

Emerging Programming Languages: A Comprehensive Analysis of Carbon, Mojo, and Swift

Daniel Garza and Kevin Sykes

University of Northern Colorado

CS 395

Professor Harris

February 2, 2024

Abstract

The ever-evolving landscape of programming languages presents developers with a myriad of choices, each boasting unique features and strengths. This research paper takes a deeper look at three upcoming programming languages in the tech space: Carbon, Mojo, and Swift. Our study encompasses features and paradigms of each language, the purpose of each language, and what other languages each one competes with. Through this exploration, we aim to contribute valuable insights to the ongoing dialogue on programming language evolution, adoption trends, and the changing needs of developers in the tech industry. A brief history of the evolution of programming languages will also be included in the paper to give readers a general understanding and sense of the direction that programming languages have been taking in recent years.

The Carbon programming language is a language that is designed to be a successor to C++. It is designed to support performance-critical software and has many features that improve on existing C++ ones. Mojo is a groundbreaking programming language, emerging as a transformative tool in the landscape of software development, offering a blend between efficiency and safety without compromise. It marries the flexibility of dynamic typing languages with the safety and robustness of static typing languages, this allows them to facilitate rapid development cycles. It includes many features such as type inference, pattern matching, as well as built-in support for parallel processing and vectorization, which is crucial for emerging AI and ML research and programming. The Swift programming language is a language used primarily by Apple to create software for their devices. It was created as a replacement for Objective-C which was becoming obsolete as it lacked many modern features of new programming languages. Swift brings a wide array of modern features to the language that a language like Objective-C heavily lacked. Swift is a unique language since it does not directly compete with any modern languages since it is designed almost exclusively for Apple devices.

Introduction

Within the intricate web of technological evolution, the programming and computer landscape stands as a testament to perpetual innovation and intellectual inquiry. As we begin an exploration of emerging languages-Carbon, Mojo, and Swift-it becomes imperative to situate our analysis within the historical context that underpins the evolution of programming languages. From the rudimentary binary systems involving manual switches and cables to the epoch of punched cards, each historical moment contributes to the dynamic timeline of programming language development. By contextualizing these emerging languages with the broader historical framework, we plan to unveil the curtain on the intricate interplay between the past and present in the domain of software development. When we gain insights into the historical development of programming languages, we become equipped to discern how these new languages, such as Carbon, Mojo, and Swift, strategically position themselves to build upon the legacy of their predecessors, addressing challenges and innovating within the ever-changing landscape of software development.

Brief History of Programming Languages and Innovations

Coding and programming languages are the pillars of our modern digital world. While most of the significant developments in the tech industry have happened within living memory, the history of programming and computer logic dates to the 1800s. One of the earliest examples of a computer, a machine capable of computations, was in 1822 with Charles Babbage's difference engine. The purpose of the difference engine was to automate the process of performing mathematical calculations and to print numerical tables by the method of finite differences. The machine consisted only of gears and other intricate mechanisms that powered the machine without the need for electricity. Thus, it can be said that the first programming language was physical motion, something that was very manual. "Eventually, physical motion was replaced by electrical signals when the US government built the ENIAC in 1942," (*A History of Computer Programming Languages*, n.d.) The ENIAC was a U.S. government funded project during WWII to build an electronic computer that could be programmed to compute the values of artillery range tables, a typically cumbersome math calculation. It followed many of the same principles of Babbage's engine and hence, could only be "programmed" by presetting the switches and rewiring the entire system for each new "program" or calculation. This process

proved to be very tedious and the need for responsive, more automated programs became ever more apparent. In 1945, John Von Neumann developed two important concepts that directly affected the path of computer programming languages. The first was known as the “shared-program technique” and the second was “conditional control transfer,” (*A History of Computer Programming Languages*, n.d.). The “shared-program technique” stated that the computer hardware should be simple while more complex instructions or code should be used to control the hardware. This would allow the computer or machine to be reprogrammed much faster. The “conditional control transfer” concept would give rise to the notion of subroutines, or small blocks of code that could be executed in any order, instead of a single set of chronologically ordered steps. This means that the computer code should be able to branch based on logical statements such as IF, THEN, and looped over with a FOR statement. “Conditional control transfer” gave rise to the idea of “libraries”, which are blocks of code that can be reused over and over,’ (*A History of Computer Programming Languages*, n.d.). In 1949, a few years after Von Neumann’s work, the language Short Code appeared. It was the first computer language for electronic devices, and it required the programmer to change its statements into 0’s and 1’s by hand. Then in 1951, Grace Hopper wrote the first compiler, A-0. A compiler is a program that turns the language’s statements into 0’s and 1’s for the computer to understand. This led to much faster programming, as the programmer no longer had to do the work by hand and could use higher-level languages that were closer to human-readable English than machine code. In 1957, the language FORTRAN appeared. The language was designed at IBM for scientific computing, and it provided the programmer with low-level access to the computer hardware. This language only included IF, DO, and GOTO statements, but at the time, these were a big step forward for expressing logical statements in code. In 1959, COBOL was developed to address the shortcomings of FORTRAN in handling input and output for business computing. COBOL introduced the concept of grouping data into arrays and records for better organization. Notably, COBOL programs are structured with sections like an essay, and their English-like grammar makes them easily accessible to business professionals (*A History of Computer Programming Languages*, n.d.). C was developed in 1972 by Dennis Ritchie while working at Bell Labs in New Jersey. C uses pointers extensively and was built to be fast and powerful at the expense of being hard to read. Ritchie developed C for the new Unix system being created at the same time. Unix gives C advanced features such as dynamic variables, multitasking, interrupt handling,

forking, and strong, low-level, input-output. Because of this, C is commonly used to program operating systems like Unix, Windows, MacOS, and Linux (*A History of Computer Programming Languages*, n.d.). In the late 1970's and early 1980's, a new programming paradigm was brewing. It was known as Object Oriented Programming, or OOP. Object Oriented Programming would find its way into the C language through extensions and would become known as "C With Classes." This set of extensions developed into one of the most popular languages, C++, which was released in 1983. Heading into the 1990s, interactive TV was the exciting new technology at the time. Sun Microsystems decided that interactive TV needed a special, portable language and that language eventually became Java. Java had its fair share of optimization issues, but Java excelled in code portability and garbage collection.

As long as technology continues to innovate and evolve, new programming languages will continue to give rise. Programming first got its start with a list of steps to follow to make a computer perform a task, even if that was by physical movements. These steps eventually found their way into software and began to acquire new and better features to express algorithms. Modern programming languages differ from old languages in that they allow the programmer to express their thoughts more intuitively and creatively. Modern languages also frequently aim to minimize the friction that comes with learning a new language by using succinct, but powerful syntax. The first major languages were characterized by the fact that they were initially intended for only one purpose, while the languages of today are differentiated by the way they are programmed, as they can be used for nearly any purpose. The programming languages that are Carbon, Mojo, and Swift aim to significantly improve upon the most popular programming languages of the last decades.

Carbon Language

Carbon is a programming language created by Google in 2022 with the purpose of being a successor language to C++. It was first introduced at the 2022 CppNorth software conference as an open-source project with the intent of making it easier for development teams to manage complex C++ systems.

Purpose

To grasp the purpose that Carbon serves in the programming language market, it is crucial to first understand the current state of the language it aspires to largely overtake—C++.

“C++ remains the dominant programming language for performance-critical software, with massive and growing codebases and investments,” (*Carbon-Language*, n.d.). However, with millions of software developers using C++, it is difficult to meet their needs. This is largely in part due to the accumulating decades of technical debt. Google has acknowledged the formidable challenge of incrementally improving C++, recognizing that directly inheriting the legacy of C or C++ would not be conducive to unleashing the full potential of Carbon. In addition to the accrual of tech debt in C++, the decision to inherit the legacy of C was a quick solution rather than a time-consuming one. C++ inherited features from C such as textual preprocessing and inclusion, which helped C++’s success initially by giving it instant access to a large C ecosystem but with significant tech debt ranging from integer promotion rules to complex syntax. (*Carbon-Language*, n.d.). C++ has also prioritized backward compatibility including both syntax and application binary interface. This is mainly motivated by preserving its access to existing C and C++ ecosystems. However, when this approach is taken, features typically get added very frequently without any regard to cleaning up in the form of replacing or removing outdated legacy features (*Carbon-Language*, n.d.). Another reason C++ is extremely difficult to improve is because of its current evolution philosophy and direction. “A key example of this is the committee’s struggle to converge on a clear set of high-level and long-term goals and priorities aligned with ours. When pushed to address the technical debt caused by not breaking the ABI, C++’s process did not reach any definite conclusion,” (*Carbon-Language*, n.d.). This failed to meaningfully change C++’s direction and showed how this process can fail to make directional decisions. Carbon, on the other hand, has a more accessible and efficient evolution process built on open-source principles, processes, and tools. Even with Carbon still being an experimental project, Google has explicitly laid out its goals and priorities of the Carbon language and how they directly impact the decisions they will make along the way. Google plans to build a holistic collection of tools that provide a rich developer experience when using Carbon. Consequently, in the development of Carbon, Google opted to build upon robust language foundations, incorporating features like modern generics. This strategic approach positions Carbon to thrive for decades to come, mitigating technical problems and ensuring a solid foundation for future advancements.


Features/Paradigms

Carbon is strategically crafted with the aim of enticing a significant number of C++ developers to transition to this new language. The key to ensuring a successful adoption lies in Carbon's commitment to supporting nearly all C++ features while simultaneously elevating the most critical aspects of a programming language. Safety and more specifically memory safety, remains a key challenge for C++ and it would need to be something that a successor language approaches very intentionally (*Carbon-Language*, n.d.). The Carbon language would first aim at the low hanging fruit in the memory space that C++ does not capitalize on. Carbon language tracks uninitialized states better and requires increased enforcement of initialization. With this approach, it would provide an extra layer of protection against initialization bugs. Carbon also aims to design fundamental APIs and idioms to support dynamic bounds checks in debug and hardened builds. This would allow developers to test their code in more real time to make sure their code is accessing valid memory segments. Carbon also aims to create a default debug build mode that is both cheaper and more comprehensive than existing C++ build modes even when combined with Address Sanitizer (*Carbon-Language*, n.d.). Address Sanitizer is a memory error detector for C/C++, and it finds errors like: Use after free (dangling pointer dereference), buffer overflows, use after return, use after scope, initialization order bugs, and memory leaks. Another key feature that Carbon supports is a modern generics system with checked definitions, while still supporting opt-in templates for seamless C++ interoperability. Checked generics provide numerous advantages as compared to C++ templates. Generics in Carbon are fully type-checked, which removes the need to instantiate your classes to check for errors. This would also avoid the compile-time cost of re-checking the definition of every instantiation. Additionally, when you are using a definition-checked generic, usage error messages are much clearer, and they directly show you which requirements are not met since it is type-checked (*Carbon-Language*, n.d.). Generics in Carbon also enable automatic, opt-in type erasure and dynamic dispatch without a separate implementation. Type erasure is when object types are hidden at runtime and are replaced with a common base type, which can be helpful for creating more generic or flexible code. Dynamic dispatch refers to the ability to determine, at runtime, which version of a function to execute based on the actual type of that object. The Carbon language also has strong, checked interfaces which result in fewer accidental usages on implementation details and a clearer contract for developers using that code (*Carbon-Language*, n.d.).

Competition

Carbon enters a competitive landscape where established languages, particularly C++, hold a significant presence. The decision to directly compete with C++ while also facilitating collaboration demonstrates Carbon's ambition to carve its mark in a domain that is dominated by mature languages. The challenge lies not only in attracting new developers, but also accommodating those familiar with C++ by offering a smooth transition. Additionally, Carbon competes with other modern languages that aim to address similar concerns of efficiency, safety, and ease of use. By defining itself as a successor language to C++, Carbon seeks to strike a balance between innovation and compatibility, acknowledging the importance of a user-friendly transition from older languages to a new and robust programming language.

Mojo Language

Mojo is not just a programming language; it is a revolution in the realm of artificial intelligence (AI) waiting to happen, having only just recently been released to the general public in May of 2023, it was released internally to Modular Inc. in 2022. Modular Inc is the parent company spearheading the creation of Mojo, and with the help of Chris Lattner, they came up with the idea for it. They hoped to create a language (which was not their initial goal) that was innovative in a way that allowed a scalable programming model that could be used to target accelerators on other systems in the ever bridging the gap between research and production in AI (*Modular Docs - Why Mojo* , n.d.).


Purpose

However, Mojo is not just another language aiming for speed and performance, it has been designed with scalability in mind. This is incredibly important for the ever-evolving landscape of AI, where specialized accelerators and cutting-edge systems are constantly emerging. Unlike general-purpose languages, Mojo aims to meet the specific needs of AI development. It does this by offering built-in support for parallel processing, vectorization, and other techniques crucial for AI workloads.

Features/Paradigms

Having many benefits, some of Mojo's more prominent alluring features include the leveraging of technologies of LLVM and MLIR. This allows for efficient code generation across various platforms. Seeing that Mojo draws inspiration from Python, it contains many if not all of Python's context allowing it to be easy for Python developers to transition with ease. Being still in development, it has many goals on its roadmap to improve and add many features; features like a borrow checker (inspired by Rust) allow for enhanced memory safety and metaprogramming capabilities for even greater flexibility. Allowing one to say goodbye to memory leaks and segmentation faults, ensures memory safety without sacrificing performance and keeps the focus on code-logic rather than low-level memory issues. Mojo, like Python, is a dynamically typed programming language, giving it the ability to be incredibly flexible. But it is also statically typed, allowing it the ability to be safer and more predictable through its support. This static typing is built at the core of Mojo, allowing it to detect potential type mismatches before execution, and prevent runtime errors and crashes. It is like building a house with a solid foundation.

Competition

It is important to realize that Mojo does not aim to be a replacement for Python, but rather an improvement or extension of an already amazing language. It does this by its use of CPython to run all existing Python modules, giving a developer complete access to the already existing robust environment of Python. They hope that they will also implement full compatibility with the Python ecosystem, while also offering low-level performance and low-level control, with the added ability to deploy any multitude of subsets of code accelerators (*Modular Docs - Why Mojo* , n.d.). Unlike Python, it has had what many described as a painful transition, and most developers have experienced this first-hand: the switch from Python 2 to Python 3. Mojo hopes that rather than being a rewrite of Python, it will be an entirely new language. Instead, they can embrace the already strong syntax and design efforts of Python.

Swift Language

Swift is a programming language that was developed by Apple in 2014 and was introduced at Apple's Worldwide Developers Conference. Swift was designed as an alternative to Objective-C, Apple's main language of development before Swift.

Purpose

Apple created Swift as a replacement for all languages based on C, including Objective-C, C++, and C. The reason behind creating Swift was Apple's evolving hardware landscape, necessitating a more modern language equipped with advanced features and improved performance. As Apple's hardware continued to advance, Swift emerged to meet the demand for a language that could seamlessly integrate with the evolving ecosystem while providing enhanced capabilities and efficiency. Developers of Objective-C consistently found that the syntax of the language was cumbersome and error-prone. Swift allows for enhanced productivity as it allows for a more expressive form of programming alongside powerful features that many C-based languages lack.

Features/Paradigms

With its inception, Swift not only aimed to replace established C-based languages, but also sought to drastically alter the Apple ecosystem of development in iOS, macOS, watchOS, and tvOS. One of the key pillars of Swift's success lies in its rich set of features, which are very intentionally designed to bring an improved programming experience to developers. One quality-of-life feature that is present in Swift is that the language supports inferred types. This can make the code cleaner and less prone to mistakes. For Swift to be considered a successor to C-based languages, it needs to have made some significant changes to its approach to safety. Swift has automatically managed memory that is implemented using tight, deterministic reference counting rather than a garbage collector. This will result in a lot less overhead and minimal resource usage (Inc, A. n.d.). Some other notable safety checks in place in Swift are variables are always initialized before use, arrays, and integers are checked for overflow, and the enforcement of exclusive access to memory guards against many programming mistakes. The syntax of Swift was created to make it easy to define your intent. Swift can define a variable with the keyword, `var`, or as a constant with, `let`. One very helpful safety feature in Swift is that objects cannot be

nil or null (Inc, A. n.d.). The Swift compiler will not allow you to make a nil object because you will be met with a compile-time error. However, if you do want to use nil objects, Swift has a feature called “optionals” to let the compiler know that you understand the behavior of what you are doing. Some other additional features to note are tuples and multiple return values, generics, functional programming patterns, and advanced flow control with do, guard, defer, and repeat keywords (Inc, A. n.d.). Swift also takes advantage of the high-performance LLVM compiler technology (Inc, A. n.d.). This ensures the code is converted into optimized machine code to get the most out of the hardware. Swift also has Objective-C and C++ interoperability so that your Swift code can live alongside your existing Objective-C or C++ code in the same project. This allows for a smoother transitioning experience and makes Swift easier to adopt.

Competition

Since Swift lives mainly in the Apple ecosystem, it is somewhat difficult to find another language that directly competes with it. Apple seems to have in mind the idea of having one ubiquitous language for all their platforms and because of this, they have created a very well-rounded programming language that is suitable for a wide array of tasks or projects. However, Swift does compete strongly with a language like Objective-C. This is largely because Objective-C was the main language of Apple development before Swift. Swift was designed to replace Objective-C; however, the two languages continue to coexist as developers often need to choose between them based on project requirements. Swift also indirectly competes with Java and Kotlin in the mobile app development space. While Swift is specific to Apple platforms, Java and Kotlin are used for Android development. Cross-platform frameworks like Flutter and React Native also play an interesting role in the competition.

Conclusion

Our exploration into the emerging programming languages of Carbon, Mojo, and Swift has revealed a vision of innovation and purpose. Carbon, with its ambition to succeed C++, presents a forward-looking approach that acknowledges the challenges of technical debt and endeavors to redefine the space of performance tuned programming languages. Mojo, positioned as an alternative for Python with a focus on Artificial Intelligence and Machine Learning workflows, offers promises of speed and multi-system support. Swift, at the forefront of Apple's development ecosystem, continues to be a driving force for efficient and expressive application

development. As we dove into the intricate features and paradigms of these languages, it became evident that each has carved its niche in response to the evolving needs of the tech industry. The departure from traditional approaches, as seen in Carbon, represents a bold stride towards progress, while Mojo seeks to redefine established norms in the realm of Artificial Intelligence and Machine Learning. Swift, as the main language for Apple devices, remains integral to the developer toolkit, as it embodies a beautiful balance of performance and developer-friendly features. As technology evolves, the choices that developers and creators make with these languages could play a pivotal role in the future of programming.

References

A History of Computer Programming Languages. (n.d.). Retrieved February 1, 2024, from


https://cs.brown.edu/~adf/programming_languages.html

Carbon-Language. (n.d.). *Carbon-language/carbon-lang*: Carbon Language's main repository.

GitHub. <https://github.com/carbon-language/carbon-lang>

Inc, A. (n.d.). *Swift—Apple Developer*. Retrieved February 2, 2024, from

<https://developer.apple.com/swift/>

Modular Docs—Why Mojo . (n.d.). Retrieved February 1, 2024, from

<https://docs.modular.com/mojo/why-mojo.html>