The Identity of Programming Languages

Jaran Chao, *BS, Comp. Sci.*

***Abstract*—Programming languages write every piece of software imaginable. However, the theory surrounding them is cloudy and vague. The following project explains in-depth the design choices of creating a programming language. The main purpose of this project is to research what gives at programming language it's identity. This research helped influence the design choices of the toy programming language created for this project.**

I. Introduction

THERE are many programming languages, each with its programming paradigms, styles, and features. Many factors contribute to giving a language its identity. Differences in grammar, abstraction, and ideology make each language unique. However, subtle differences in how the language is implemented also affect its identity. This work will tackle these subtle differences and their effects on language identity.

II. Compiled versus Interpreted

THE first major distinction between different languages is if it is compiled or interpreted. Each language has its own way of executing source code and almost every language uses a compiler in some form. Some languages such as Python, Ruby, and Lua will compile to an internal bytecode and then run an interpreter on that bytecode. These languages are considered interpreted. Other languages such as C/C++, Rust, and Go will compile ahead-of-time to a native binary that the OS and CPU can understand and execute. And other languages such as Java, C#, and Elixir compile ahead-of-time to a bytecode, which is then interpreted on a language VM. Both of these types of languages are considered compiled however these distinctions can get very blurry and cause confusion.

Let us first define what the terms "compiled" and "interpreted" mean. These definitions are taken directly from *Crafting interpreters*[1] by Nystrom as they fit our needs perfectly.

- When we say a language implementation "is compiled" or "is a compiler", we mean it translates source code to some other form but does not execute it. The user has to take the resulting output and run it themselves.
- When we say an implementation "is interpreted" or "is an interpreter", we mean it takes in source code and executes it immediately. It runs programs "from source".

For some languages, this is purely black and white. GCC and Clang for C/C++, rustc for Rust, and gc for Go will compile their respective languages to machine code. The executable is then ran directly by the end user, who may not even know which tool or even language was used to compile it. Therefore, these are compilers

For others such as PHP3 and MRI (Matz's Ruby Interpreter), the user would run the program directly from the source code. The underlying implementations would parse it into a syntax tree and immediately execute it by traversing said syntax tree. No other intermediate steps occurred, internally or externally. So, these are most definitely interpreters.

However, more recent languages have landed themselves in more of a grey area. Utilizing aspects of both compilers and interpreters. CPython, for example, is a language that is considered to be interpreted from a user's perspective, however a peek under the hood shows the truth. CPython is both interpreted, but also has a compiler. When running a Python program, the source code is parsed, then compiled into Python's internal bytecode format, and finally the generated bytecode is then interpreted by the Python Virtual Machine (PVM). Java is another example of this grey area. Java is a language that is considered to be compiled from a user's perspective. While this is true, the Java compiler, javac, compiles Java source code ahead-of-time into Java bytecode. This bytecode is then stored in .class files which is then interpreted by the Java Virtual Machine (JVM).

In practice, most modern languages utilize both compilers and interpreters in the build process as you can see below:

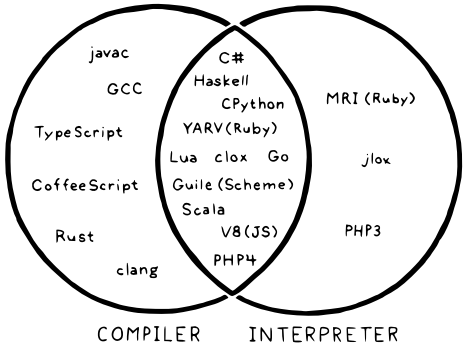

Fig. 1. Venn diagram of compiled languages vs interpreted languages[1]

III. Compilation Target

NOW that we understand the differences between compiled and interpreted languages, we can better understand each of their capabilities. Each have their own advantages and disadvantages.

A. Compiling to Native Binaries

Many languages compile to native binaries. C/C++, Rust, and Go are common examples of languages that compile to native binaries. An advantage of native binaries is that they require little to no overhead. Unlike some of the other aforementioned languages, the native binary is able to be executed directly without having to spin up a VM. Because the native binary consists of machine code that is understood

by the underlying OS and CPU, no additional software or translation units are needed. This specificity allows for a compiler such as GCC or rustc to make specific optimizations for specific architectures.

This level of precision, however, is a double-edged sword. While it allows for compilers to make very specific optimizations, it also means for a program to run on another platform with a different CPU or OS that program must be recompiled for that specific set platform. While this may not be too much of a hassle for end users as it is just a couple of CLI commands. For a language creator, this does not scale well as the compiler would need to be rewritten every time the language decides to support a new architecture. Though, this problem can be circumvented by choosing a target language to compile to. Then, instead of writing and maintaining the compiler backend for multiple different architectures, you can utilize the existing compilation tools of the target language as an optimizer and code generator. Most common targets are LLVM IR[2] and C.

In spite of simplifying the compiler for a language, this now also requires that the target platform has the ability to run the target language, introducing a huge dependency into your language. It is for this reason that the toy programming language did not choose to compile to a native binary as a substantial dependency was not ideal and rewriting the code generation process for multiple architectures was also heavily undesirable.

B. Tree Walk Interpreters

In more recent years, most interpreted programming languages have strayed away from direct traversal of the source code's syntax tree. The main reason is that traversing trees is very slow and very resource intensive. Traversing the syntax tree and executing the code at every node are very costly operations. As well as storing the tree and all the data required at every node takes a lot of memory, especially for larger programs that can have thousands of lines of active code.

However, the biggest advantage of tree walk interpreters is their simplicity. For student projects and embedded and domain specific languages, this implementation is very common as the speed and memory constraints become less of a concern as the programs will be smaller and simpler. The speed of this implementation was the reason the toy programming language did not choose to utilize a tree walk interpreter.

C. Compiling to IR

Many languages compile to a intermediate representation (IR), usually in the form of language specific bytecode, before running an interpreter. Both interpreted and compiled aforementioned languages alike. Each have their own minor differences, but both types work in almost identical fashion. Every language will first parse and compile the source code into some IR format. This IR is then interpreted by a language VM. However, the key difference is when the IR is interpreted. For languages like Python, Lua and Ruby, the IR is interpreted on their respective language VMs immediately after the compilation step is completed which is why these languages are considered interpreted. However, languages like Java and C# compile to an IR ahead-of-time and then this IR can be executed by the language VM later which is why these languages are considered compiled.

The biggest advantage of this approach is that optimizations are able to be performed on the IR such as dead code elimination, loop and data optimizations, and tail call optimization just to name a few. These optimizations can greatly increase the speed of the program once it is passed to the interpreter. This approach is also scalable as it allows for the additions of new platforms as the only requirement is to rewrite the language VM which is not as complex as the whole language itself. It was for these reasons that the toy programming language chose to utilize compiling to an IR and having a language VM.

IV. Type System

THE next major distinction is the type system of a language. The type system is the main basis for the rest of the features of a language. Type systems often take one of two forms: static or dynamic. This is a highly contested topic and different sources will cite different definitions for these terms. For this work, these terms will be defined as the following.

- When we say a language is "statically typed", we mean that the value of the variable or function will be known ahead-of-time. This usually means compile time and therefore most statically typed languages will have an intermediary step which will perform some sort of compilation.
- When we say a language is "dynamically typed", we mean that the value of the variable or function will be determined by the value stored at runtime.

A. Dynamic Typing

Dynamic typing is most commonly utilized by scripting languages. This speed up the development process as less time is dedicated to knowing the type of the variables and functions in use, instead allowing for the underlying interpreter to determine if the code is utilizing valid types during execution. However, this approach to typing has many drawbacks. According to "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript", almost 15% of all software related errors are type related errors[3]. While this is not conclusive evidence that dynamic typing leads to more error prone code, there is a reason why dynamic languages such as Python[4] and Ruby[5] have started to implement some form of type analysis to ensure type safety.

B. Static Typing

A wide variety of languages utilize static typing and for many different applications. Typescript for web applications, C++ and Rust for performance intensive applications, Kotlin and Swift for mobile applications, and many more. As stated before, static typing can allow for less error-prone code. The ahead-of-time type checking can catch many bugs in development. Type checking at compile time also leads to performance

boosts. Compared to dynamically typed languages, statically typed languages do not need to do type checking at runtime since it has the guarantee that once a program is compiled it is a valid program, at least from the type system's perspective.

This insurance, however, comes with many disadvantages. Conforming to a type system may lead to more verbose programs and increase development time. Telling a compiler the type signature of every variable and function is tedious. However, to combat this, languages such as C++[6], Java[7], Kotlin[8], Scala[9], Swift[10], and more, implement a form of type inference. This gives software creator a more dynamically typed development experience while not sacrificing the advantages of having a static type system.

V. Programming Paradigm

THE final major distinction are the chosen programming paradigms that a language chooses to support. There are vast amount of programming paradigms, too many to cover in this work, each with their own method of structuring programs. However, the two most commonly supported paradigms in programming languages are Object Oriented and Functional. Due to their popularity, the toy programming language has plans to support both these paradigms.

A. Object Oriented Programming

Object Oriented programming consists of objects that store data in fields and the ability to interact with other objects in a program through methods. Many programming languages support object oriented programming. Languages such as C++, Java, Kotlin, Swift, Scala, and Javascript. However, many of these languages are multi-paradigm languages, meaning they support more than one programming paradigm.

B. Functional Programming

Another one of the most commonly supported programming paradigms is functional programming. Functional programming consists of data and functions. In functional programming, functions are considered first class citizens and can be passed around the same way data can be passed. This allows for more complex abstractions giving by other paradigms to be modeled as functions. Languages such as Haskell, Kotlin, Scala, C++, Javascript, and Python give extensive support for this programming paradigm.

VI. Conclusions

THERE are many factors that give a programming language its identity. We have only discussed the three most important characteristics that defines a programming language. This work is only but scratching the surface of the many design choices that can subtly change the identity of a programming language.

References

[1] Robert Nystrom. *Crafting interpreters*. Genever Benning, 2021.

[2] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[3] Zheng Gao, Christian Bird, and Earl T. Barr. "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. May 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

[4] Łukasz Langa ¡lukasz at python.org¿ Guido van Rossum ¡guido at python.org¿ Jukka Lehtosalo ¡jukka.lehtosalo at iki.fi¿. *PEP 484 – type hints*. 2014. URL: https://www.python.org/dev/peps/pep-0484/.

[5] Masataka Pocke Kuwabara. *RBS collection was released!* Sept. 2021. URL: https://dev.to/pocke/rbs-collection-was-released-4nmm.

[6] *Placeholder type specifiers (since C++11)*. URL: https://en.cppreference.com/w/cpp/language/auto.

[7] *Type inference*. URL: https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html.

[8] Marat Akhin and Mikhail Belyaev. *Kotlin Language Specification*. URL: https://kotlinlang.org/spec/introduction.html.

[9] *Scala - Type inference*. URL: https://docs.scala-lang.org/tour/type-inference.html.

[10] Apple Inc. *Swift.org - Type Safety and Type Inference*. URL: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#ID322.