not include the concept of classes but instead emulates

object-oriented concepts through its small and extensible language features. Additionally, Lua lacks a rich standard library [22–25]. Some programming languages, such as Squirrel and Wren, are designed to compete with Lua for game scripting. They use a different syntax based on braces and provide direct support for classes. Similar to Lua, they also lack a rich standard library. Although Wren seems no longer under active development, a smaller version of the language called Lox is used to introduce how interpreters are developed, allowing us to observe the usage of dynamic languages in introducing compiler and virtual machine concepts [26]. On the other hand, we have another lightweight scripting language called Tcl, which was introduced five years before Lua. While not as lightweight as Lua, Tcl comes with a rich standard library. Tcl is known for its command-based syntax—where everything is treated as a string—and its popular GUI library (Tcl/Tk), which is used in other programming languages like Perl, Python, and Ruby for GUI tasks [27–31].

Microsoft Visual FoxPro (VFP) is a fast-commercial dBase dialect that natively supports object-oriented programming. It includes a powerful IDE with auto-complete features and a GUI builder (Form Designer) like Visual Basic. However, the latest release of the language (Visual FoxPro 9.0 SP2) is a 32-bit Windows product and is no longer actively developed [34].

Python is immensely popular. Although not as lightweight as Lua or Tcl, Python boasts a rich standard library and supports various programming paradigms. It has found widespread use in scientific computing and machine learning. Python comes with an integrated development environment (IDE) called IDLE, although it lacks a Form Designer/Builder, which exists in Visual FoxPro. While such tools exist for Python through external libraries and tools, having such features in the standard tools of the language could increase its usage in GUI development, especially since most standard IDEs for desktop platforms come with these features [35–37].

The Ruby programming language is an example of a dynamic language that is used to create DSLs. This reduces the development cost and the learning curve required to create a DSL. Unfortunately, these DSLs will not look like Supernova or SQL because they are influenced by the Ruby syntax [43,44].

With respect to syntax translation, numerous projects have attempted to address this gap. For example, several Arabic programming languages (Like Supernova) have been developed with a focus on using Arabic syntax. However, most of these projects remain unused in production due to limited features and are no longer actively developed [16–21]. Another approach involves creating packages that introduce translation as a feature. For instance, zhpy is a Python package that enables the writing of source code using traditional Chinese keywords, which is then converted to Python. However, this approach can suffer from multiple issues, including additional development and testing efforts, as well as lower compile-time performance due to the extra layer of translation before invoking the Python interpreter. Some programming language designers have recognized this challenge and intentionally introduced syntax localization. For instance, the Supernova programming language supports both Arabic and English syntax simultaneously, allowing an easy way to share libraries written in different languages. However, adding translation support for additional human languages without modifying the language implementation remains a complex task. In contrast, the Citrine programming language provides multiple versions that support over 100 human languages. Unfortunately, it does not offer an easy mechanism for sharing code across these language versions within the same project, as sharing code requires translation [18–20].

The implementation of dynamic programming language virtual machines could use a Global Interpreter Lock (GIL) to ensure that only one thread can access the interpreter at a time. This provides safety and avoids race conditions but prevents better performance from using threads on multi-core systems for CPU-bound tasks [38–40]. For a programming language like Python, there is ongoing work towards removing the GIL in recent versions.

The proposed dynamic programming language is designed to incorporate and improve features related to the first five categories: Lightweight, Embeddable, Scripting,

and Batteries-Included, providing powerful support for embedded DSLs, and it comes with an IDE suitable for GUI development. Additionally, it provides syntax flexibility and supports non-English syntax. Importantly, it does not belong to other categories, such as domain-specific languages, concurrency-oriented languages, or those focused solely on performance. While the proposed language is not a domain-specific language itself, it could be used to create domain-specific languages. Additionally, while it is not specifically designed around concurrency or performance, its proposed implementation—using a VM without a GIL—allows the use of threads to improve the performance of CPU-bound applications. The proposed programming language is designed to have a small implementation and provide direct support for multiple programming paradigms in the first place, then be fast enough and provide better runtime performance.

3. Materials and Methods

In this Section, we delve into our system design and implementation. We highlight the essential features of the proposed dynamic programming language, Ring, and present the system architecture. Our focus lies on the language features that facilitate localization, syntax customization, and the development of domain-specific languages. The language has been meticulously designed to offer syntax flexibility and empower users to customize the language syntax according to their specific needs. This leads to the ability to create internal domain-specific languages (IDSLs) that look like external domain-specific languages without the need to create specific parsers for them where the language constructs will be enough to achieve this goal.

3.1. System Architecture

In Figure 4, we present the system architecture, which comprises three layers: the language layer, the batteries-included layer, and the tools layer. The language layer is closely tied to the compiler and the virtual machine implementation. It defines the core programming language features, syntax, and semantics. In the Batteries-Included Layer, we encounter various extensions and libraries that cater to different domains. These include support for GUI, databases, web development, game development, and even platforms like Raspberry Pi Pico. The tools layer encompasses both command-based utilities (such as the package manager and REPL) and graphical tools like the form designer.

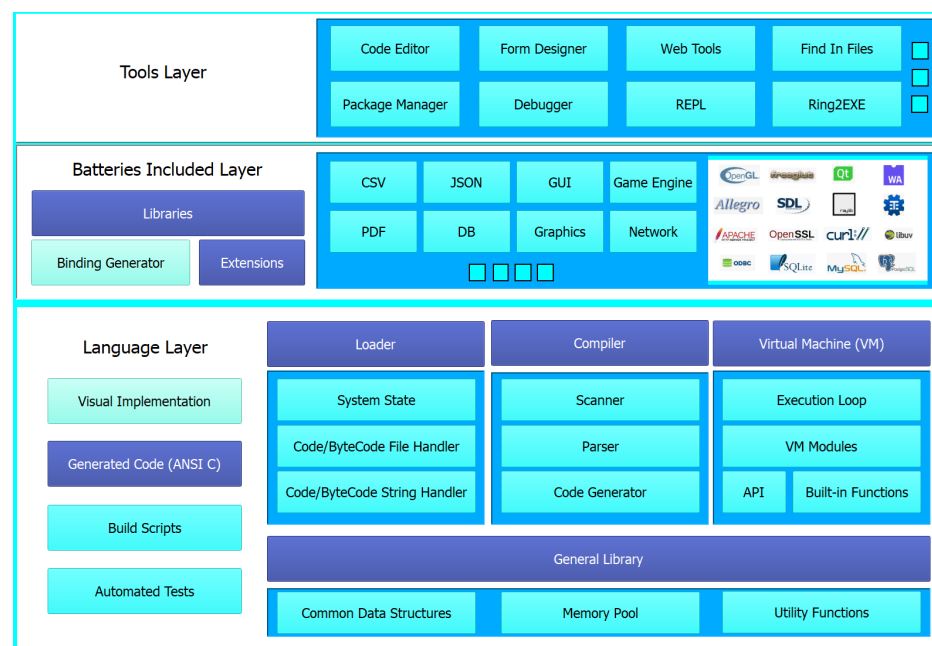


Figure 4. The proposed system architecture.

In the language layer, we have visual implementation, generated code, build scripts, and automated tests. Visual Implementation is developed using Programming Without Coding Technology (PWCT) software (version 1.9) [13,45]. The generated code is written in the C programming language (specifically ANSI C) and necessitates a C compiler to build the Ring executable. Throughout development, we employed multiple compilers, as illustrated in Table 3. With respect to the build scripts and the automated tests, we have employed batch files and shell scripts to automate the build process. Additionally, we have a CMake file that can generate the C project for multiple compilers [62]. This file uses CMake version 3.5. After each update to the project's source code and before committing code using Git [63], we used to run a comprehensive suite of tests. This process is now automated through a Ring program that executes each test in a separate process and verifies the output against the expected results. We are using Git version 2.42.

Table 3. C Compilers used for building our Ring Compiler/VM.

C Compiler	Platform/OS (Target)
Watcom C/C++	MS-DOS
Microsoft Visual C/C++	Microsoft Windows
GNU C/C++	Ubuntu Linux
Clang	macOS
Android-clang	Android
Emscripten	WebAssembly
GNU ARM embedded toolchain	Raspberry Pi Pico

The General library provides common features used by other components, such as the loader, compiler, and VM. These features include functions for processing files and directories, especially when Ring is used on an operating system that provides a file system. Additionally, the library implements Strings, Lists, and Hash Tables. One of the crucial features offered by the library is the Memory Pool, which pre-allocates memory. The size of the pre-allocated memory depends on the environment: a few kilobytes are allocated when using Ring on microcontrollers like the Raspberry Pi Pico, while several megabytes of memory are pre-allocated when using Ring on desktop environments such as Windows, Linux, and macOS.

Since the proposed programming language can serve as both a scripting language and an embeddable language, it needs features that fulfill these dual roles. This is achieved through the loader, which plays a managerial role in our system design. The loader determines what actions will be taken and what will be avoided. It can print usage information, process source code or bytecode files, execute code from strings, halt operations at specific points (such as obtaining scanner tokens), or display applied grammar rules, among other tasks. To achieve these objectives, the loader calls the Compiler/VM components.

Since the implementation never uses C global variables, the loader also creates the system state, and the state pointer is passed to different functions that require access to common information about the processed files, the current stage, the memory pool, and so on.

In the Ring compiler, we have three main modules: the scanner, the parser, and the code generator. The scanner reads the textual source code and converts it into tokens (such as keywords, operators, identifiers, and constants). The parser processes these tokens, checking for correct adherence to the language grammar, and then invokes the code generator functions to produce bytecode. Ring employs a single-pass compiler [64], where parsing, code generation, and optimization are interleaved. However, the language performs only a few optimizations during code generation.

All these decisions are made in favor of maintaining a small implementation. The language implementation utilizes a stack-based virtual machine [3]. This virtual machine is specifically designed for the language and contains many instructions that directly map to

its features. In total, there are 128 instructions within the virtual machine. The VM comes with 255 built-in functions and provides an API for extensions written in the C language.

In the Batteries Included Layer, we have a powerful tool called the Binding Generator (Like SWIG for Python [65]). Our tool is written in Ring itself and allows us to use straightforward configuration files that describe and customize the functions and classes available in C/C++ libraries. Once these configuration files are in place, the generator works by producing the extension code. This code enables us to seamlessly use those C/C++ functions and classes from within the Ring language programs. To build extensions, we can employ a C/C++ compiler, resulting in dynamic link libraries (DLLs), shared objects (SOs), or dynamic libraries (Dylibs) according to the platform (Windows/Linux/macOS) [66].

In Table 4, we find a list of external C/C++ libraries used by the standard Ring extensions provided by the language. The selection of these libraries is based on our experience of using them in previous projects. Most of these libraries enjoy popularity within the C/C++ community and cover various programming domains, including Database, Graphics, Multimedia, Games, Terminal, GUI, Network Programming, and Web Development.

Table 4. A list of C/C++ Libraries used by Ring Extensions in the Batteries Included layer.

Domain	C/C++ Libraries/Tools	Count
Terminal User Interface (TUI)	ConsoleColors and RogueUtil	2
Network and Security	LibCurl, Libuv, and OpenSSL	3
Web Servers	HTTPLib and Apache Web Server	2
Database	ODBC, SQLite, MySQL, and PostgreSQL	4
Games & multi-media	Allegro, LibSDL, RayLib and Tilengine	4
Graphics	OpenGL, FreeGLUT and StbImage	3
Graphical User Interface (GUI)	Qt, Libui, and NAppGUI	3
Common Files	MiniZip, PDFGen and CJSON	3
SDK for Specific Platforms	Android SDK and Raspberry Pi Pico SDK	2

These libraries exhibit different characteristics that impact the produced software. For instance, the Qt GUI Framework offers an extensive array of classes and features, but delving into it necessitates investing more time to study the framework [67]. Consequently, programs built with Qt may have larger runtimes. On the other hand, a lightweight GUI library like Libui has fewer features compared to Qt, but it excels in being compact. Notably, most of the GUI tools in the Tools layer are based on RingQt.

In the Tools layer, we have a group of command-based tools such as the Ring Package Manager, Ring2EXE, and the REPL. Additionally, we have GUI-based tools like the Ring Notepad, which serves as our code editor, and the Form Designer, which is used for designing application user interfaces and generating code following the MVC design pattern [68]. Furthermore, Ring includes an application for searching text in multiple files—a common feature required for large projects. All these tools are written using the Ring programming language itself. Software documentation helps users learn and use it. The language is distributed with documentation of over 2000 pages in the English language that cover the different features and concepts. Also, there are chapters that cover the different extensions, libraries, and tools provided by the language. The documentation is created using Sphinx, a Python-based documentation tool. Specifically, we are using Sphinx version 6.2.1, HTML Help Workshop version 4.74.8702, and MiKTeX 23.4.

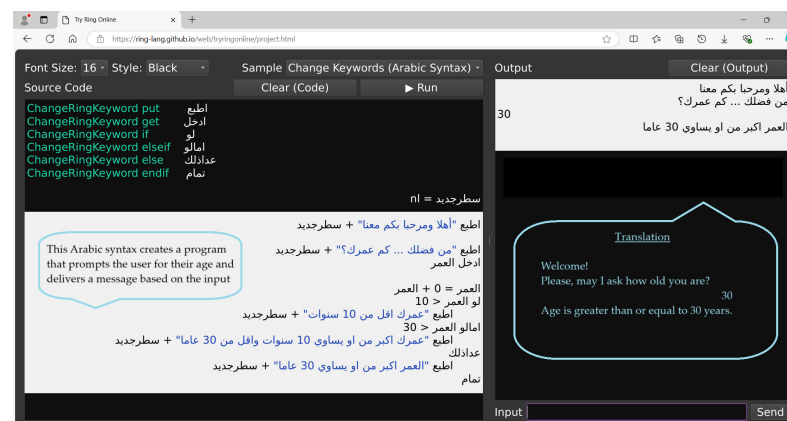
3.2. Non-English Syntax

The language scanner (The first phase in the compiler) supports specific commands (illustrated in Table 5) that allow users to change the language keywords and operators multiple times, facilitating easy translation of the language syntax.

Table 5. Scanner commands provided by the Ring Compiler.

Command	Parameters	Usage
ChangeRingKeyword	OldKeyword NewKeyword	Change language keyword
ChangeRingOperator	OldOperator NewOperator	Change language operator
LoadSyntax	Syntax file name as literal	Load syntax file
EnableHashComments	None	Support using # for comments
DisableHashComments	None	Disable using # for comments

In Figure 5, we present a WebAssembly application developed using Ring for on-line language experimentation. Additionally, we provide an example of how Scanner commands can be used to switch language keywords to Arabic syntax. The code begins by translating keywords (such as put, get, if, elseif, and endif) from English to Arabic. Subsequently, it employs this Arabic syntax to create a program that prompts the user for their age and delivers a message based on that input.

**Figure 5.** Arabic syntax within a WebAssembly application developed using Ring.

Rather than including these Scanner commands at the start of every Arabic source code file, we can use the LoadSyntax command. This command allows us to load syntax files containing groups of these commands. Additionally, instead of placing the LoadSyntax command at the beginning of each source code file, we can simply add a file named (ringsyntax.ring) to the Arabic project folder. The Scanner will automatically load this file whenever we use any Ring source code file in the same folder.

This approach draws inspiration from the use of (__init__.py) files in Python modules and the concept of (.htaccess) files in the Apache HTTP Server [69,70].

The application is developed using Ring through the following steps:

1. The user interface is designed using the Ring form designer. The form file (try.rform) generates the tryView.ring file, which contains the RingQt source code that defines the window controls and layouts and sets the default style;
2. In the controller class (tryController.ring), we determine the Ring code that will be executed based on user interaction with the application GUI;
3. The (style.ring) file contains the Style class, which changes colors based on the selected style. The default style is Black, and the user can change it to another predefined style (Black, White, Blue, Modern, or Windows);
4. The (samples.ring) file contains a Ring list that provides the predefined samples, where each sample is represented through a nested list containing the sample name and the sample code;
5. The (ringvm.ring) file contains a class that enables the sample's source code to be run in an isolated Ring virtual machine. If the sample produces a runtime error or terminates the program, we can still use the Try Ring Online application executed by the caller VM;

6. The (onlinering.ring) file contains functions that are automatically called before executing any sample code in the nested VM. These functions override the standard functions used for input/output operations. When the console application requires an input, these functions will pause the sub VM. Later, when the user enters the required data, the caller VM will update the state of the sub VM, set the variable value, and then resume the sub VM.

3.3. Domain-Specific Languages

One of the features provided by the Ring language is the ability to create domain-specific languages (DSLs) on top of classes. These DSLs can employ specific syntax, and we have the freedom to design this syntax. The underlying idea relies on using braces to access objects, granting us the ability to utilize the attributes and methods provided by those objects. Unlike some other programming languages that offer the “with” statement, in Ring, this feature is provided through an operator. This operator allows us to use this feature within expressions and in various places throughout the code. Notably, Ring does not require semicolons or new lines between statements. We can type different statements on the same line without any fuss. Additionally, in Ring, every expression is an acceptable statement, giving us the freedom to write various values, all of which will be accepted by the compiler.

Ring classes also support properties. Typing a property name can invoke the getter method and execute the associated code. Moreover, Ring goes a step further by allowing us to define methods like `braceStart()` and `braceEnd()`. These methods are automatically called when we access an object using braces. Furthermore, the language automatically invokes a method called `braceExprEval()` when we write an expression inside braces. With these features, coupled with the ability to customize language keywords and operators, we can construct domain-specific languages that resemble external DSLs such as CSS, QML, SQL, and Supernova.

As an example, we will implement a tiny DSL. This simple DSL accepts a group of numbers. While entering numbers, we can highlight some of these numbers as important. Additionally, we could stop the processing using the “stop” command. The results of the process include both the summation of the entered numbers and a list of the important numbers. When typing numbers, using new lines is optional. Also, we do not need to use `()` or `[]` to group these numbers.

Table 6 presents an example of how to use this tiny DSL and the expected output. In the example, we entered a group of numbers while asking for some of them to be highlighted using the (Important) word. Then, we decided to stop processing after the number 60 using the (Stop) word.

Table 6. Example about a simple domain-specific-language implemented using Ring.

Usage (Ring Code)	Output
new DSL {	
200	
400 Important	
50	
600 Important	Sum: 1520
60	Important:
10 20 30	400
40 50 60 Stop	600
70 80 90	
800 Important	
}	

To implement this tiny DSL, we only need to write one Ring class containing seven lines of compact code. Figure 6 demonstrates the implementation and analysis of the Ring features used in this class code.

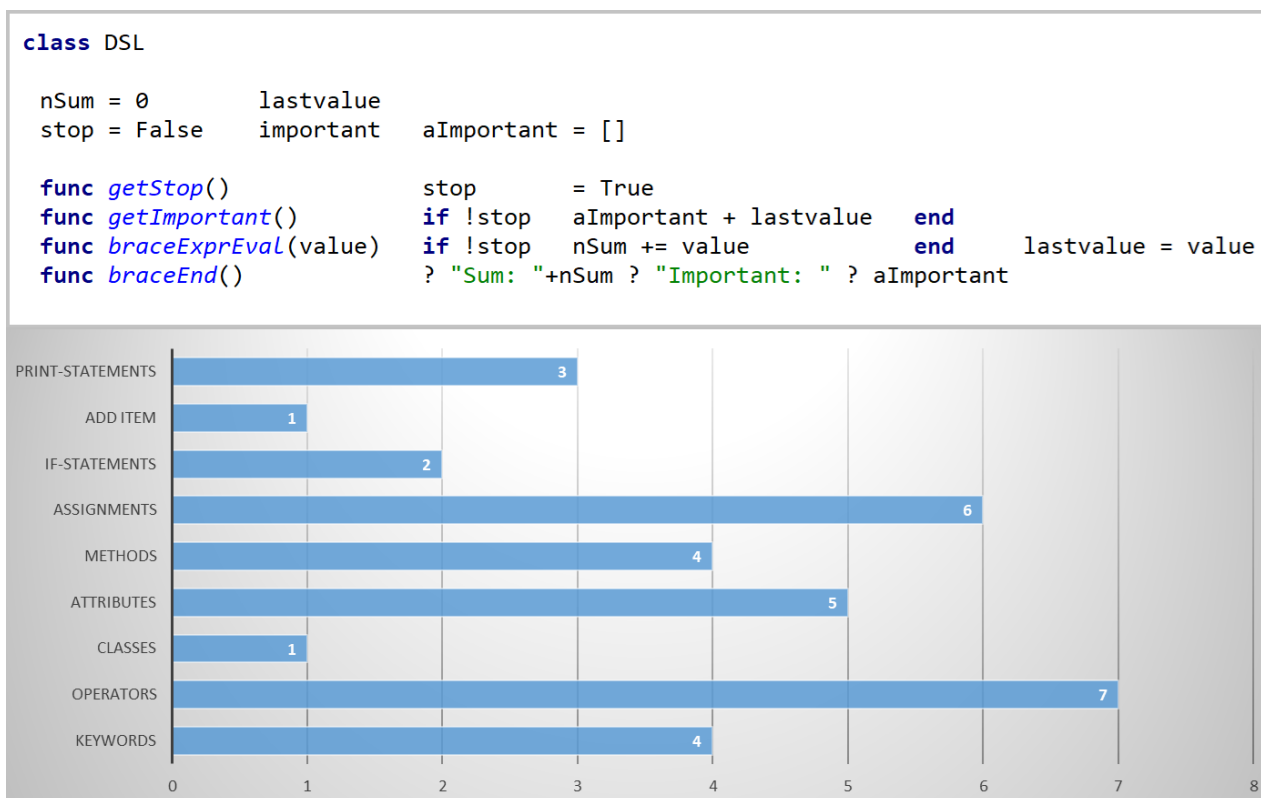


Figure 6. Ring code to implement a simple domain-specific language.

We begin by defining the class and setting its name (DSL). After that, we declare five attributes within our class: `nSum`, `aImportant`, `important`, `stop`, and `lastvalue`. Since our tiny DSL has two commands (Important and Stop), after declaring them as attributes, we define the methods `getImportant()` and `getStop()` to determine what happens when these words are used. The `braceExprEval()` method is called each time the language processes a number inside the braces that access the object. We sum up these numbers by adding them to the `nSum` attribute, and we store the last encountered number in the `lastvalue` attribute. Finally, when we finish using braces, the `braceEnd()` method is automatically called, printing the results.

3.4. Object-Oriented Programming (OOP)

The Ring programming language provides direct support for many features related to the object-oriented programming paradigm [71], such as classes, objects, encapsulation, composition, aggregation, inheritance, polymorphism, and operator overloading. Additionally, we have extended these features by introducing new capabilities, such as using braces to access objects (`braceStart()`, `braceEnd()`, `braceExprEval()`, etc.) to facilitate the implementation of domain-specific languages (DSLs) using classes.

This allows us to seamlessly blend our DSL implementation with the well-known features of object-oriented programming. For instance, consider Figure 7, where we introduce an update to our domain-specific language. This updated DSL supports retrieving groups of computer prices and highlighting acceptable prices while displaying the output through a Graphical User Interface (GUI). The figure contains four sections. The first section represents the data used as input for the object created from the `PickPrice` class. The second section presents the `PickPrice` class. The third section is related to the DSL class. The fourth section, on the right side of the figure, shows the program's output: a simple GUI window containing a list box.

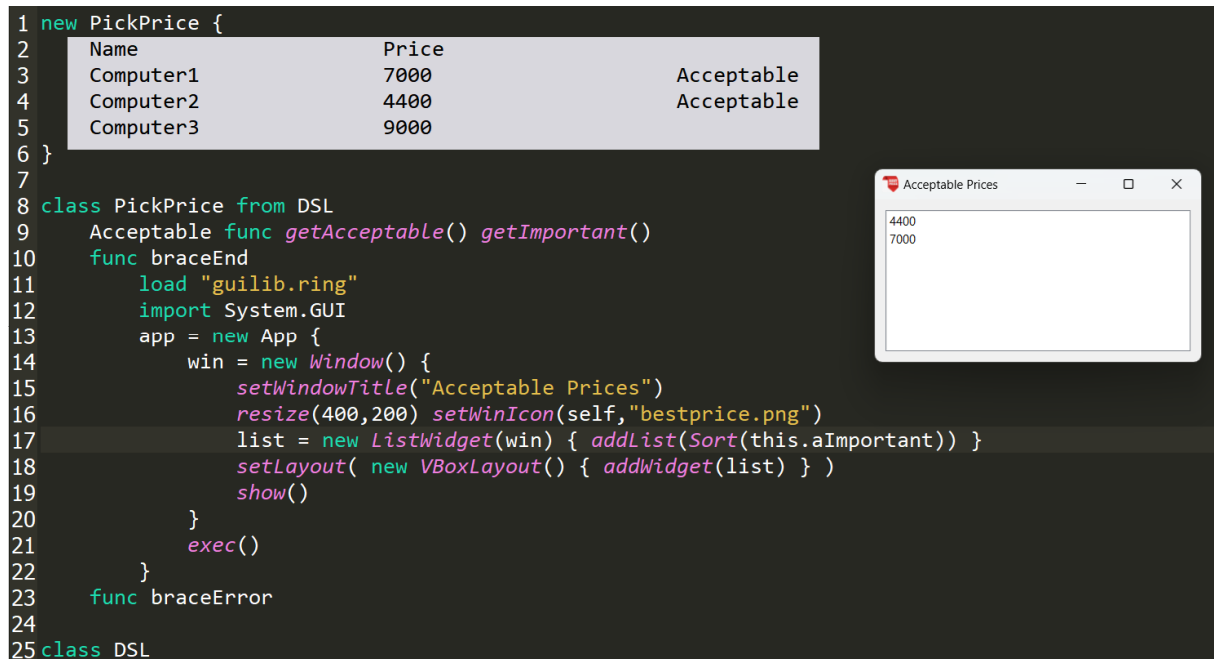


Figure 7. Extending our DSL using inheritance and the GUI library.

We define a new class called `PickPrice`, which inherits from the existing `DSL` class (still present in our code, starting from line 25). Inside the `PickPrice` class, we introduce a new attribute called `Acceptable` (used in place of the word "Important"). We define the `getAcceptable()` method, which is called when the Ring language encounters the word "Acceptable" within code, which uses braces to access an object created from the `PickPrice` class. Essentially, it acts as a wrapper method, invoking the existing `getImportant()` method from the parent class (`DSL`) to reuse its functionality.

At line 10, we override the `braceEnd()` method implementation. Instead of the command-based user interface, we replace it with a GUI. The GUI code leverages the `Guilib` library provided by the Ring programming language, utilizing the popular `Qt` framework. In line 17, we sort the numbers in our list (`aImportant`) using the `Sort()` function and then add the items to a `ListWidget`. Line 23 introduces the `braceError()` method, designed to prevent errors when typing words like "Name", "Price", "Computer1", etc.

While we are inside class methods, using braces to access objects changes the current object. Consequently, we need different variables to reference the opened object or the object instance created from the current class. That is why we have (`self`) and (`this`). In line 16, while we are inside the window object, we pass `this` object to the `setWinIcon()` function using the (`self`) variable. In line 17, to access the `aImportant` attribute (inherited from the `DSL` class), we use the (`this`) variable to refer to the object instance that will be created from the `PickPrice` class.

3.5. Batteries Included

The language ships with numerous extensions that wrap up popular C/C++ libraries such as `Allegro`, `LibSDL`, `OpenGL`, `Qt`, and `SQLite`. These extensions provide convenient bridges between Ring and these well-established libraries, making it easier for developers to harness their capabilities. Ring libraries add another layer of abstraction on top of Ring extensions. One standout example is the Game Engine for two-dimensional (2D) Games. This engine consists of a set of classes built around the `Allegro` and `LibSDL` game programming libraries [72,73]. What is interesting is that the engine encourages a declarative coding style reminiscent of CSS or QML. So, when working with it, we can express the game logic in a way that feels quite intuitive and expressive.

The language comes with an extension called RingPico, which supports Raspberry Pi Pico SDK [74]. In Figure 8, we see an example of using this extension. We can employ procedural programming and directly call the functions provided by the extension, much like we would when coding in a language such as C. Additionally, we have the option to build classes around these functions or leverage the features of the Ring language for more declarative code as demonstrated in the Figure.

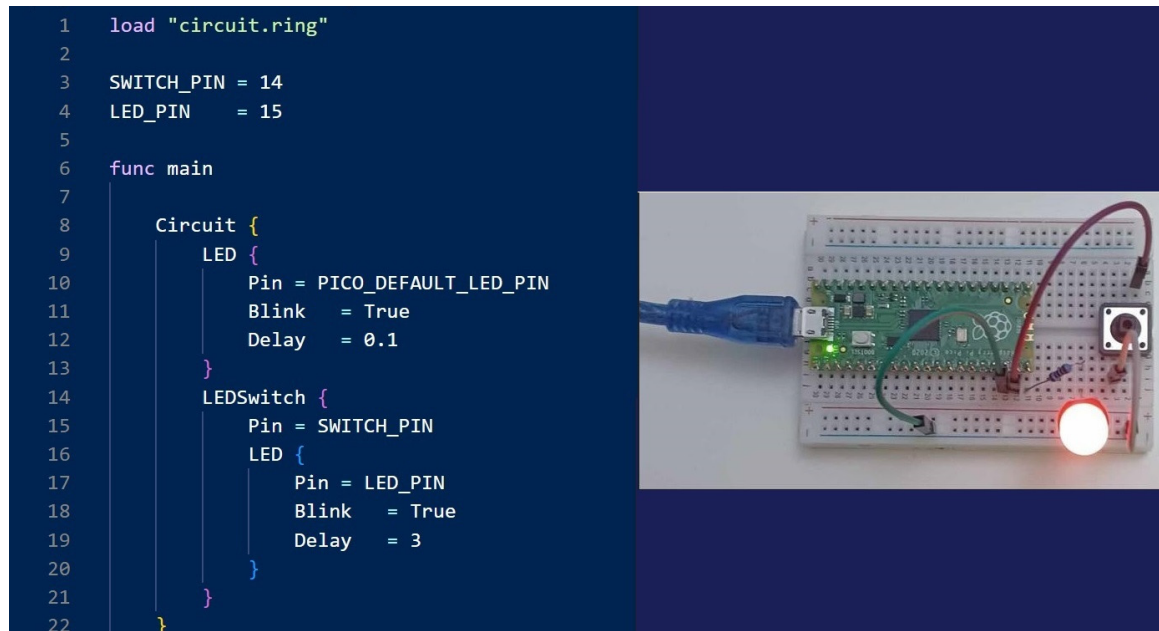


Figure 8. Using Declarative Style in Ring for Raspberry Pi Pico programming.

In this example (see Figure 8), we load the file `circuit.ring`, which contains the classes `Circuit`, `LED`, and `LEDSwitch`. Inside this file, an object called `Circuit` is created from the `Circuit` class as a global variable. We can access this object directly using braces, as shown in Line 8. Once we are inside the object, using words like `LED` or `LEDSwitch` is equivalent to accessing attributes defined in the `Circuit` class.

Using these attributes will call the `getLED()` and `getLEDSwitch()` methods. Invoking these methods creates new objects from the `LED` and `LEDSwitch` classes, which are then added to a list of objects defined inside the `Circuit` class. Additionally, the `getLED()` and `getLEDSwitch()` methods return these newly created objects. We can access these objects using braces, just as we did in lines 9, 14, and 16. Once we have access to an object, we can set its attributes.

The additional magic related to the execution loop and responding to attribute values (such as `Pin`, `Blink`, and `Delay`) is handled through the `braceEnd()` method. This method initiates the execution loop and calls other methods that check the defined objects and their attributes.

3.6. The IDE and the Form Designer

We developed GUI-based tools (Demonstrated in Figure 9) like the Ring Notepad, which serves as our code editor, and the Form Designer, which is used for designing application user interfaces and generating code following the MVC design pattern [70]. Furthermore, Ring includes an application for searching text in multiple files—a common feature required for large projects. All these tools are written using the Ring programming language itself, totaling around 15,000 lines of Ring code. Developing these tools based on the Qt framework requires knowledge of GUI development, object-oriented programming, and how to organize large programs in the Ring language.

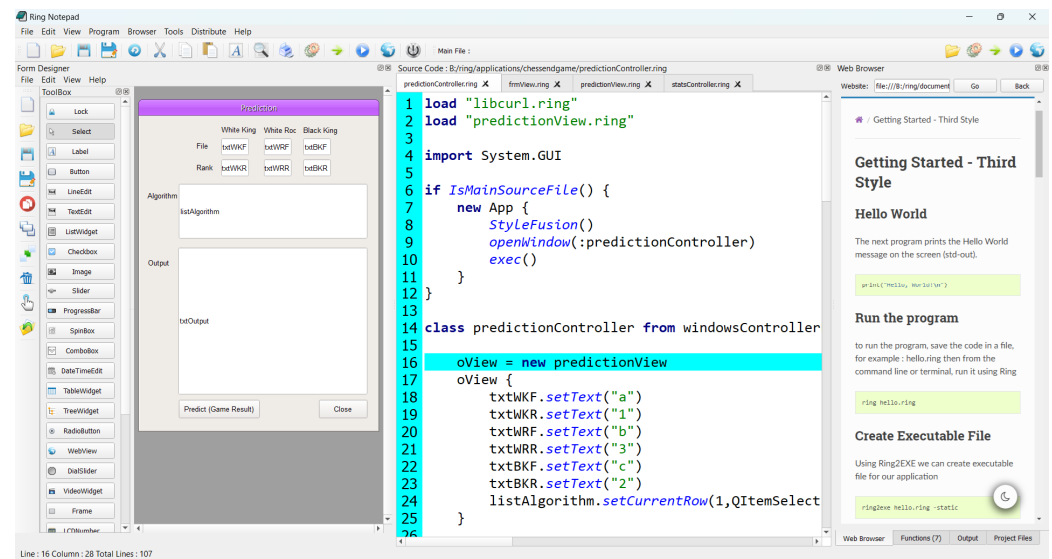


Figure 9. Ring IDE (Code Editor, Form Designer, etc.) is developed using Ring itself.

3.7. The Implementation (Using Visual Programming)

Ring Visual Implementation is developed using Programming Without Coding Technology (PWCT) software (version 1.9) [13,45]. This software offers various visual programming languages, including HarbourPWCT, PythonPWCT, SupernovaPWCT, C#PWCT, and CPWCT [75,76].

Each visual language corresponds to a specific textual programming language used in the code generation process. In our case, we utilized CPWCT to design different components, such as the General Library, Loader, Compiler, and Virtual Machine, through visual programming. Subsequently, we obtained the source code in the C programming language.

PWCT is designed to provide precise control, like what we experience with textual code editors, while also offering visual programming advantages such as reducing errors and the ability to work with multiple dimensions and a rich user interface. Notably, PWCT includes a powerful feature called the Time Dimension during visual programming. With this feature, each step or block generated in the program stores information about development time. Programmers can watch the program evolve step by step, revise the order of the construction process, and even run the program at specific points in time.

The General library plays a crucial role in successfully implementing the Ring language as a lightweight programming language. Other components extensively reuse the library functions to implement many language features using lists and hash tables instead of specific C structures. The list implementation (demonstrated in Figure 10) uses a Doubly Linked List, Deque (Double-Ended Queue), and Singleton Cache to cache the pointer of the last visited item and the index of the next item. This allows for quick traversal of the list through a function that receives the item index as a parameter.

Additionally, there is an optional array of pointers that can be used in specific situations to quickly find an item (through the item index) and use it without the need for the traverse process or using the singleton cache. Furthermore, if the size of the list items is known when creating new lists, memory could be allocated as a continuous block to be cache-friendly and minimize cache misses. Using this data structure to implement language features enables us to create a lightweight language with numerous capabilities while also contributing to stability and reducing memory management errors. It is like writing High-Level code like dynamic language code. However, it is essential to acknowledge one clear disadvantage of this approach: lower performance and increased memory usage compared to using specific C structures or arrays when implementing features. To strike a balance, during language development, we identified performance bottlenecks—such as function call implementations—and replaced them with specific low-level implementations

based on a pre-allocated array of structures. Such optimizations became necessary once we started supporting the Raspberry Pi Pico microcontroller.

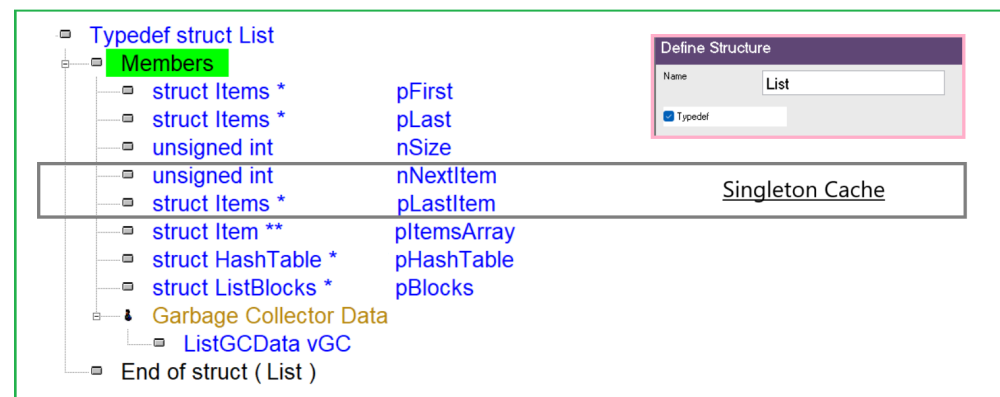


Figure 10. Using PWCT to define the List structure which uses a singleton cache.

While the Singleton Cache consumes less memory compared to the array of pointers, it introduces a challenge: every read operation from the list items could potentially trigger a write operation if the cache is updated. Such behavior is undesirable, especially when sharing lists between threads. In this scenario, we opt for the array of pointers to avoid relying on the singleton cache.

In Figure 11, we observe how the language grammar rules are implemented using PWCT. For each group of grammar rules, we define a specific function. We had a step that described each rule, allowing us to focus easily on specific rules using features like collapsing and expanding in the steps tree. Each group of steps associated with the same component can have an interaction page (a data entry form) that receives component parameters and controls the steps' generation and update processes. Notably, all components, including the "Call Function" component, are created within the PWCT environment itself. We have the flexibility to create new visual components or update existing ones. During the development of the Ring Compiler/VM, we exclusively used the standard components provided by PWCT. No new components were necessary because the available ones sufficed for implementing the required features. In the toolbar, there is a combobox for selecting the visual programming language. PWCT initially started with "HarbourPWCT" as the default visual language, but we specifically chose "CPWCT" to develop our project based on the C language.

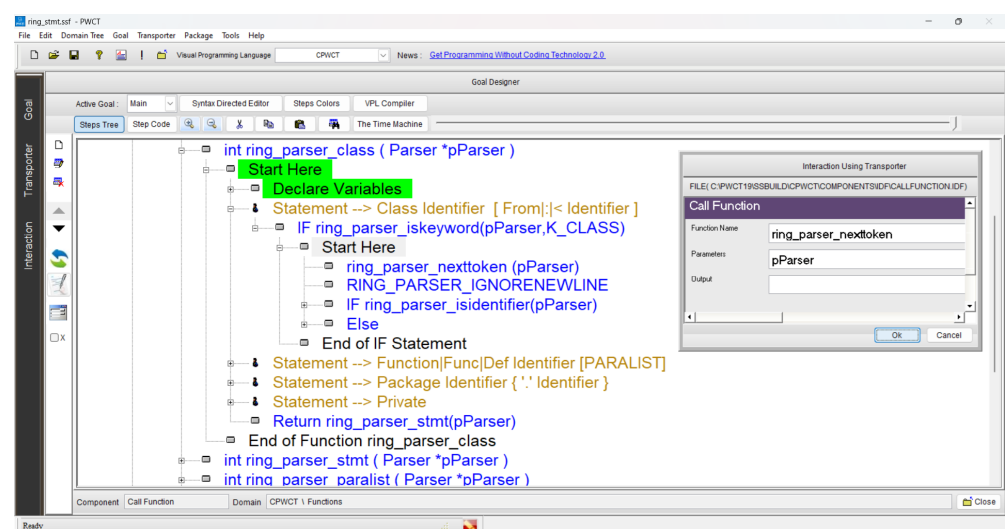


Figure 11. Implementing the Ring language grammar using PWCT.

In the same Figure, you will notice a button labeled “The Time Machine.” Clicking this button provides a menu of options that allow us to play the program like a movie, revealing the construction steps. Another crucial button is the “VPL Compiler.” By using this button, we can examine the composition of different visual components, as demonstrated in Figure 12. An interesting feature in the results is the count of interactions (visual components) and the number of steps within the steps tree. These metrics provide insight into the abstraction level offered by the interaction pages of the visual components.

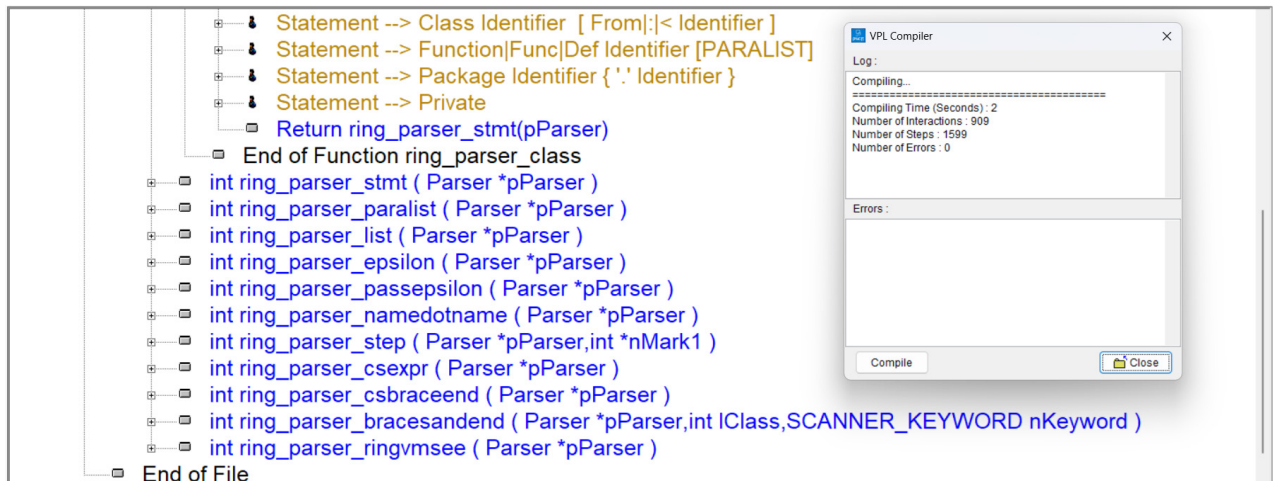


Figure 12. Using the VPL Compiler to get statistics about the visual representation.

An interesting question arises: why do we need a Visual Programming Language (VPL) Compiler if the visual language itself is designed to prevent errors? The answer lies in our ability to disable the Syntax Directed Editor, allowing us to manually arrange the generated steps for visual components to do quick organization and refactoring. However, this flexibility can sometimes lead to mistakes. Imagine a scenario where a component is inadvertently placed in the wrong location, and the programmer does not immediately notice the error. In such cases, the VPL compiler becomes invaluable—it can catch these composition errors and help ensure the program is correct.

The Ring compiler generates bytecode, where each instruction must contain an operation code and can include zero, one, or two arguments. This bytecode is stored by the Ring compiler as a Ring List, allowing the compiler to easily insert instructions during code generation. However, when this bytecode is passed to Ring VM, it undergoes a conversion process to a more suitable representation for execution. This representation is stored as a single continuous block in memory (rather than multiple byte-code chunks). Notably, it includes extra space that can be utilized for new instructions produced at runtime by the Eval() function. By having this additional space, the need for frequent memory reallocation is reduced.

Within the Virtual Machine (VM), the bytecode representation employs fixed-size instructions. Specifically, the size of each instruction is 16 bytes for 32-bit microcontrollers (such as the Raspberry Pi Pico) and 24 bytes for 64-bit desktop environments. This bytecode is writable, allowing the VM to update instructions during runtime—caching certain values and even replacing instructions with faster alternatives that utilize pointers for variables, thus avoiding costly search processes.

In Figure 13, we observe the structure of the bytecode on the left side. On the right side, we find the VM instructions that have been added to enhance performance. Opting for a writable long-byte code format is somewhat unconventional; for instance, Python uses 2 bytes per instruction, while Lua uses 4 bytes [77,78]. However, our deliberate choice of a long-byte code format serves two key purposes:

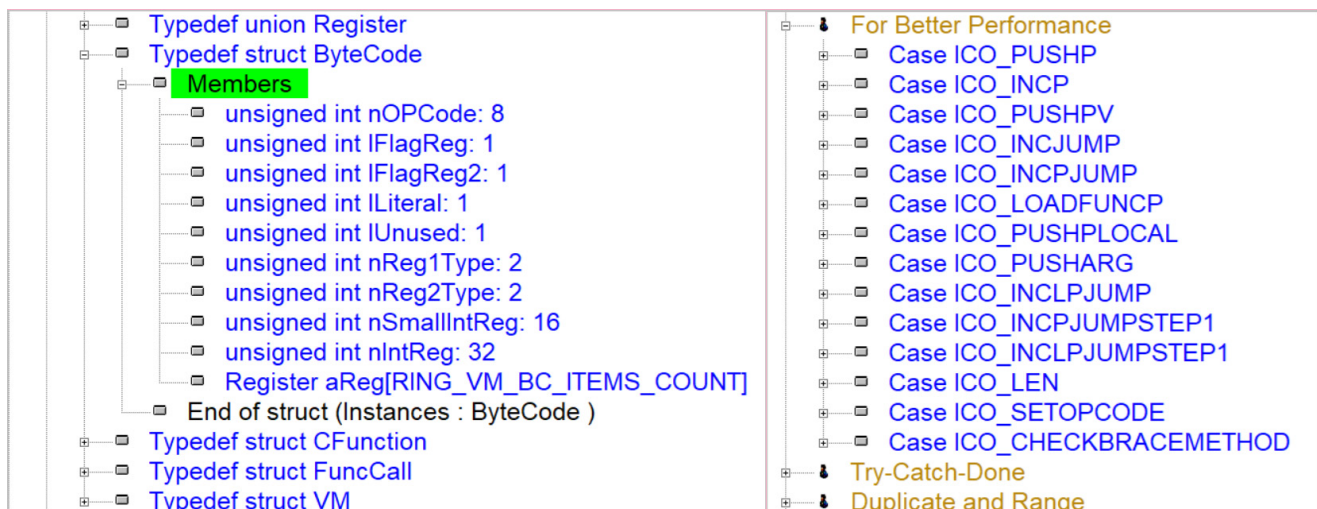


Figure 13. Ring Virtual Machine implementation using PWCT.

Simplicity of Implementation: Despite supporting a language with a substantial number of features (128 instructions), we aimed for a compact implementation. The writable long-byte code format allows us to achieve this without unnecessary complexity.

Performance Optimization: By using a longer bytecode format, we gain flexibility. We can improve the performance of specific instructions without resorting to a just-in-time compilation of machine code or significantly increasing the overall implementation size.

When considering the disadvantages of using a writable long-byte code format, it is essential to address a few key points. First, this approach results in larger memory requirements, which increases the likelihood of cache misses—a factor that directly impacts performance. Additionally, storing the byte code in writable memory can be costlier, especially on microcontrollers like the Raspberry Pi Pico [79,80]. However, to mitigate the drawbacks associated with the larger bytecode size, the Virtual Machine (VM) incorporates a clever strategy: some instructions serve multiple purposes. This technique is well-known and proves useful in common scenarios. By identifying instructions that are frequently used together, we can optimize their representation. In doing so, we avoid allocating unnecessary space for instructions that do not require the full extent of the provided memory [81].

The Virtual Machine (VM) implements several useful features when embedded within projects. For instance, during program execution, we can suspend or resume the VM [82], allowing a running program to request VM suspension while preserving its state. Additionally, the VM supports having multiple language states—meaning that more than one instance of the Ring VM can coexist within the same application. These features are of practical use in the “Try Ring Online” application. Within this application, when we write and run a program, it creates a new language state specific to our program. If the program requires input from the console, it halts the sub-virtual machine, signaling to the main VM that the console application is awaiting input. Users then type their input in a GUI provided by the main VM and click “Send”. These data are copied to a variable associated with Sub VM, and a resume operation follows. As a result, the console application in Sub VM can receive the input. This approach enables us to create a playground for the Ring language as a WebAssembly application without the need for threads.

In summary, all the modules related to the General Library, Loader, Compiler, and Virtual Machine are designed using visual programming through PWCT. We have 43 visual source files that generate 44 C source files and 28 C header files. Each visual source file could generate one or more textual files. To achieve a lightweight implementation while retaining a programming language with rich features and ensuring performance comparable to scripting languages, we made the following design decisions [77–82]:

1. Applying principles such as “Don’t Repeat Yourself” (DRY) and the “Keep it simple and stupid” (KISS) principle;
2. Using a single-pass compiler where the parsing and code generation are interleaved;
3. The ability to separate the compiler and the virtual machine and use any of them alone;
4. All the built-in functions are grouped in optional modules through preprocessor directives;
5. Ring Lists vs. C Structures: In most cases, we opted for Ring Lists over C structures;
6. Selective Use of C Structures: However, in specific features where performance impact matters significantly, we chose to use C structures. These targeted optimizations enhance critical parts of the language;
7. Flexible List Implementation: Our list implementation combines various data structures and optimization techniques, including Doubly Linked Lists, Deques (Double-Ended Queues), Singleton Caches, arrays of pointers, Hash Tables, and continuous memory blocks. This flexibility accommodates diverse use cases;
8. One block for Bytecode Storage: The bytecode resides in a single continuous memory block, avoiding fragmentation. Moreover, we intentionally allocate extra space within this block. This foresight reduces the need for frequent memory reallocation during runtime, especially when using the Eval() function;
9. Writable Long-Byte Code Format: The bytecode format uses a longer representation, which allows for performance improvements. During runtime, instructions can be dynamically replaced with faster alternatives, all without resorting to just-in-time compilation to machine code or bloating the implementation size;
10. The Virtual Machine does not use a global interpreter lock (GIL), which results in better performance when utilizing threads for CPU-bound tasks.

4. Results

In this section, we present the various results related to our study. First, we introduce information about the early users, followed by download statistics. Next, we discuss multiple use cases. Additionally, we delve into our findings concerning Ring’s visual implementation using the PWCT visual programming language. Then, we present the results related to Ring’s lightweight implementation, followed by the performance benchmarks.

4.1. Early Users and the Programming Language Used Prior to Ring (2016)

Once we launched the Ring website, we posted a message in the Ring Group seeking users interested in trying or testing the language and contributing by reporting bugs. In the public group, interested users shared their age, gender, country (location), and the programming languages they used prior to Ring. We noticed 43 messages, with 42 males and 1 female. Most of the users are between the ages of 20 and 35, and 81% reported that they were using C++, PHP, C#, Java, or Python, as demonstrated in Figure 14. We noticed that 28 users (65%) were using statically-typed languages, while 15 users (35%) were using dynamically-typed languages. This diverse usage background reflects the rich experience of our users with different programming languages, leading to various feature requests in different directions. Developers who used C# requested the development and addition of the Form Designer to our code editor (Ring Notepad), which was developed and added to Ring in version 1.3. Developers with a C/C++ background asked for features related to C/C++ extensions, leading to the revision and improvement of the Ring API and the addition of tutorials on using it. Additionally, developers who used PHP for web development requested better support for web development, which led to the addition of the Apache web server to Ring Notepad in Ring 1.6. Those users helped us discover and fix many issues. They also improved the Ring documentation by adding the Frequently Asked Questions (FAQ) chapter. Over time, they contributed over 800 samples of the Ring language to the RosettaCode website.

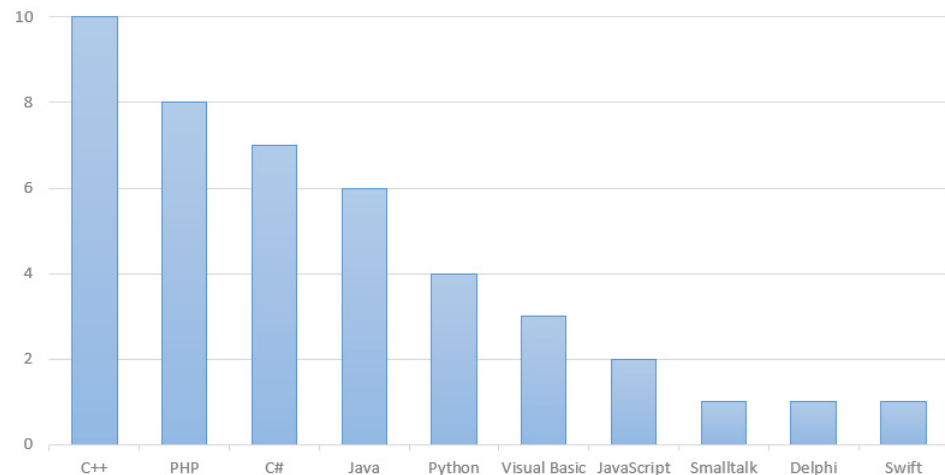


Figure 14. Early users and the language used prior to Ring.

4.2. Feedback from Online Course (2017–2018)

In 2017, we presented a free online course consisting of 18 videos in Arabic that introduced the Ring programming language (covering input/output, control structures, procedural programming, and object-oriented programming). We then invited interested learners to watch the course and submit the samples they wrote during their learning through GitHub so we could track their progress. We received samples from 76 participants.

In Table 7, we present the course content, while in Table 8, we introduce the statistics about the course. Twenty participants (26.3%) were not interested and finished fewer than two lessons, while 56 participants (73.7%) were interested and finished two or more lessons. Of those 56 participants, 23 (30% of the total) finished the course. We noticed that two participants became active contributors to Ring language samples and applications. The contributors help us test, report bugs, and add samples, applications, and tutorials. As of 2024, Ring is distributed with hundreds of samples and over 70 applications/games, each ranging from a few hundred to a few thousand lines of Ring code. With respect to the female participants, four of them completed the course, one completed just one lesson, and the last one completed three lessons.

Table 7. Course Content.

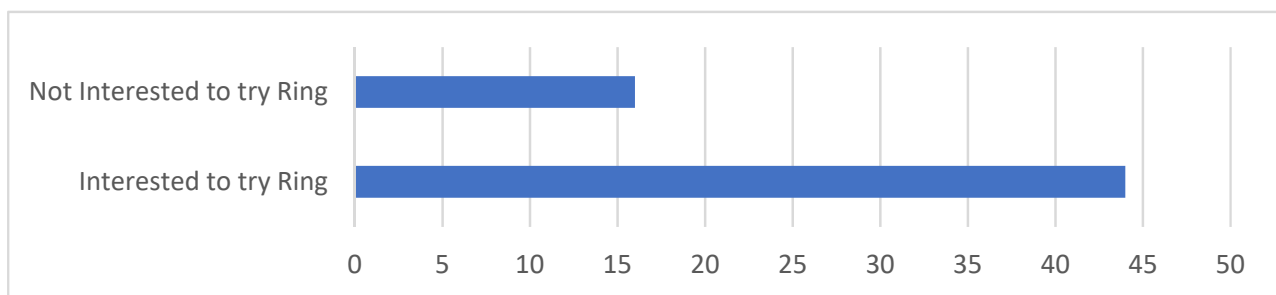
Lesson	Description	Duration (H:M:S)
1	Installing Ring and writing the Hello World program	0:30:43
2	Input/output, data types, strings, and numbers	0:36:33
3	Arithmetic/Logical operators and the if Statement	0:58:20
4	Lists, nested lists, and loops	1:01:71
5	While Loop	0:48:34
6	Defining and using Functions	0:45:30
7	Standard functions	0:28:39
8	Using Eval()	0:23:01
9	Internet Library	0:38:47
10	Database and SQL	0:33:13
11	Classes and Objects	0:47:07
12	Declarative Programming	0:52:29
13	Domain-Specific Languages	0:19:46
14	Domain-Specific Languages (Part 2)	0:46:33
15	Functional Programming	0:42:37
16	Reflection and Meta-Programming	0:27:57
17	Memory Management and variables scope	0:58:28
18	Interactive Debugger	0:22:08

Table 8. Statistics from Online Course.

Variable	Value
Male	70
Female	6
Completed less than two lessons	20
Completed more than one lesson	56
Completed the course	23
Contributors	2

4.3. Feedback After a One-Hour Lecture (2019)

In 2019, we presented a one-hour lecture about the Ring language to third-year students at the College of Computer and Information Sciences at King Saud University in Saudi Arabia. The lecture was presented twice: the first time to 35 students and the second time to 25 students. All 60 students were male. They had studied multiple courses related to programming, including Introduction to Programming and Object-Oriented Programming. They used Java during these programming courses. After the one-hour lecture, we told them, “If you are interested in the Ring language, try to download and install it, write some simple programs, and see if you become more interested in learning about the language”. As shown in Figure 15, out of the 60 students, 44 were interested, and all of them successfully installed the language and tried writing some programs using it. One of the students said, “Why don’t we learn Ring instead of Java? It seems easier”. Another student said, “This language looks like Python”.

**Figure 15.** Feedback from students about Ring language after a one-hour lecture.

4.4. Downloads Statistics and Users Group (2016–2024)

Ring, as an open-source programming language, is hosted on GitHub. Users have two options to get the language: they can clone the source code or download a precompiled binary release for Windows, Linux, or macOS.

The project has garnered more than 1200 stars from developers worldwide. To foster discussions about the language, Ring maintains an official Google Group (over 450 members). The group contains conversations covering various aspects of the language across more than 2800 topics [83].

External services tracking GitHub downloads indicate that the project has been downloaded over 18,000 times. Furthermore, a mirror exists for the project files hosted on Sourceforge. This mirror tracks download counts and their associated countries. Impressively, the downloads from this mirror have surpassed 62,500.

In Figure 16, we present the operating systems used during downloads, while in Figure 17, we present the countries that have the most downloads [84]. We expect that each programming language could be more popular in specific countries due to marketing reasons and the availability of educational resources. Many YouTube videos about the Ring language are presented in Arabic by Egyptian developers and YouTubers. This could also be one of the reasons Egypt has more users than other countries.

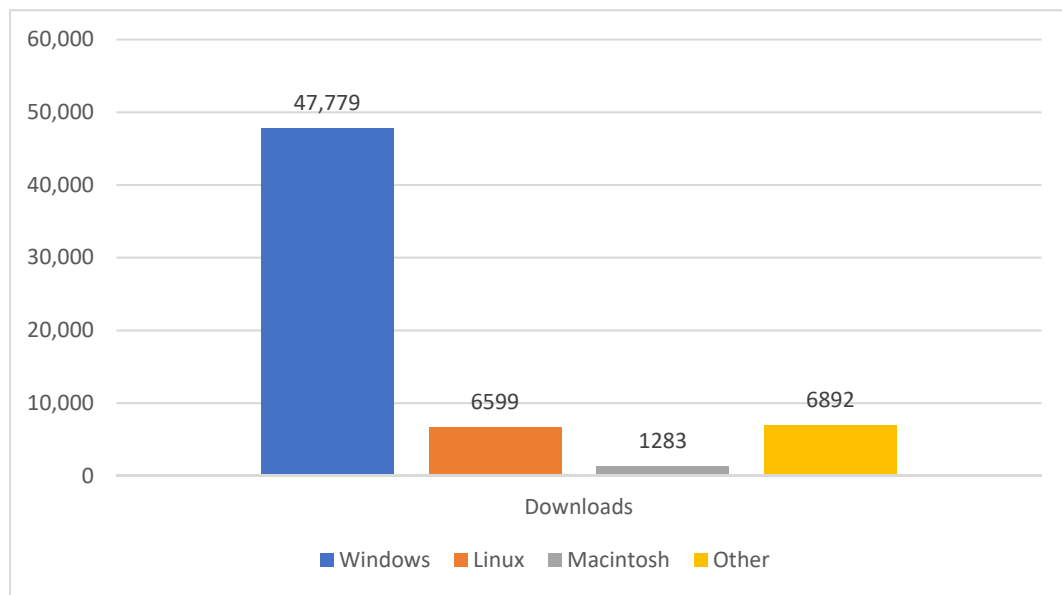


Figure 16. Ring downloads statistics from Sourceforge (grouped by the Operating System).

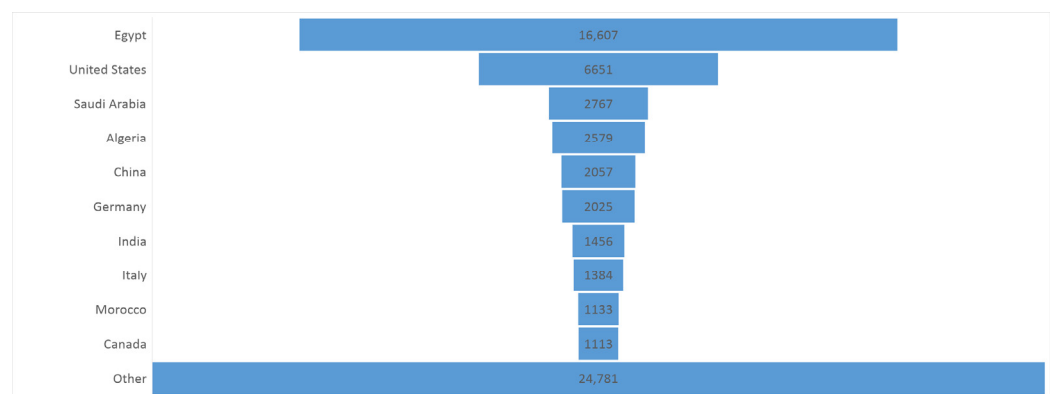


Figure 17. Ring downloads statistics from Sourceforge (grouped by the Country).

4.5. Use Cases and Printed Books (2017–2024)

In Table 9, we present some of the use-cases of the proposed programming language and environment. These uses-cases are related to different domains like Front-end applications for Machine Learning models, Games development, Text/Data processing, and Web development. We selected just one or two use-cases for each domain to avoid unnecessary duplication. For more projects, the Ring language is distributed with over 80 applications/games/tools.

Table 9. Some use cases for the Ring programming language and environment.

Ref.	Type	Domain	Description
[85]	Research Paper	Front-end apps for ML Models	Predicting citations count
[86]	Research Paper	Front-end apps for ML Models	Predicting game result
[87]	Printed Book (USA)	Games Development	Shooter Game
[88]	Steam Game	Games Development	Puzzle Game
[89]	Research Paper	Text/Data Processing apps	Predicting impedance
[90]	Printed Book (Egypt)	Text/Data Processing apps	Arabic Poetry Analysis
[91]	YouTube Videos	Desktop/Web development	Free course
[92]	Research Paper	LLMs Training	Dataset preparation

The first two use-cases are based on using the form designer and the standard libraries like GUILib, InternetLib, and JSONLib, to create front-end applications for machine learning models [85,86]. These applications could offer a user-friendly interface that receives input from users. The input is then transmitted to the machine learning model over the internet, and the resulting prediction is returned in JSON format. Afterward, the application processes this data and displays the outcome. Additionally, the GUI (Graphical User Interface) could include features such as data visualization, statistics, or a display for the dataset using the grid control, as demonstrated in Figure 18.

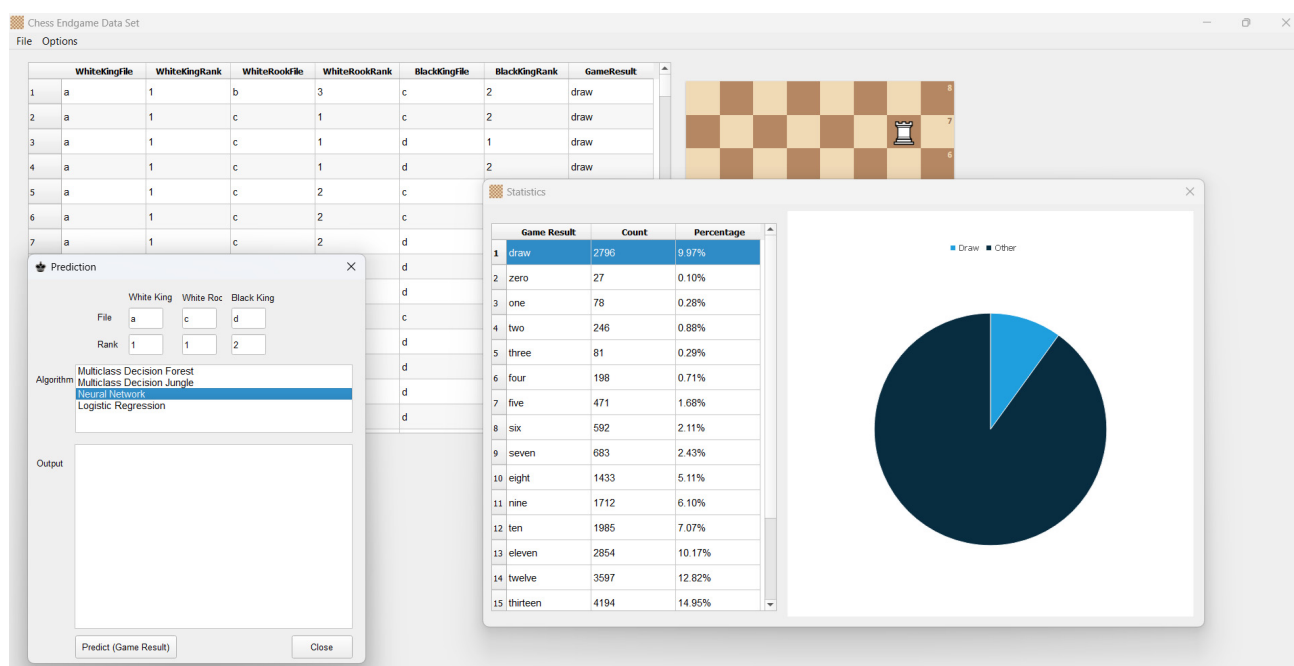


Figure 18. A GUI application developed using the Ring language.

The third use-case is about using the Ring programming language for 2D game development. This is explained through a printed English book (In the USA). The book contains nine chapters and is over 600 pages. The source code is available online through a GitHub project [87].

Figure 19 shows a puzzle game available on the Steam platform, written in Ring code and utilizing the Allegro and OpenGL libraries. The game, titled "Gold Magic 800", comprises 44 levels [88]. These levels have been meticulously designed by a specific level designer, which was also developed using Ring. Notably, the Level editor employs the Qt library. This game serves as an excellent example of how different libraries provided by the Ring language can be seamlessly mixed within the same project.

In [89], the Ring language is used to prepare a dataset before using it to train a machine learning model. Another use-case is developing a Ring program that analyzes Arabic poetry. The application contains over 3000 lines of Ring source code and is explained in detail in a printed Arabic book (In Egypt) [90].

In [91], A YouTube channel with over 350 K subscribers provided over 500 videos about the Ring programming language. These videos start by explaining the language fundamentals and how to apply the different programming paradigms using it. The videos cover desktop and web development, too. In [92], the authors used Ring language samples and documentation to train LLMs how to write Ring programs.



Figure 19. The GoldMagic800 game—A puzzle game developed using RingAllegro.

4.6. Visual Implementation (2013–2024)

In Table 10, we present the results of our visual implementation. The Table includes details for each visual source file: the amount of storage used on the hard disk, the memory used by PWCT after loading the file, the number of visual components, the count of steps within the steps tree, the lines of code in the generated source files (.c), and the lines of code in the generated header files (.h) if the visual source also generates such files. Finally, we provide the total lines of source code (without comments/blank lines) generated by the visual source file.

Table 10. Results of using the PWCT visual programming language to implement Ring compiler and virtual machine.

Modules	File Name	Storage (MB)	Memory (MB)	Components	Steps	Visible Steps	Comment	LOC
Loader	ring	2.07	21.5	211	320	287	35	236
	state	7.7	32	513	841	720	77	640
General Library	general	4.32	24.6	211	388	321	18	287
	hashtable	2.27	35	189	322	268	16	251
	item	4.04	24.3	231	440	356	54	301
	items	0.54	19.3	52	87	74	3	63
	list	10.85	40.1	969	1798	1432	118	1378
	string	3.94	26.6	271	497	399	18	383
Compiler	hashlib	1.72	25.5	54	81	70	3	59
	codegen	5.74	28.4	425	700	588	68	543
	expr	14.52	46.8	705	1263	1059	155	918
	objfile	5.98	29.1	524	934	757	85	606
	parser	3.61	23.9	327	460	415	49	372
	scanner	10.18	37.9	790	1318	1097	75	1006
	stmt	11.59	41.2	913	1603	1376	278	1132

Table 10. Cont.

Modules	File Name	Storage (MB)	Memory (MB)	Components	Steps	Visible Steps	Comment	LOC
Virtual Machine	vm	14.18	48.1	1498	2285	1992	278	1655
	vmapi	7.23	31.6	522	874	744	103	673
	vmduprange	0.8	21.2	70	127	104	6	94
	vmerror	1.5	20.7	139	265	220	37	186
	vmeval	2.48	21.8	233	428	371	81	295
	vmexit	1.82	21.1	83	167	132	15	119
	vmexpr	12.28	43.8	986	1817	1445	57	1383
	vmext	10.5	36.2	34	72	59	9	43
	vmfuncs	7.78	32.7	498	1000	838	206	675
	vmgc	11.64	41.9	922	1746	1422	240	1264
	vmjump	1.97	21.3	119	231	183	11	170
	vmlists	7.21	30.7	379	674	549	36	513
	vmoop	11.13	39.9	820	1497	1268	215	1081
	vmperformance	3.45	25.6	192	332	273	24	255
	vmstackvars	6.85	30.4	487	997	770	81	697
	vmstate	5.5	27.2	385	619	578	146	435
	vmstrindex	0.69	19.6	49	78	67	2	61
	vmthread	1.59	44.4	148	262	219	32	191
	vmtrycatch	0.76	19.7	20	38	33	5	25
	vmvars	8.51	33.8	362	685	569	91	497
Built-in Functions	vminfo_ext	6.06	28.1	289	443	389	24	360
	dll_ext	10.74	37.1	89	147	123	4	112
	file_ext	9.57	36.9	688	1235	991	26	961
	genlib_ext	22.75	66.6	1732	2965	2438	169	2308
	list_ext	5.65	28.5	531	982	782	35	740
	math_ext	3.93	25.2	336	649	489	3	497
	os_ext	5.25	26.8	313	572	464	23	427
	refmeta_ext	8.05	33.5	636	1075	886	26	851

Each visual source file belongs to one of the modules, such as Loader, General Library, Compiler, Virtual Machine, or the built-in functions. PWCT stores each visual source file in two files: *.SSF and *.FPT. The storage size listed in the table represents the summation of the file sizes of both files. The “components” column includes the total number of components used within the visual source file, even accounting for repeated usage of the same components. Each component corresponds to an interaction page (data-entry form) and may generate one or more steps.

We present a summary of the results in Table 11. Also, we highlight the results for each module in Figure 20. In this Figure, we notice that the Virtual Machine is the largest module while the optional “built-in functions” is the second largest module.

Table 11. Summary of visual implementation size.

Criteria	Total
Modules	5
Visual Source Files	43
Storage Size (MB)	278.95
Memory (MB)	1350.6
Visual Components	18,945
Steps	33,314
Steps (Visible)	27,617
Lines of Code (LOC)	24,743
Comments	3037
LOC including comments	27,780

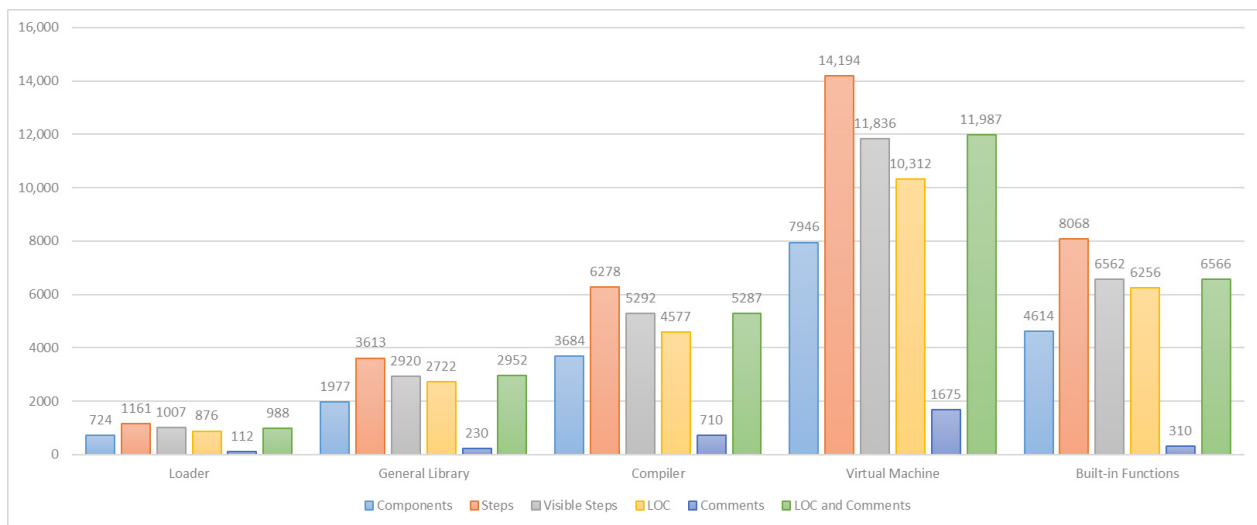


Figure 20. Visual implementation size for each module.

In Figure 21, we present the loading time required to display the visual representation and the code generation time for each visual source file. These values were measured 10 times for each file. In Figure 22, we present the code generation time for large visual source files. The time is measured in seconds, and tests are performed using a Victus Laptop [13th Gen Intel(R) Core(TM) i7-13700H, Windows 11, PWCT 1.9].

File Name	LT1	CGT1	LT2	CGT2	LT3	CGT3	LT4	CGT4	LT5	CGT5	LT6	CGT6	LT7	CGT7	LT8	CGT8	LT9	CGT9	LT10	CGT10
ring	0.141	0.208	0.126	0.204	0.126	0.213	0.13	0.201	0.13	0.213	0.119	0.205	0.12	0.206	0.13	0.188	0.11	0.192	0.11	0.207
state	0.518	1.307	0.505	1.288	0.519	1.228	0.5	1.318	0.51	1.321	0.486	1.255	0.49	1.241	0.49	1.286	0.505	1.227	0.534	1.259
general	0.22	0.315	0.202	0.283	0.22	0.299	0.2	0.298	0.21	0.299	0.204	0.313	0.22	0.298	0.22	0.299	0.204	0.289	0.204	0.314
hashtable	0.173	0.211	0.142	0.205	0.157	0.214	0.15	0.205	0.13	0.205	0.141	0.22	0.14	0.203	0.14	0.205	0.125	0.204	0.142	0.204
item	0.236	0.385	0.22	0.346	0.219	0.375	0.22	0.377	0.22	0.377	0.221	0.393	0.22	0.383	0.22	0.365	0.219	0.394	0.204	0.371
items	0.031	0.017	0.037	0.015	0.032	0.016	0.05	0.016	0.05	0.016	0.032	0.016	0.03	0.031	0.05	0.031	0.047	0.006	0.047	0.016
list	1.002	5.564	0.959	5.646	0.958	5.633	0.96	5.627	0.96	5.632	0.989	5.605	0.98	5.641	0.99	5.936	0.982	5.554	0.975	5.612
string	0.252	0.457	0.236	0.472	0.252	0.489	0.25	0.489	0.24	0.471	0.252	0.486	0.24	0.485	0.24	0.488	0.236	0.481	0.236	0.487
hashlib	0.063	0.015	0.062	0.016	0.041	0.015	0.05	0.016	0.05	0.016	0.047	0.016	0.05	0.015	0.05	0.016	0.047	0.016	0.047	0.015
codegen	0.377	0.865	0.346	0.91	0.345	0.834	0.35	0.896	0.35	0.928	0.346	0.871	0.35	0.881	0.35	0.906	0.346	0.853	0.363	0.878
expr	0.999	2.864	1.043	2.818	1.024	2.825	1.04	2.854	1.04	2.84	1.034	2.77	1.04	2.798	1.06	2.854	1.035	2.843	1.039	2.778
objfile	0.441	1.668	0.439	1.675	0.441	1.528	0.44	1.725	0.44	1.555	0.452	1.654	0.44	1.557	0.44	1.718	0.441	1.551	0.455	1.671
parser	0.204	0.394	0.204	0.393	0.189	0.393	0.19	0.401	0.2	0.401	0.188	0.408	0.19	0.393	0.19	0.382	0.189	0.377	0.188	0.377
scanner	0.846	2.975	0.737	3.011	0.726	3.028	0.75	3.026	0.72	2.981	0.731	2.986	0.74	3.006	0.74	2.982	0.74	3.01	0.736	3.025
stmt	0.956	4.468	0.942	4.402	0.95	4.44	0.86	4.5	0.93	4.453	0.943	4.44	0.99	4.505	0.88	4.439	0.897	4.661	0.972	4.437
vm	1.102	8.629	1.082	8.681	1.099	8.573	1.1	8.561	1.13	8.625	1.184	8.525	1.17	8.587	1.15	8.629	1.135	8.547	1.102	8.576
vmapi	0.471	1.377	0.472	1.321	0.471	1.401	0.5	1.34	0.49	1.371	0.47	1.385	0.47	1.358	0.5	1.403	0.47	1.395	0.488	1.338
vmduprange	0.063	0.041	0.047	0.032	0.047	0.032	0.05	0.04	0.06	0.045	0.063	0.039	0.06	0.045	0.06	0.048	0.063	0.047	0.047	0.48
vmerror	0.094	0.16	0.11	0.141	0.11	0.14	0.11	0.14	0.11	0.141	0.11	0.155	0.11	0.142	0.11	0.141	0.11	0.141	0.11	0.141
vmeval	0.157	0.327	0.15	0.362	0.137	0.357	0.16	0.366	0.16	0.363	0.157	0.362	0.16	0.345	0.14	0.347	0.157	0.362	0.157	0.361
vmexit	0.079	0.063	0.078	0.062	0.094	0.048	0.08	0.063	0.08	0.069	0.079	0.063	0.08	0.048	0.08	0.062	0.079	0.063	0.087	0.058
vmexpr	1.098	5.743	1.161	5.725	1.144	5.842	1.2	5.805	1.14	5.7	1.163	5.772	1.18	5.722	1.16	5.754	1.165	5.728	1.161	1.754
vmext	0.106	0.026	0.11	0.016	0.094	0.015	0.09	0.031	0.1	0.015	0.094	0.016	0.1	0.016	0.11	0.016	0.094	0.015	0.091	0.016
vmfuncs	0.565	1.84	0.549	1.818	0.565	1.748	0.55	1.868	0.57	1.811	0.582	1.815	0.57	1.83	0.58	1.85	0.549	1.826	0.55	1.838
vmgc	0.977	5.235	1.084	5.252	1.034	5.243	1.04	5.266	1.05	5.237	1.067	5.269	1.05	5.413	1.06	5.305	1.067	5.213	1.086	5.255
vmjump	0.126	0.112	0.11	0.111	0.11	0.109	0.11	0.103	0.11	0.127	0.11	0.11	0.09	0.11	0.11	0.11	0.11	0.109	0.126	0.122
vmlists	0.442	0.815	0.471	0.784	0.455	0.849	0.44	0.866	0.45	0.879	0.458	0.843	0.44	0.879	0.45	0.882	0.449	0.878	0.465	0.817
vmloop	0.898	3.888	0.925	3.923	0.879	3.937	0.93	3.922	0.97	3.862	0.926	3.873	0.93	3.93	0.86	3.95	0.911	3.899	0.879	3.968
vmperformance	0.162	0.215	0.16	0.22	0.173	0.226	0.16	0.196	0.17	0.22	0.161	0.229	0.17	0.236	0.17	0.206	0.172	0.204	0.173	0.22
vmstackvars	0.551	1.804	0.534	1.777	0.533	1.819	0.55	1.851	0.53	1.772	0.534	1.819	0.55	1.806	0.54	1.8	0.534	1.757	0.549	1.775
vmstate	0.283	0.725	0.282	0.738	0.283	0.66	0.27	0.676	0.28	0.723	0.283	0.733	0.28	0.662	0.28	0.674	0.28	0.724	0.268	0.707
vmstrindex	0.047	0.007	0.047	0.016	0.032	0.015	0.05	0.016	0.05	0.016	0.031	0.016	0.05	0.016	0.05	0.007	0.047	0.015	0.047	0.022
vmthread	0.11	0.127	0.094	0.142	0.095	0.142	0.09	0.141	0.11	0.142	0.094	0.143	0.1	0.149	0.1	0.142	0.094	0.142	0.094	0.142
vmtrycatch	0.032	0.016	0.032	0.016	0.031	0.006	0.03	0.016	0.04	0.01	0.017	0.004	0.03	0.015	0.03	0.015	0.031	0.004	0.016	0.016
vmvars	0.487	0.911	0.503	0.894	0.526	0.863	0.49	0.867	0.5	0.898	0.517	0.92	0.5	0.891	0.49	0.839	0.488	0.879	0.502	0.895
vminfo_ext	0.284	0.371	0.283	0.378	0.283	0.362	0.3	0.36	0.28	0.366	0.277	0.362	0.28	0.363	0.3	0.348	0.282	0.388	0.267	0.376
dll_ext	0.205	0.042	0.188	0.047	0.171	0.048	0.19	0.047	0.19	0.047	0.172	0.047	0.19	0.039	0.19	0.043	0.178	0.047	0.174	0.047
file_ext	0.81	2.704	0.75	2.669	0.802	2.666	0.77	2.713	0.82	2.656	0.747	2.661	0.76	2.675	0.8	2.661	0.8	2.651	0.754	2.732
genlib_ext	2.545	14.66	2.429	14.78	2.431	14.751	2.43	14.66	2.47	14.78	2.426	17.04	2.43	15.113	2.44	14.77	2.339	14.81	2.415	14.84
list_ext	0.439	1.728	0.456	1.759	0.445	1.73	0.44	1.762	0.44	1.778	0.44	1.761	0.44	1.71	0.46	1.717	0.439	1.735	0.456	1.683
math_ext	0.314	0.79	0.283	0.816	0.299	0.829	0.31	0.816	0.29	0.769	0.289	0.756	0.28	0.815	0.3	0.8	0.283	0.786	0.314	0.774
os_ext	0.33	0.642	0.329	0.647	0.33	0.629	0.31	0.598	0.3	0.597	0.314	0.598	0.31	0.597	0.33	0.643	0.298	0.603	0.314	0.646
refmeta_ext	0.611	2.01	0.595	2.015	0.596	2.062	0.62	2.055	0.59	2.038	0.596	2.047	0.6	2.014	0.6	2.043	0.628	1.993	0.698	2.013

Figure 21. The loading time (LT) and code generation time (CGT) for each visual source file.

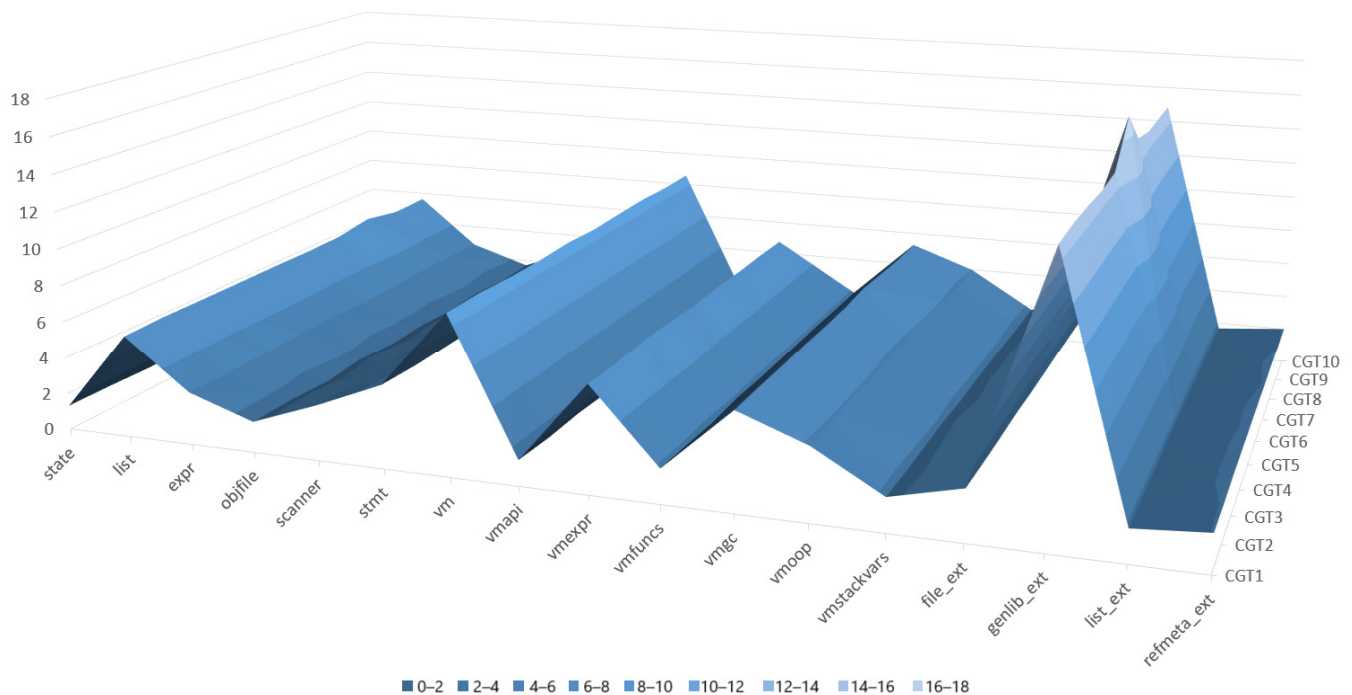


Figure 22. Code generation time (CGT) for large visual source files (measured in seconds).

4.7. Lightweight Implementation

Developing a lightweight programming language is not just about providing a language with a small implementation. It is merely the beginning, and we must pay attention to the growth in the language size over time. In Table 12, we present the growth percentage in implementation size for the Ring programming language and other known lightweight programming languages. The table presents the LOC of the first release and the LOC of the latest release. The LOC includes the compiler, VM, and the built-in functions. The growth in code size can be attributed to several factors, including fixing bugs, adding new features, performance improvements, the expansion of libraries and built-in capabilities, and enhancements in compatibility and interoperability.

Table 12. Growth in implementation size.

Language	Period	Implementation	LOC (FR)	LOC (LR)	Growth
Ring	2016–2024	C	16,402	24,743	51%
mRuby	2014–2024	C	18,134	23,742	31%
Squirrel	2004–2022	C++	9311	13,991	50%
Lua	1993–2024	C	5603	20,081	258%

Since Ring is designed to be a lightweight language, we have monitored the growth of the implementation size over the years. From 2016 to 2024, the implementation size has increased from 16 KLOC in Ring 1.0.0 to 24.7 KLOC in Ring 1.21.2, as demonstrated in Figure 23. The growth percentage in the implementation size is 51%. In Figure 24, we present the code size for the Lua Compiler/VM. The source code was written from 1993 to 2024, and the implementation size increased from 5.6 KLOC in Lua 1.0.0 to 20 KLOC in Lua 5.4.7. The growth percentage in implementation size is 258%.

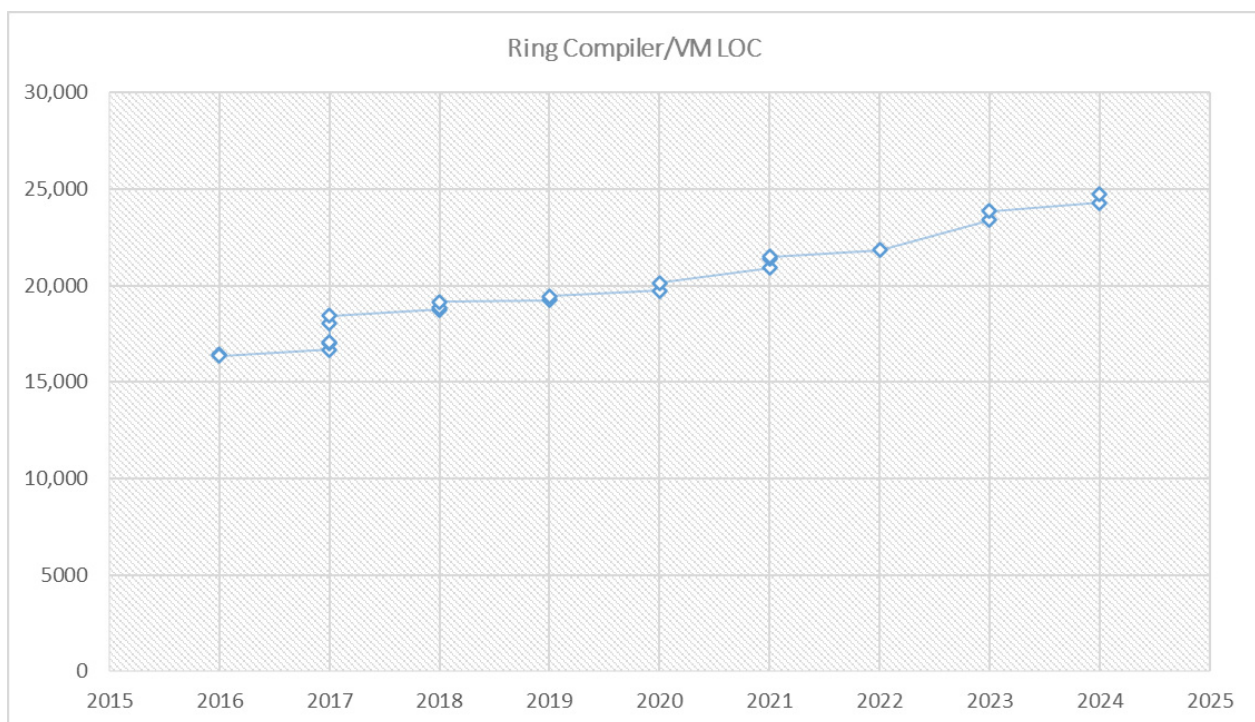


Figure 23. Generated code size for Ring Compiler/VM from 2016 to 2024.

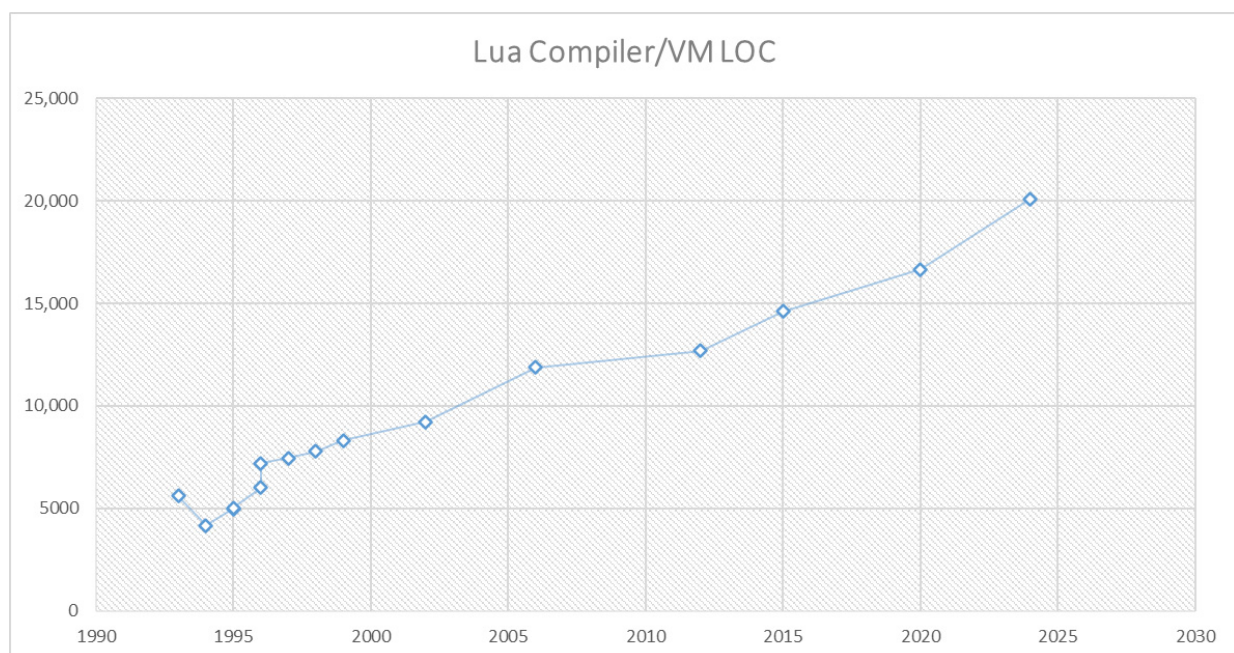


Figure 24. Code size for Lua Compiler/VM from 1993 to 2024.

In Figure 25, we present the generated code size for the Ring Compiler/VM for different Ring releases. The textual source code is generated in ANSI C and can be used by traditional programmers who may prefer text-based coding. This approach also enables adoption in settings where visual programming tools are less practical.

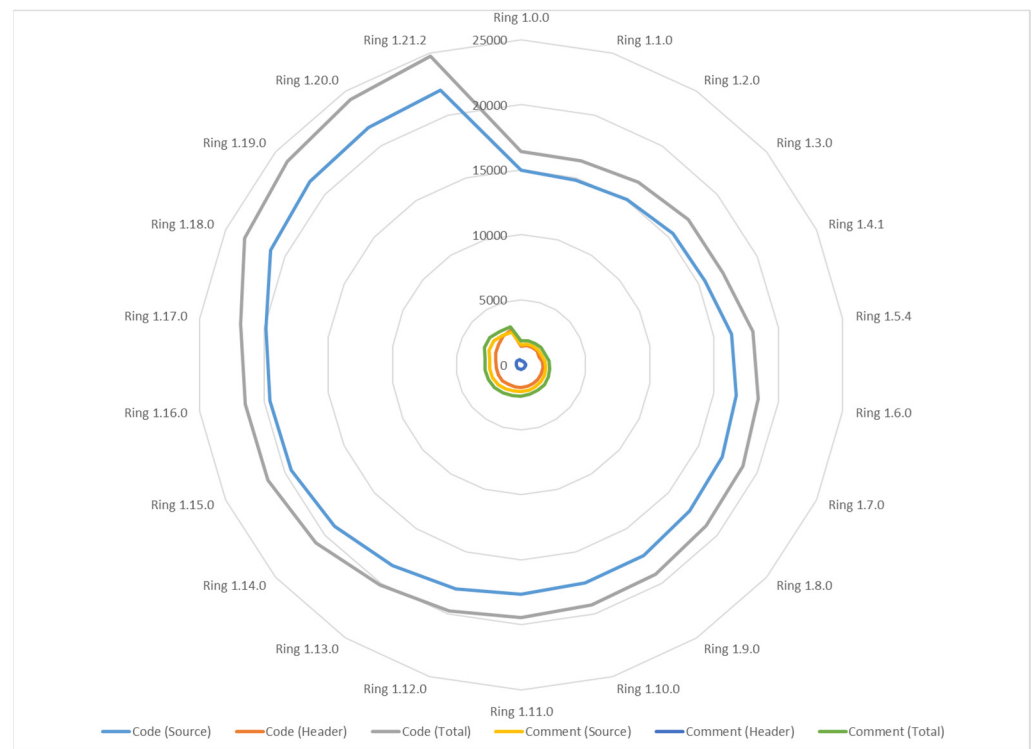


Figure 25. Generated code size for Ring Compiler/VM from Ring 1.0.0 to Ring 1.21.2.

4.8. Performance Benchmarks

In Table 13, we provide a benchmark comparison of various versions of the Ring programming language (Ring 1.17, Ring 1.19, and Ring 1.21), including its WebAssembly implementations on Edge and Chrome browsers, against VFP 9.0 and Python 3.13. Tests are performed using a Victus Laptop [13th Gen Intel(R) Core(TM) i7-13700H, Windows 11].

Table 13. Performance benchmarks.

Benchmark	Ring 1.17 (2022)	Ring 1.19 (2023)	Ring 1.21 (2024)	Ring 1.21 (We- bAsm/Edge)	Ring 1.21 (We- bAsm/Chrome)	VFP 9.0	Python 3.13
Loop (500 K)	9	5	4	7	7	8	7
Loop (1 M)	18	11	9	13	13	15	14
Loop (10 M)	185	113	91	133	132	47	149
Loop (100 M)	1896	1154	954	1362	1332	595	1534
MathMax (100 K)	136	25	7	11	12	16	7
MathMax (1 M)	1384	245	69	117	119	94	66
MathMax (10 M)	13,847	2474	708	1161	1204	909	776
MathMax (100 M)	139,373	24,868	7178	11,833	11,935	8968	7315
FuncCall (100 K)	111	19	4	10	9	16	3
FuncCall (1 M)	1134	194	39	97	94	110	32
FuncCall (10 M)	11,337	1943	398	1001	962	1102	444
FuncCall (100 M)	113,142	19,542	4058	10,164	9563	11,214	3297
FibDP (500)	6	5	3	3	4	6	0.1
FibDP (700)	11	10	5	5	6	13	0.3
FibDP (1000)	21	19	10	10	11	15	0.4
FibDP (1200)	29	27	15	15	17	16	0.4
FibDP (500)	6	5	3	3	4	6	0.1
FibRec (20)	22	6	1	2	3	16	0.3
FibRec (25)	244	65	13	22	23	31	3
FibRec (30)	2887	763	148	252	254	377	39
FibRec (35)	33,847	8660	1691	2795	2810	4357	431
ListFill (100 K)	10	10	6	11	12	14	5
ListFill (500 K)	54	50	31	56	56	30	19
ListFill (1 M)	108	99	62	112	113	63	35
ListFill (10 M)	1085	1007	643	1143	1145	565	376

The benchmarks cover a range of computational tasks, including looping (Loop), mathematical calculations (MathMax), function calls (FuncCall), dynamic programming Fibonacci calculations (FibDP), recursive Fibonacci calculations (FibRec), and list filling (ListFill), with varying input sizes. These benchmarks are designed to reflect features that are very common in many programs, ensuring their relevance and applicability across different use cases. VFP was selected for this comparison because it is a multiparadigm, dynamic language used in the development of PWCT. Python 3.13 was selected due to its popularity and versatility as a dynamic language that supports multiple programming paradigms. The performance, measured in milliseconds, indicates substantial improvements in the newer versions of Ring, particularly in Ring 1.21. For instance, the execution time for FuncCall (100 M) decreased dramatically from 113,142 ms in Ring 1.17 to 4058 ms in Ring 1.21. Figure 26 presents the performance of this benchmark.

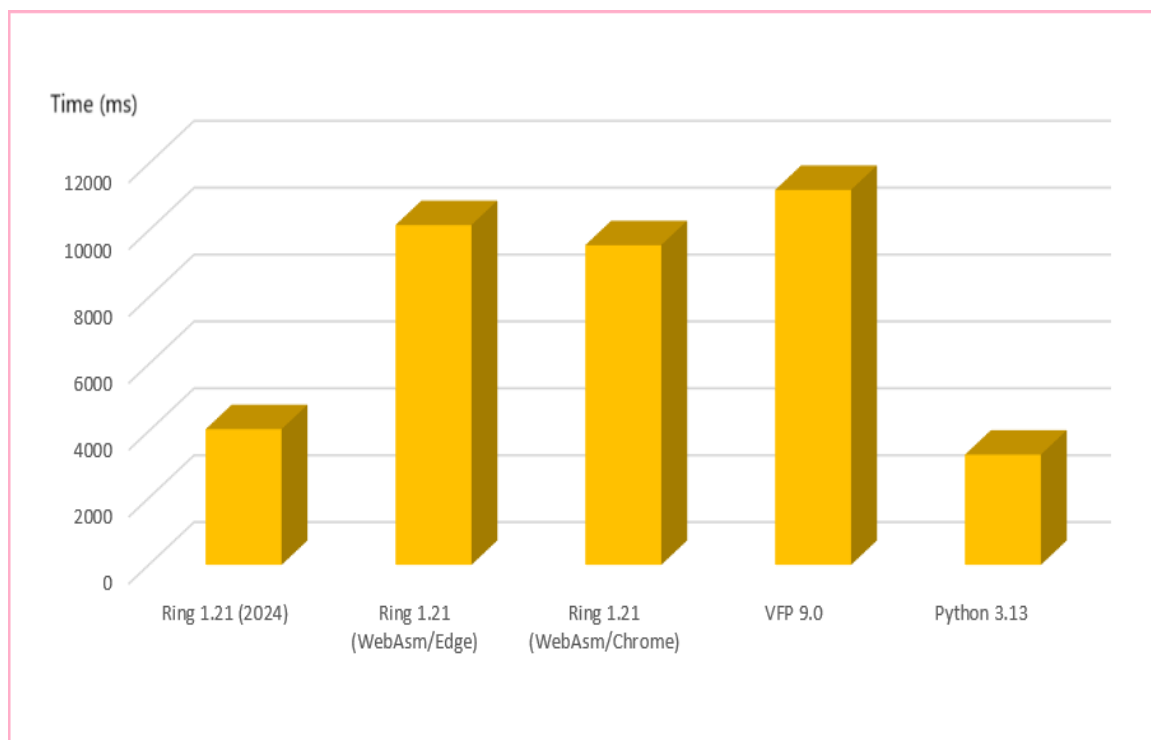


Figure 26. Function call (100 M) benchmark for Ring, VFP, and Python.

WebAssembly implementations show a slight increase in time compared to native executions. Overall, the data highlights significant performance enhancements in newer versions of Ring and offers a comparison of the efficiency of different programming environments. With respect to Ring support for microcontrollers, which is relatively new (first support started with Ring 1.21, released in September 2024), the performance results for Ring 1.21 running on the Raspberry Pi Pico reveal some interesting insights when compared to Ring 1.21 on a desktop. The Loop (500 k) benchmark was completed in 3.35 s, while MathMax (100 k) and FuncCall (100 k) took 3.54 s and 3.32 s, respectively. The Fibonacci Recursive (FibRec) at 25 iterations took 5.81 s, and the Dynamic Programming Fibonacci (FibDP) at 500 iterations was notably fast at 0.89 s. However, the ListFill (100 k) benchmark resulted in an “OUT OF MEMORY” error, which is expected given that the Raspberry Pi Pico has only 264 KB of SRAM. A dynamically typed language like Ring may encounter challenges on resource-constrained devices such as the Raspberry Pi Pico due to its limited memory and processing power.

Ring is distributed with support for game programming libraries, which enable us to create benchmarks for graphics and animations. In Figure 27, we demonstrate different frames from the Waving Cubes sample provided by the RayLib open-source library. This

sample presents an animation of 3375 cubes by changing their position, color, and size. In Table 14, we present the performance results for this sample in Ring, C, and Python. C demonstrates the highest efficiency with 480 frames per second (FPS). Ring 1.21 significantly improves upon its predecessor, achieving 170 FPS compared to Ring 1.19's 40 FPS, showcasing notable advancements in performance. Python 3.13 provides 85 FPS.

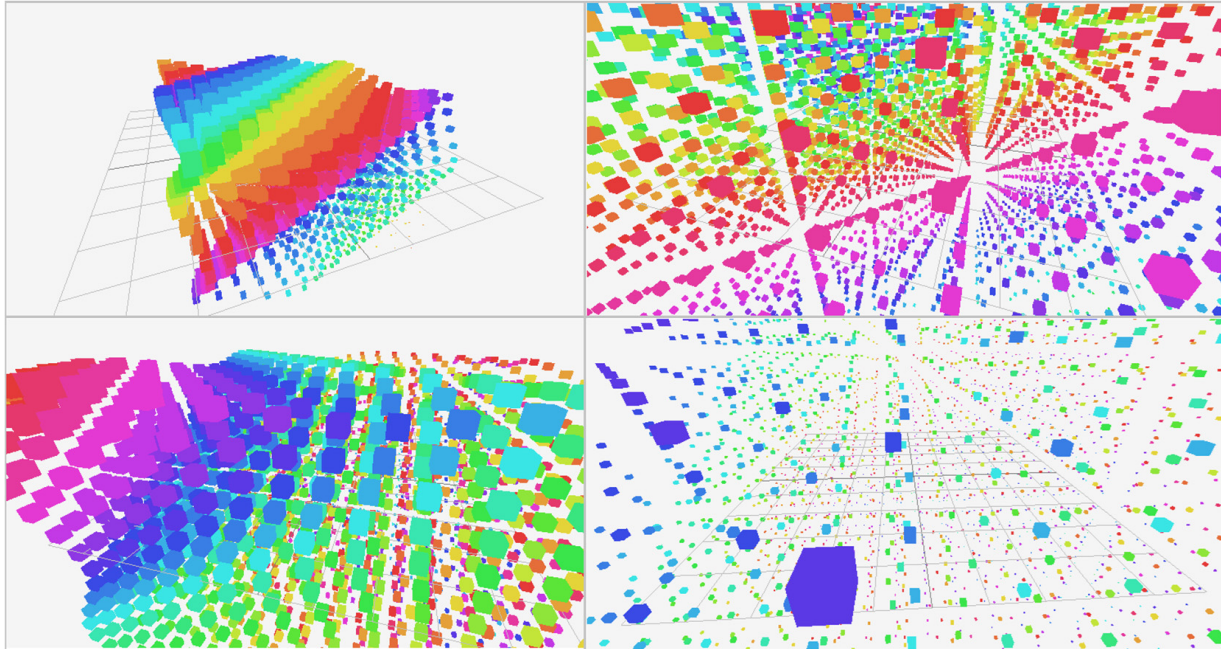


Figure 27. Different frames from the waving cubes animation.

Table 14. The Waving Cubes benchmark.

Language	FPS (Min)	FPS (Max)
C	470	480
Ring 1.21	161	170
Python 3.13	80	85
Ring 1.20	33	40

The Binding Generator for Ring extensions is the first significant program written in the Ring language itself and serves as an example and benchmark for file and text processing. The generator comprises 1407 lines of Ring code. It operates very efficiently by processing configuration files and generating the C/C++ code required for the extensions. The largest extension we have with the Ring language is RingQt, which generates over 211,000 lines of C/C++ code in just 3.42 s, as demonstrated in Table 15. Other extensions are smaller, and their code-generation process is completed in less than a second.

Table 15. Using the Code Generator to generate RingQt source code.

Variable	Value
Extension	RingQt
Configuration Files (Input)	439
Input Size	478 KB
Generated Files	197
Generated Lines of Code	211,174
Output Size	6.27 MB
Processing Time	3420 ms

The Ring IDE is designed as a project that includes a Code Editor, Form Designer, Web Browser, and a “Find in Files” application. Over the past eight years, it has proven to be a robust and reliable tool, supporting the development of all Ring samples and applications distributed with the language. The IDE’s performance has consistently been impressive, with no notable issues encountered during regular use. To ensure its reliability, a stress test was conducted by opening all Ring applications and samples distributed with the language. Memory usage was observed to be 348 MB at startup, slightly increasing to 365 MB after opening 78 applications (253 files, totaling 76,924 lines of code) and further to 439 MB after opening each file in the samples (1302 files, totaling 65,563 lines of code). This increase is attributed to the autocomplete feature, which caches all the words in each opened file. Tests are performed using a Victus Laptop [13th Gen Intel(R) Core(TM) i7-13700H, Windows 11, and Ring 1.21.2].

Additionally, the performance of opening and displaying a source code file was less than 250 ms for most files, as shown in Table 16. This performance depends on the file size. This demonstrates the Ring IDE’s capability to handle extensive development tasks without compromising performance. However, we continue to state in the Ring Group that the Ring IDE is just an example of Ring usage. Ring, as a language, can be used with different code editors based on programmer preference, which makes sense if the programmer is using multiple languages in a project. The Ring IDE only supports Ring source code files, which is a limitation in situations requiring a mix of programming languages.

Table 16. Loading and displaying files in Ring IDE.

Application/Sample	Size (LOC)	Loading Time (ms)
Analog Clock	256	36
Image Pixel	548	66
Knight Tour	646	67
Othello	780	78
Visualize Sort	817	81
Game Of Life	903	90
Laser	1051	94
Checkers	1354	124
Get Quotes History	3401	117
Discrete Fourier Transform	6417	203

5. Discussion

From Table 8, we notice that dozens of users learned the language through the available resources (documentation, samples, applications, and videos), and their feedback helped us grow the educational resources distributed with the language. For example, chapters such as FAQ, Scope Rules, Performance Tips, and General Information were added and enhanced based on this feedback. Also, many graphics programming samples (OpenGL Camera and background, Collision detection, Chess 3D, etc.) are developed by one of those users. Another important lesson learned from being close to users and responding to their feedback is that this not only encourages more people to get involved and report issues but also motivates users to become active contributors to the open-source project. However, we acknowledge the limitations and potential biases, especially regarding the demographic homogeneity (predominantly male participants) and the regional limitations of early feedback, which could affect the generalizability of the results.

From the statistics in Figures 16 and 17, we can conclude that Ring, as a programming language and research prototype, has been tried by thousands of users [83,84]. Based on the use cases demonstrated in Table 8, the Ring programming language has proven to be versatile. It has been effectively utilized in a variety of domains, including desktop development (Ring IDE and Chess End Game application), game development (Shooter Game and Gold Magic 800 puzzle game), and data analysis (Arabic poetry analysis application). The language’s lightweight and embeddable nature, combined with its support for many

programming paradigms, allows for rapid development [85–92]. However, this versatility still needs to benefit more from the language’s ability to create domain-specific languages and localization packages.

Abstraction is a known dimension in the Cognitive Dimensions Framework (CDF), which is utilized in many research studies for the usability analysis of visual programming languages [93]. Abstraction involves grouping elements or entities into a single entity to either reduce viscosity (making it less difficult to modify) or align the notation with the user’s conceptual structure. Abstractions are useful for modification and transcription tasks (copying content from one structure to another). They play a crucial role in visual implementation, as they significantly influence ease of use and can also increase protection against errors [94,95].

From the results in Table 11, we observe that Ring visual implementation comprises 18,945 visual components, which in turn generated 24,743 lines of code. This finding highlights the significant advantage of visual implementation: it increases the abstraction level by 23.5% while concealing unnecessary details. Specifically, this advantage becomes apparent when specifying component data through interaction pages (data-entry forms). These interaction pages generate and update the steps tree based on the provided data. However, a different scenario emerges when considering the generated steps tree. The steps tree aims to provide additional information about the program structure and related details, resulting in a total number of steps that exceed the lines of code, as demonstrated in the “Steps” column. On the other hand, PWCT offers a “Read Mode” that allows users to hide many of these implementation details. In this mode, the “Visible Steps” column shows a count slightly less than the total lines of code. Despite this difference, the Steps Tree has a clear advantage: it facilitates easy interaction with groups of steps. Its tree structure directly provides two dimensions of interaction—siblings and children—which enhances usability and navigation within the visual implementation.

This higher level of abstraction translates into a more productive development process by allowing developers to focus on the overall structure and functionality of the program rather than getting bogged down in the minutiae of code syntax. The visual components provide a more intuitive and accessible interface, making it easier for developers to understand and modify the codebase. Additionally, this approach improves usability by reducing the likelihood of syntax errors and simplifying the debugging process. [13,45,76].

However, we have observed certain disadvantages while using PWCT:

- **Large Storage Size:** Visual implementations tend to occupy more storage space compared to their textual counterparts. This is an important consideration, especially when dealing with large projects;
- **Memory Requirements for Multiple Instances:** The PWCT environment is designed to open one visual source file at a time. To work with multiple files simultaneously, you need to run multiple instances of PWCT. Unfortunately, this approach comes with a memory cost. Opening all the visual source files related to the Ring compiler and virtual machine implementation (43 files) requires approximately 1.3 GB of memory. This could become an issue for larger projects that contain more visual source files, where opening these files simultaneously for quick navigation could be problematic. This might be required when searching in multiple files. We also noticed that PWCT supports search/replace in a single visual source file, which is not practical for large projects. To work around this issue, we used external tools to search in the generated textual source code;
- **Limitations of the Steps Tree Editor:** The Steps Tree editor lacks support for drag-and-drop functionality. Moving steps within the tree is possible only through cut-and-paste operations. While this may not be a critical issue, it is worth noting for usability purposes because it forces using the keyboard to move steps faster from one location to another;

- **Performance Challenges with Large Visual Source Files:** When dealing with visual source files containing thousands of components, performance can become an issue. Loading such files or generating source code may exhibit slower behavior. For example, our largest file (`genlib_ext`) contains 1732 components and 2965 steps. Loading the file and displaying the visual representation takes over two seconds, while generating the source code takes over 14 s, as demonstrated in Figures 21 and 22. This results in slow development and iteration when tasks involve updating many visual source files;
- **No support for importing textual source code.** This missing feature introduces many limitations. If a programmer submits a GitHub pull request by modifying the textual source code of the Ring Compiler/VM, we cannot simply import these changes. Additionally, this is a barrier to integrating with AI tools that generate textual code. While in visual implementation, we could use external libraries provided through textual code (without visual implementation), the problem occurs when the generated source code (from visual implementation) is modified directly by other contributors in the project without making the change through visual programming first and then generating the textual source code. This leads to effort duplication. This demonstrates a limitation on collaborative development efforts. For users who might wish to integrate Ring with existing workflows or tools and are looking to modify the language implementation, we recommend making a choice and sticking to it: either use the visual implementation and make any changes through it first or use the generated textual source code and continue development based on it;
- **PWCT is designed to work only under Microsoft Windows.** The support for other operating systems is not native and requires extra tools (Like Wine for Linux).

Despite these challenges, our successful use of PWCT to develop and maintain the Ring programming language compiler and virtual machine demonstrates its value. However, addressing these scalability issues will be crucial if PWCT is to be adopted in larger projects in the future.

Suggestions to mitigate these challenges:

1. Separate the visual source into many files with clear names and purposes;
2. Keep each visual source file to fewer than a few thousand steps;
3. Open related visual source files according to the current task while closing unrelated visual source files (or not opening them) to provide easy navigation between PWCT instances through the operating system features;
4. External tools are needed when searching multiple generated source code files.

From Table 12 and Figures 23 and 25, we demonstrated the growth of the Ring language over eight years; while being a lightweight language, we noticed a growth in the implementation size from 16 KLOC in 2016 to 24 KLOC in 2024. This percentage of growth (51%) requires attention, and we could focus in the next years on reducing the implementation size since the core features have been implemented and the implementation is stable and usable. With respect to adding new features, we will try to keep most of these new features in the libraries and the new domain-specific languages. From Tables 13–16, we notice that the performance of the Ring programming language has improved over time, and it is now fast enough for many use cases as a scripting language. However, improving Ring's performance remains a challenge, and we aim to provide optimizations and enhancements with each new release.

6. Conclusions and Future Work

In this research paper, we introduced Ring—a dynamically typed programming language developed and maintained for many years using visual programming, where the generated code is based on ANSI C. Ring combines a lightweight implementation with several advantages, such as a rich and versatile standard library, along with direct support for classes and object-oriented programming. Customization is a key feature of Ring, allowing developers to easily modify the language syntax multiple times. Moreover, Ring empowers the creation of domain-specific languages through features that extend object-oriented

programming. Beyond its language design, Ring provides a practical development environment and facilitates rapid GUI application development.

Ring is adaptable across diverse platforms. Rather than creating separate language implementations for specific contexts, the same Ring implementation serves a wide range of environments. From desktop systems to WebAssembly and even 32-bit microcontrollers like the Raspberry Pi Pico, Ring addresses the problem of missing language features that exist in other implementations.

In summary, Ring emerges as a lightweight, versatile, and customizable dynamic language developed using visual programming, adapting seamlessly to the ever-evolving landscape of software development. The implementation based on visual programming increases the abstraction level by 23.5% and provides a more user-friendly implementation through visual programming advantages, such as avoiding syntax errors.

In the future, we plan to build multiple projects on top of the Ring programming language, such as a localization package for many human languages, various domain-specific languages for different fields, and a modern platform for visual programming. Our priority is to provide a complete translation of all language syntax and libraries into Arabic. Following this, we aim to develop a domain-specific language for GUI development, like the Supernova language, but based on Ring's features that extend object-oriented programming to support the creation of internal domain-specific languages. After that, we will focus on creating a visual programming language for game development.

Author Contributions: Conceptualization, M.S.F. and Y.A.A.; methodology, M.S.F. and Y.A.A.; software, M.S.F.; validation, M.S.F. and Y.A.A.; formal analysis, M.S.F. and Y.A.A.; investigation, M.S.F. and Y.A.A.; resources, M.S.F. and Y.A.A.; data curation, M.S.F. and Y.A.A.; writing—original draft preparation, M.S.F.; writing—review and editing, M.S.F. and Y.A.A.; visualization, M.S.F.; supervision, Y.A.A.; project administration, Y.A.A.; funding acquisition, Y.A.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data used to support the findings of this study are available from the corresponding author upon request.

Acknowledgments: The authors thank members of the Ring project team for their contributions in testing the software and adding more samples.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ANSI	American National Standards Institute
CDF	Cognitive Dimensions Framework
CGT	Code Generation Time
CSS	Cascading Style Sheets
DRY	Don't Repeat Yourself
DSL	Domain Specific Language
FAQ	Frequently Asked Questions
GIL	Global Interpreter Lock
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDSL	Internal Domain-Specific Language
IoT	Internet of Things
JSON	JavaScript Object Notation
KISS	Keep it simple and stupid
KLOC	Thousands of lines of code
LLM	Large Language Model
LOC	Lines of Code

LT	Loading Time
ML	Machine Learning
OOP	Object Oriented Programming
PWCT	Programming Without Coding Technology
REPL	Read-Eval-Print-Loop
SQL	Structured Query Language
SWIG	Simplified Wrapper and Interface Generator
VFP	Visual FoxPro
VI	Visual Implementation
VM	Virtual Machine
VPL	Visual Programming Language

References

1. Vanilla, O.; Yu, J.; Abdalla, H.B. Cesno: The Initial Design of a New Programming Language. In Proceedings of the International Conference on Computing, Control and Industrial Engineering, Wuhan, China, 21–22 June 2024; Springer Nature: Singapore, 2024; pp. 128–146.
2. Noone, M.; Mooney, A. Visual and textual programming languages: A systematic review of the literature. *J. Comput. Educ.* **2018**, *5*, 149–174. [\[CrossRef\]](#)
3. Wilhelm, R.; Seidl, H. *Compiler Design: Virtual Machines*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 10.
4. Mitchell, J.C. *Concepts in Programming Languages*; Cambridge University Press: Cambridge, UK, 2003.
5. Ortin, F.; Garcia, M.; Perez-Schofield, B.G.; Quiroga, J. The StaDyn programming language. *SoftwareX* **2022**, *20*, 101211. [\[CrossRef\]](#)
6. Gao, K.; Mei, G.; Piccialli, F.; Cuomo, S.; Tu, J.; Huo, Z. Julia language in machine learning: Algorithms, applications, and open issues. *Comput. Sci. Rev.* **2020**, *37*, 100254. [\[CrossRef\]](#)
7. Pang, A.; Anslow, C.; Noble, J. What programming languages do developers use? A theory of static vs dynamic language choice. In Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Lisbon, Portugal, 1–4 October 2018; pp. 239–247.
8. Callaú, O.; Robbes, R.; Tanter, É.; Röthlisberger, D. How developers use the dynamic features of programming languages: The case of smalltalk. In Proceedings of the 8th Working Conference on Mining Software Repositories, Honolulu, HI, USA, 21–22 May 2011; pp. 23–32.
9. Armstrong, J. A history of Erlang. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, San Diego, CA, USA, 9–10 June 2007; pp. 6–1–6–26.
10. Paulson, L.D. Developers shift to dynamic programming languages. *Computer* **2007**, *40*, 12–15. [\[CrossRef\]](#)
11. Kay, A.C. The early history of Smalltalk. In *History of Programming Languages—II*; Association for Computing Machinery: New York, NY, USA, 1996; pp. 511–598.
12. McCarthy, J. History of LISP. In *History of Programming Languages*; Association for Computing Machinery: New York, NY, USA, 1978; pp. 173–185.
13. Fayed, M.S.; Al-Qurishi, M.; Alamri, A.; Al-Daraiseh, A.A. PWCT: Visual language for IoT and cloud computing applications and systems. In Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing, Cambridge, UK, 22–23 March 2017; Volume 2017, pp. 1–5.
14. Dasgupta, S.; Hill, B.M. Learning to code in localized programming languages. In Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale, Cambridge, MA, USA, 20–21 April 2017; Volume 2017, pp. 33–39.
15. Maloney, J.; Resnick, M.; Rusk, N.; Silverman, B.; Eastmond, E. The scratch programming language and environment. *ACM Trans. Comput. Educ. (TOCE)* **2010**, *10*, 1–15. [\[CrossRef\]](#)
16. Bassil, Y. Phoenix—The Arabic Object-Oriented Programming Language. *arXiv* **2019**, arXiv:1907.05871.
17. Hamid, M. Design of an Arabic programming language (ARABLAN). *Comput. Lang.* **1995**, *21*, 191–201.
18. Gasolin. Zhpy Python Package. 2020. Available online: <https://pypi.org/project/zhpy/> (accessed on 11 October 2024).
19. de Mooij, G. Citrine Programming Language. 2014. Available online: <https://citrine-lang.org/> (accessed on 11 October 2024).
20. Mahmoud, F. Supernova Programming Language. 2010. Available online: <https://supernova.sourceforge.net/> (accessed on 11 October 2024).
21. Lyu, R.Y.; Kuo, Y.H.; Liu, C.N. Machine translation of English identifiers in python programs into traditional Chinese. In Proceedings of the 2016 International Computer Symposium (ICS), Chiayi, Taiwan, 15–17 December 2016; pp. 622–625.
22. Stanojević, M.; Stanojević, B. Lua APIs for mathematical optimization. *Procedia Comput. Sci.* **2024**, *242*, 460–465. [\[CrossRef\]](#)
23. Ierusalimsky, R.; de Figueiredo, L.H.; Celes, W. The evolution of Lua. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, San Diego, CA, USA, 9–10 June 2007; pp. 2–1–2–26.
24. Ierusalimsky, R. *Programming in Lua*; Roberto Ierusalimsky: Rio de Janeiro, Brazil, 2006.
25. Ierusalimsky, R.; De Figueiredo, L.H.; Filho, W.C. Lua—An extensible extension language. *Softw. Pract. Exp.* **1996**, *26*, 635–652. [\[CrossRef\]](#)
26. Nystrom, R. *Crafting Interpreters*; Genever Benning: Mumbai, India, 2021.

27. Apodaif, M.G.; Shohdy, A.; Farghal, A. Layout Automation of Variable Gain Amplifier Circuit based on TCL Language. *Sohag Eng. J.* **2024**, *4*, 87–99. [\[CrossRef\]](#)
28. Hammen, J. Bipscrip: A domain-specific scripting language for interactive music. In Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford, CA, USA, 23–26 March 2019.
29. Welch, B.B.; Jones, K.; Hobbs, J. *Practical Programming in Tcl and Tk*; Prentice Hall Professional: Hoboken, NJ, USA, 2003.
30. Wetherall, D.; Lindblad, C.J. Extending Tcl for Dynamic Object-Oriented Programming. In Proceedings of the Tcl/Tk Workshop, Toronto, ON, Canada, 6–8 July 1995; Volume 670.
31. Ousterhout, J.K. *Tcl: An Embeddable Command Language*; University of California, Berkeley, Computer Science Division: Berkeley, CA, USA, 1989.
32. Giorgi, F.M.; Ceraolo, C.; Mercatelli, D. The R language: An engine for bioinformatics and data science. *Life* **2022**, *12*, 648. [\[CrossRef\]](#) [\[PubMed\]](#)
33. Moler, C.; Little, J. A history of MATLAB. In *Proceedings of the ACM on Programming Languages*; Association for Computing Machinery: New York, NY, USA, 2022; Volume 4, pp. 1–67.
34. Granor, T.E.; Hennig, D.; Roche, T.; Martin, D. *Hacker's Guide to Visual FoxPro 7.0*; Hentzenwerke: Milwaukee, WI, USA, 2002.
35. Moore, A.D. *Python GUI Programming with Tkinter: Design and Build Functional and User-Friendly GUI Applications*; Packt Publishing, Ltd.: Birmingham, UK, 2021.
36. Hao, J.; Ho, T.K. Machine learning made easy: A review of scikit-learn package in python programming language. *J. Educ. Behav. Stat.* **2019**, *44*, 348–361. [\[CrossRef\]](#)
37. Summerfield, M. *Rapid GUI programming with Python and Qt: The Definitive Guide to PyQt Programming*; Pearson Education: London, UK, 2007.
38. Ruys, W.; Lee, H.; You, B.; Talati, S.; Park, J.; Almgren-Bell, J.; Yan, Y.; Fernando, M.; Biros, G.; Erez, M.; et al. A Deep Dive into Task-Based Parallelism in Python. In Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), San Francisco, CA, USA, 27–31 May 2024; pp. 1147–1149.
39. Eggen, R.; Eggen, M. Thread and process efficiency in python. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, NV, USA, 29 July–1 August 2019; pp. 32–36.
40. Masini, S.; Bientinesi, P. High-performance parallel computations using python as high-level language. In Proceedings of the Euro-Par 2010 Parallel Processing Workshops: HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC, Ischia, Italy, 31 August–3 September 2010; Revised Selected Papers. Springer: Berlin/Heidelberg, Germany, 2011; Volume 16, pp. 541–548.
41. Rockoff, L. *The Language of SQL*; Addison-Wesley Professional: Boston, MA, USA, 2021.
42. Attardi, J. *Modern CSS*; Springer: Berlin/Heidelberg, Germany, 2020.
43. Günther, S.; Cleenewerck, T. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In Proceedings of the 17th Conference on Pattern Languages of Programs, Irsee, Germany, 11–15 July 2012; pp. 1–35.
44. Maximilien, E.M.; Wilkinson, H.; Desai, N.; Tai, S. A domain-specific language for web apis and services mashups. In Proceedings of the Fifth International Conference on Service-Oriented Computing–ICSOC 2007, Vienna, Austria, 17–20 September 2007; Springer: Berlin/Heidelberg, Germany, 2007; Volume 5, pp. 13–26.
45. Fayed, M.S.; Al-Qurishi, M.; Alamri, A.; Hossain, M.A.; Al-Daraiseh, A.A. PWCT: A novel general-purpose visual programming language in support of pervasive application development. *CCF Trans. Pervasive Comput. Interact.* **2020**, *2*, 164–177. [\[CrossRef\]](#)
46. Suresh, P.; Daniel, J.V.; Parthasarathy, V.; Aswathy, R.H. A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment. In Proceedings of the 2014 International Conference on Science Engineering and Management Research (ICSEMR), Chennai, India, 27–29 November 2014; pp. 1–8.
47. Zikria, Y.B.; Ali, R.; Afzal, M.K.; Kim, S.W. Next-generation internet of things (iot): Opportunities, challenges, and solutions. *Sensors* **2021**, *21*, 1174. [\[CrossRef\]](#)
48. Fayed, M.S. Network generations and the security challenge in IoT applications. *arXiv* **2022**, arXiv:2201.01927.
49. Silangern, H.; Tanaka, K. Internet of Things Implemented with Mruby. In Proceedings of the International Conference on Computational Science and Its Applications, Hanoi, Vietnam, 1–4 July 2024; Springer Nature: Cham, Switzerland, 2024; pp. 165–181.
50. Tanaka, K.; Ogura, S.; Krishnamoorthy, R.; Sugiyama, K.I.; Kawahara, M. Flexibility and Productivity in IoT Programming: A Case Study with Mruby. In Proceedings of the International Conference on Advanced Research in Technologies, Information, Innovation and Sustainability, Madrid, Spain, 18–20 October 2023; Springer Nature: Cham, Switzerland, 2023; pp. 17–27.
51. Plauska, I.; Liutkevičius, A.; Janavičiūtė, A. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics* **2022**, *12*, 143. [\[CrossRef\]](#)
52. Halfacree, G.; Everard, B. *Get Started with MicroPython on Raspberry Pi Pico: The Official Raspberry Pi Pico Guide*; Raspberry Pi Press: London, UK, 2021.
53. Ray, P.P. An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet* **2023**, *15*, 275. [\[CrossRef\]](#)
54. Yan, Y.; Tu, T.; Zhao, L.; Zhou, Y.; Wang, W. Understanding the performance of WebAssembly applications. In Proceedings of the 21st ACM Internet Measurement Conference, Virtual Event, 2–4 November 2021; pp. 533–549.
55. Gallant, G. *WebAssembly in Action: With Examples Using C++ and Emscripten*; Simon and Schuster: New York, NY, USA, 2019.

56. Juric, S. *Elixir in Action*; Simon and Schuster: New York, NY, USA, 2024.
57. Vegi, L.F.D.M.; Valente, M.T. Code smells in Elixir: Early results from a grey literature review. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 580–584.
58. Deo, A.K.A.; Gupta, S.; Kundu, R.; Jaiswal, P.; Fatma, T.; Dehury, M.K. Performance and Metrics Analysis Between Python3 via Mojo. In Proceedings of the 2024 2nd International Conference on Sustainable Computing and Smart Systems (ICSCSS), Virtual Event, 10–12 July 2024; pp. 1291–1297.
59. Xurshid, M. Differences between Mojo and Python programming languages. In Proceedings of the Conference on Digital Innovation: Modern Problems and Solutions, Ho Chi Minh City, Vietnam, 21–22 June 2023.
60. Lutz, M. *Programming Python*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2001.
61. Thomas, D.; Fowler, C.; Hunt, A. *Programming Ruby—The Pragmatic Programmer's Guide*; Pragmatic Programmers, LLC: Raleigh, NC, USA, 2004; Volume 238.
62. Marson, R.L.; Jankowski, E. Build management with CMake. In *Introduction to Scientific and Technical Computing*; CRC Press: Boca Raton, FL, USA, 2016; pp. 119–132.
63. Spinellis, D. Git. *IEEE Softw.* **2012**, *29*, 100–101. [\[CrossRef\]](#)
64. Wirth, N. The programming language Pascal. *Acta Inform.* **1971**, *1*, 35–63. [\[CrossRef\]](#)
65. Cottom, T.L. Using SWIG to bind C++ to Python. *Comput. Sci. Eng.* **2003**, *5*, 88–97. [\[CrossRef\]](#)
66. Franz, M. Dynamic linking of software components. *Computer* **1997**, *30*, 74–81. [\[CrossRef\]](#)
67. Lazar, G.; Penea, R. *Mastering Qt 5: Create Stunning Cross-Platform Applications Using C++ with Qt Widgets and QML with Qt Quick*; Packt Publishing, Ltd.: Birmingham, UK, 2018.
68. Leff, A.; Rayfield, J.T. Web-application development using the model/view/controller design pattern. In Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference, Seattle, WA, USA, 4–7 September 2001; pp. 118–127.
69. Thau, R. Design considerations for the Apache Server API. *Comput. Networks ISDN Syst.* **1996**, *28*, 1113–1122. [\[CrossRef\]](#)
70. Westra, E. *Modular Programming with Python*; Packt Publishing, Ltd.: Birmingham, UK, 2016.
71. Stroustrup, B. What is object-oriented programming? *IEEE Softw.* **1988**, *5*, 10–20. [\[CrossRef\]](#)
72. le Clercq, A.; Almroth, K. *Comparison of Rendering Performance Between Multimedia Libraries Allegro, SDL and SFML*; KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science: Stockholm, Sweden, 2019.
73. Fahy, R.; Krewer, L. Using open source libraries in cross platform games development. In Proceedings of the 2012 IEEE International Games Innovation Conference, Rochester, NY, USA, 7–9 September 2012; pp. 1–5.
74. Smith, S.; Smith, S. How to Connect Pico to IoT. In *RP2040 Assembly Language Programming: ARM Cortex-M0+ on the Raspberry Pi Pico*; Apress: Berkeley, CA, USA, 2022; pp. 265–289.
75. Fayed, M.S. General-purpose visual language and information system with case-studies in developing business applications. *arXiv* **2017**, arXiv:1712.10281.
76. Chin, J.M.; Chin, M.H.; Van Landuyt, C. A String Search Marketing Application Using Visual Programming. *e-J. Bus. Educ. Scholarsh. Teach.* **2013**, *7*, 46–58.
77. Ierusalimsky, R.; De Figueiredo, L.H.; Celes Filho, W. The Implementation of Lua 5.0. *J. Univers. Comput. Sci.* **2005**, *11*, 1159–1176.
78. Ike-Nwosu, O. *Inside the Python Virtual Machine*; Leanpub: Victoria, BC, Canada, 2015.
79. Zagallo, T. A New Bytecode Format for JavaScriptCore. 2019. Available online: <https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/> (accessed on 12 October 2024).
80. Dahm, M. Byte code engineering. In *JIT'99: Java-Information-Tage 1999*; Springer: Berlin/Heidelberg, Germany, 1999; pp. 267–277.
81. Nässén, H.; Carlsson, M.; Sagonas, K. Instruction merging and specialization in the SICStus Prolog virtual machine. In Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, Florence, Italy, 5–7 September 2001; pp. 49–60.
82. Pedersen, J.B.; Kauke, B. Resumable Java Bytecode—Process Mobility for the JVM. In *Communicating Process Architectures*; IOS Press: Amsterdam, The Netherlands, 2009; pp. 159–172.
83. Ring Team. Ring Programming Language Website. Available online: <http://ring-lang.github.io> (accessed on 13 October 2024).
84. Downloads Statistics from the Ring Programming Language Mirror Hosted on SourceForge. Available online: <https://sourceforge.net/projects/ring-lang/files/stats/timeline?dates=2016-01-21+to+2024-10-01> (accessed on 11 November 2024).
85. Alohal, Y.A.; Fayed, M.S.; Mesallam, T.; Abdelsamad, Y.; Almuhawwas, F.; Hagr, A. A Machine Learning Model to Predict Citation Counts of Scientific Papers in Otolaryngology Field. *BioMed Res. Int.* **2022**, *2022*, 2239152. [\[CrossRef\]](#)
86. Fayed, M.S. Classification of the chess endgame problem using logistic regression, decision trees, and neural networks. *arXiv* **2021**, arXiv:2111.05976.
87. Ayouni, M. *Beginning Ring programming*; Apress: New York, NY, USA, 2020; Volume 978.
88. Steam platform. GoldMagic800 Game. 2018. Available online: https://store.steampowered.com/app/939200/Gold_Magic_800/ (accessed on 11 October 2024).
89. Alohal, Y.A.; Fayed, M.S.; Abdelsamad, Y.; Almuhawwas, F.; Alahmadi, A.; Mesallam, T.; Hagr, A. Machine Learning and Cochlear Implantation: Predicting the Post-Operative Electrode Impedances. *Electronics* **2023**, *12*, 2720. [\[CrossRef\]](#)
90. Ghanem, M. *Developing Poet Software Using Ring Language*; MetaBook: Mansoura, Egypt, 2021; ISBN 978-977-6928-38-1. Available online: <https://ring-lang.github.io/ref/GhanemBookUsingRing.pdf> (accessed on 11 October 2024). (In Arabic)

91. Hassouna, A. Over 500 Videos to Learn the Ring Language. Available online: <https://www.youtube.com/playlist?list=PLHIfW1KZRIfI6KzfLziFl650MmThnQ0jT> (accessed on 11 October 2024).
92. Su, H.; Jiang, S.; Lai, Y.; Wu, H.; Shi, B.; Liu, C.; Liu, Q.; Yu, T. ARKS: Active Retrieval in Knowledge Soup for Code Generation. *arXiv* **2024**, arXiv:2402.12317.
93. Green, T.R.G.; Petre, M. Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *J. Vis. Lang. Comput.* **1996**, *7*, 131–174. [[CrossRef](#)]
94. Green, T.; Blackwell, A. Cognitive dimensions of information artefacts: A tutorial. In Proceedings of the BCS HCI Conference, Sheffield, UK, 1–4 August 1998; Volume 98, pp. 1–75.
95. Green, T.R.; Blandford, A.E.; Church, L.; Roast, C.R.; Clarke, S. Cognitive dimensions: Achievements, new directions, and open questions. *J. Vis. Lang. Comput.* **2006**, *17*, 328–365. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.