CSci 365: Organization of Programming Languages

Report 3

Max Forst & Kenneth Jahnke

11/26/2024

**Turing**

Turing was developed in 1982 at the University of Toronto by Ric Holt and James Cordy. It was named after Alan Turing, a British pioneer of modern computer science, who you may have heard about from the movie, “The Imitation Game.” It is a high-level programming language, aimed to be easy to learn and to provide an educational language for young computer scientists to learn. For that reason, it was popular in many high schools and universities in Canada. It was implemented to be a relatively portable language, being able to be run on a variety of different machines, like IBM PCs, Apple Computers, and UNIX systems. Turing was inspired by languages such as Pascal, Euclid, and SP/k, taking things like machine independence and clean syntax and adopting them as its own.

Over the years, Turing has gained more variants, notably Object-Oriented Turing, and Turing+. Actually, “Object-Oriented Turing” was renamed “Turing” in 2001, and the original was renamed “Classic Turing”. For the purposes here though, we will only focus on the Classic Turing language. Previously a commercial programming language developed for Holt Software Associates, Turing became freeware and available to download for any personal, commercial, or educational use in 2007. Incidentally, Holt Software Associates also decided to stop supporting Turing in 2007, and it has seen no further development since then. This is likely due to the fact that other more modern and more versatile languages, like Python, were rising in popularity and overshadowed languages such as Turing.

As mentioned earlier, Turing is a descendant of languages such as Pascal, Euclid, and SP/k, so let’s compare their features and see what Turing did differently. Compared to all three of those languages, Turing had a more simplified syntax, making it easier for beginners to understand the code. In addition, Turing added support for directly interacting with graphics, which gave users the ability to engage with visual elements of coding, unlike its predecessors. Some more features that stemmed from its ease of use was how Turing able to run across many different platforms and had a more robust error handling system than its predecessors. However, this simplification of syntax and ease of use had some drawbacks too. These changes meant that Turing didn’t have several low-level operations, such as pointers, which Pascal had, or modular programming, which was available in Euclid and SP/k. So, while Turing did manage to bring some of the best aspects together from other languages, it ultimately fell short because of the lack of the ability to do more structured programming.

Although not directly related, we can also compare Turing to Python, as they served similar roles during the rise of these languages. Doing this we can see why languages like Python ultimately beat out languages like Turing for the top spot in educational and professional programming. Unlike Turing, Python had many extensive libraries, offered by both the Python developers themselves and by the community at large. This is one of the major reasons that Turing was beat out by Python. Turing didn’t really offer libraries that would expand its functionality, and its community members weren’t really all that active in developing any libraries for it either, since right when it became freeware, its developer’s abandoned it. In addition, Turing had static typing while Python had dynamic typing, which made Python an easier language to grasp for new developers. Python could also handle more complex programming, such as object-oriented programming, unlike Classic Turing. While later versions of Turing did offer OOP, Python implemented it in a much better way as the entire language was OOP based. Lastly, Python was a more flexible and adaptable language compared to Turing, meaning that it was much more versatile in all the different types of use cases it could handle. With these clear advantages, it is clear to see why Turing, and other similar languages ultimately met their fate and have since been discontinued in favor of more versatile languages like Python.

Turing’s contribution to programming languages, while not as significant as others, did help several generations of programmers initially get into learning programming. Its simplicity and clean code made it the perfect tool for teaching students the fundamentals of programming. And it had a nice advantage compared to some other languages in that it was cross-platform. Unfortunately, these advantages weren’t enough to outweigh its disadvantages and concerns, like a lack of low-level programming, limited applicability, and small community support. Due to these facts, other more modern languages with all of Turing’s capabilities plus more were able to take over and contribute even more to modern programming practices than Turing ever did.

Unfortunately, Holt Software Associates went out of business and the official website is no longer available, but you can still download the different versions of Turing from the following website.

Website: <http://compsci.ca/holtsoft/>

Sources: <https://dl.acm.org/doi/pdf/10.1145/53580.53581>

<http://compsci.ca/holtsoft/doc/>

<https://research.cs.queensu.ca/home/cordy/pub/downloads/tplus/Turing_Report.pdf>

**F#**

F# was developed by Don Syme and his team at Microsoft Research, Cambridge, UK in the early 2000s. It’s first release was in 2005 for the Windows platform, and later, in 2010, it’s 2.0 version was released on Windows, macOS, and Linux as a part of Visual Studio 2010. It was designed to bring functional programming into the .NET ecosystem, utilizing several different programming methods, like functional, imperative, and object-oriented programming. It is mostly used as a cross-platform Common Language Infrastructure (CLI), which allows multiple high-level languages to be run on different platforms without needing to be rewritten for a specific machine. F# is a member of the ML language family and it was heavily inspired by the OCaml language, also taking inspiration from languages such as C#, Python, and Haskell.

While it was originally developed as a research language, it quickly grew into a fully supported Microsoft product allowing for open-source contributions and cross-platform support. F# is still in wide use today across all sorts of different platforms, with their latest version being 8.0, which came out in 2023. Today it is maintained and developed by the F# Software Foundation, Microsoft, and other open contributors from the community. While the F# Software Foundation provides and open source, cross platform compiler, Microsoft has also made it a fully supported language in Visual Studio.

Because F# is so closely related to OCaml, let’s take a look at how the two differ. Obviously, the most notable difference is how F# is compatible with the .NET ecosystem, which allows it to more easily create applications for the web, mobile, and desktop. This also made F# a better language for OOP than OCaml, as it was able to use .NET’s strong OOP capabilities. F# also has the ability to natively run asynchronous code, unlike OCaml, which does have the option, but you must use a specific library. F# also introduced more complex pattern matching, which allowed for more complex data structures, or even user defined data structures. Other features from OCaml were removed and replaced altogether with a more simplified .NET version to make F# more compatible with the framework. Some of these features include specific syntax from OCaml, classes, and modules.

F# also took inspiration from C#, Python, and Haskell, so let’s take a look at how these languages compare as well. F# utilizes OOP principles, just like C#, however one main difference is in how they are executed. F#’s version of OOP is meant to utilize the .NET framework, often simplifying the process. In addition, F#’s syntax is often more concise and easier to read compared to C#. When compared to Python, it is easy to see some differences, like how F# is statically typed and compiled, while Python is dynamically typed and interoperated. Python also has a wider range of use cases when compared to F#, but one thing that F# made sure to take away from Python was its relatively easy human readable syntax. Haskell utilized strong pattern matching, which allowed for more complex data structures, a feature that F# also made sure to adopt. In addition, F# also extended this pattern matching to user defined data structures.

Overall, F# has made a pretty big contribution to programming languages overall. It made functional programming more accessible by embedding itself in the .NET framework and being able to be used as a CLI has also contributed to its success. It has several advantages like being able to support different types of paradigms, like functional, imperative, and object-oriented programming. It also utilizes several nice features from other languages, like conciseness and a simple syntax, and strong pattern matching. There are however, some drawbacks. One of these concerns is how, despite its syntax being simple, users might find its OOP paradigm more complex than other well established OOP languages. Another drawback is how F# isn’t as established as other popular languages that essentially do the same thing, like C# or Java.

Website: <https://fsharp.org/>

Sources: <https://learn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp>

<https://fsharp.org/history/hopl-final/hopl-fsharp.pdf>

<https://adabeat.com/fp/introduction-to-f-sharp/>

**Go**

Go or (Golang) was created under everyone's favorite cyber technocracy, Google. It has seen use by many large players like Meta, Microsoft, Netflix, PayPal, Twitch, American Express, and more. G1

BACKGROUND

In 2007, Robert Pike (co-creator of UTF-8) was dissatisfied with the slow speed and tediousness of C++ when working on a Google program. The idea came to him to create a new language to resolve these issues. It would be C-like in nature, but with an injection of personal preferences and without the speed and tedious issues. G2

Pike developed Go alongside Robert Griesemer and Ken Thompson (of UNIX and B language fame). The first public release of Go was in 2009. G3 Version 1.0 was released in 2012. The most recent version, 1.23, was released in August 2024. G4

FEATURES AND GOALS

Go was developed to provide a language that large teams could utilize to create sizable programs with efficiency, ease of use, and safety. Some features:

* Simplicity in syntax. Whereas in C, semicolons must be explicitly typed to declare the end of statement, Go infers their presence. This creates efficiency in development and decreases the likelihood of errors. Another example of Go’s goal of efficiency is its `+=` operator, which is a concise increment operator compared to the verbose syntax required to perform this function in other languages. G5 A notable "keeping of tradition" from C to Go was the reliance on curly braces to wrap code statements, as the developers didn't like the vagueness of other languages like Python to know where functionality starts and ends. G2
* Concurrency. Go’s standard libraries offer built-in support for distributed systems, multi-thread tasks, web services, and high-performance networking. G5
* Default to zero. Go initializes all variables to zero by default, which reduces the likelihood of memory mismanagement errors. G5
* Garbage collection. Go has built-in garbage collection, which reduces the likelihood of errors and streamlines development because there isn’t the need to explicitly allocate/deallocate memory. G5

COMPARISON TO OTHER LANGUAGES

The main languages of comparison against Go are C and C++. Go is generally more development-intuitive with its sleights of hand, such as implicit semicolons. While these small factors may not play a huge impact on the logic, it does speed development by reducing the likelihood of compilation errors and by simply reducing the number of keystrokes. While C and C++ perform very well on embedded systems due to their low-level nature, they are simply do not have the lean-in and emphasis on safety that Go has when developing more modern systems such as network interfaces and distributed solutions need to run successfully. The manual memory management required in C/C++ creates security vulnerabilities that Go doesn't have. The syntax is generally simpler in Go. G5

While both Go and Python are memory-friendly due to automatic garbage collection and have simple, easy-to-learn syntax, Go has the edge in that it is compiled which makes it quicker at run-time, where Python is interpreted. G7

Go is comparable to Rust in that both prioritize memory safety. While Rust is generally quicker than Go due to its lack of garbage collection, it is thought to not be as easy to learn. G7

Website: <https://go.dev/>

Sources:

G1. <https://go.dev/solutions/case-studies>

G2. <https://www.britannica.com/technology/Go-programming-language>

G3. <https://go.dev/solutions/google/>

G4. <https://go.dev/doc/devel/release>

G5. <https://www.halfnine.com/blog/post/go-vs-c>

G6. <https://www.tftus.com/blog/golang-vs-other-languages-a-comparative-analysis>

**Rust**

Rust started out as a side-project for a developer at Mozilla. It has since grown into a robust language with a promising future because of its emphasis on efficiency, practicality, safety, and concurrency. R1 There were estimated to be 2.8 million developers writing code in Rust in 2023. R2

BACKGROUND

The development of Rust began in 2006 by Graydon Hoare, a computer programmer who was working for Mozilla at the time. He knew that much of the "backbone software" we interact with in daily life (such as those for elevator systems, a fact he knew from personal experience) were written in languages like C and C++. Thus, he knew how the languages within those systems were prone to accidental memory mismanagement leading to a crash in the system. Rather than just complaining about the problem (and taking the stairs), he began work on a language that would still have the ease of development of C++ but with care and emphasis on memory safety and reliability. R2 R3

Mozilla began sponsoring the Rust project in 2009, when the language became mature enough to demonstrate its core functionality and pass basic testing. R4 They announced the project to the public in 2010. The first stable release of Rust 1.0 was released in 2015. A new "edition" of Rust is released roughly every 3 years. R5 Rust 1.31 (Rust 2018) was released in 2018. R6 Rust 1.56 (Rust 2021) was released in 2021, the same year the Rust Foundation was founded and took over the project. R7 R8 As of the time of this writing Rust 2024 is scheduled to be released in early 2025. R9

FEATURES AND GOALS

Rust has three fundamental goals: concurrency, speed, and safety. Its implementation of these goals is present in several features:

* Concept of ownership. In Rust, each value is owned by one and only one variable. How it works: Suppose `functionA` stores a value at `varX`, then calls `functionB` with `varX` as an argument. The value at `varX` within `functionA` ceases to exist. Instead, ownership of that value is stored solely within `functionB`. R10
* Lack of garbage collection. Because of the concept of ownership within Rust, there is no need for any sort of garbage collection as there are not multiple copies of related values at various places within a program to be purged when they are no longer needed. This increases program speed performance because there is no need to pause to perform cleanup. Rust developers removed garbage collection features in 2013. R2 R10
* Increased memory safety. References within Rust are considered safe, meaning the compiler will check for errors that contradict the concept of ownership and not allow a program to compile if safe handing off or borrowing of values cannot be performed. This, along with discouragement of the use of pointers, vastly limits the likelihood of memory-related errors. R10
* Improved concurrency. Because of the streamlined and low-risk nature of the ownership-style data flow, Rust is an ideal match for hardware with multi-thread processors. R11
* Cross-platform. The Rust development emphasis on cross-platform compatibility increases the language's potency as it enables streamlined, low-risk development of a wide range solutions on various platforms. R11

COMPARISON TO OTHER LANGUAGES

Rust was designed primarily to fix the shortcoming of C++. Although C++ allows for abstractions and is performant due to its low-level nature, its method of memory management creates ripe opportunities for bugs, such as dangling pointers. R1 R2

Rust was prototyped in OCaml before bootstrapping itself. R12 Rust shares a few of the functional programming features of OCaml, Haskell, and F#. Its data types were inspired by Lisp, Limbo, C, SML, and OCaml. R5

Rust has advantages over a number of other languages. Its lack of garbage collection makes it quicker than languages like Python or Java. Compared to C and C++, Rust is less of headache because there's no need for manual memory management, and less prone to errors like memory leakage.

Website: <https://www.rust-lang.org/>

Sources:

R1. <https://prev.rust-lang.org/en-US/faq.html>

R2. <https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/>

R3. <https://www.rust-lang.org/>

R4. <https://prev.rust-lang.org/en-US/faq.html>

R5. <https://learning-rust.github.io/docs/why-rust/>

R6. <https://doc.rust-lang.org/edition-guide/rust-2018/index.html>

R7. <https://foundation.rust-lang.org/news/2021-02-08-hello-world/>

R8. <https://doc.rust-lang.org/edition-guide/rust-2021/index.html>

R9. <https://doc.rust-lang.org/edition-guide/rust-2024/index.html>

R10. <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html?highlight=ownership#what-is-ownership>

R11. <https://travis.media/blog/why-rust/>

R12. <https://users.rust-lang.org/t/understanding-how-the-rust-compiler-is-built/87237>

**Kotlin**

Kotlin is a modern programming language developed by JetBrains that runs on the Java Virtual Machine (JVM) and is often used to develop apps for Android devices. It started development in 2010 and was announced to the public in 2011, but it wasn’t until 2016 when Kotlin was fully released to the public in its first stable version. Its team leader, Dmitry Jemerov, said he wanted to create the language because most other languages did not have the features that he was looking for.

Kotlin is a compiled code that is unique in that when compiled, it can be transformed into several different types of code, such as JavaScript, or other native machine code. It does this by first compiling to bytecode and then using the JVM to either interpret the bytecode or compile it into native machine code. Supported native machine code includes platforms like iOS, macOS, Windows, Linux, and several other embedded machines. Kotlin was meant to be a very strong OOP language that was better than Java, but still interoperable with it so that people could slowly migrate from Java to Kotlin.

As Kotlin was meant as a replacement for Java, let’s compare the two. One new feature that Kotlin developed was called Null Safety, which allowed for code to easily bypass values that were null and replace it with something else, instead of manually having to check if the value was null with an IF statement. As an example, the following line of code will set the variable ‘pet’ to ‘animal’ only if ‘animal’ is not null, in which case it will set ‘pet’ to “cat”: val pet: String = animal ?: "cat". Another improvement over Java includes how much more concise and clean the code was. Java is known for being very ‘wordy,’ meaning that it often uses a lot of words across multiple lines of code to do a simple task. Kotlin improved this and made it so the same task could be accomplished with very few words or lines of code. One more thing that Kotlin added was a feature called extension functions, which are functions that are able to be added to classes without altering the original code for that class. This is especially useful if you are using someone else’s code and need to just add one extra little feature for convenance.

Another similar language that one might compare to Kotlin is Swift. While Kotlin is the official code used for developing apps for Android devices, Swift is the official code used to develop apps for iOS devices. The biggest difference between them though, is how Kotlin is multiplatform and can compile code to run across all sorts of different devices, Swift is only meant for Apple devices.

Kotlin has made a pretty big contribution to JVM programming. It addresses downfalls of Java while also being able to directly run Java code. It is also so impactful because of how it is used to develop apps that many of us use every day on our smartphones. It also has the advantage of being a relatively new language, meaning that it has many new modern features that other languages might lack. There are some drawbacks though, like how it is generally slower than basic Java because it is more complex, as well as how compiling to different machine codes will take longer that directly compiling for those machines in the first place. Overall though, Kotlin is a pretty neat language.

Website: <https://kotlinlang.org/>

Sources: <https://kotlinlang.org/docs/home.html>

<https://www.geeksforgeeks.org/kotlin-programming-language/>