A Survey of the History of Popular Programming Language, and What They Can Teach Us

Athabasca University - Comp 601

January 29, 2017

Jason Bell 3078931

Professor: Dr. Huntrods

Abstract

The programming language graveyards are full of dead languages. How can the creator of a new language avoid adding to this graveyard? The history of programming languages shows that they evolved in fits and starts, and as experiments: some successful, some less so. Each language that became popular introduced some novel feature which was, at the time, exciting and useful. These new features ranged from improved usability, to improved portability, to adaptions to the changing requirements of the time. Study of this history may help reveal the path to creation of a programming language that is widely adopted, rather than being an intellectual exercise.

Keywords: programming language, compiler, history

Introduction

Programming language compilers are the foundation upon which the modern world is built. This may seem like an overstatement, yet any technological innovation of the last several decades has required software, and software requires compilers for their creation.

For my undergraduate major project, I implemented a custom programming language and compiler, called Pie, named both as a nod to Python as inspiration, and in a fit of whimsy. I selected this project when, after experiencing frustration with a Java technology, I asked myself: wouldn’t it be great to use a language of your own making, tailored to your own needs? This idle thought led to researching .NET compiler technologies, .NET being my preferred development platform. Given that these technologies turned out to be of high quality, and simple to use, I decided to select this as my major project.

This paper discusses the history and technologies behind compilers and programming languages, as my graduate project will be a continuation of my undergraduate project. During this project, there will be many choices of technology and methodology to be made. Which choices are made will determine whether Pie becomes an accepted and popular language, or just an intellectual exercise. This paper surveys the history of some of the most widely used programming languages, and the reasons behind their popularity, in the hope that this will help reveal ways in which Pie could be designed in an appealing way.

History

In 1952 Grace Hopper implemented the A-0 System, the first software compiler, for the UNIVAC I [1]. The A series languages supported subroutines and passing of arguments between them [2]. While still a very low-level language, it was high level relative to machine code. Hopper worked with UNIVAC programmers to add an increasing number of English language statements to the programming language. This process occurred iteratively through versions A-1, A-2, A-3, and B-0, also known as FLOW-MATIC [3]. FLOW-MATIC was therefore the effectively the first programming language that was easily human readable. The following years saw an explosion in high level languages: Fortran, Lisp, ALGOL, and COBOL, each drawing obvious inspiration from FLOW-MATIC.

Fortran, produced by IBM in 1954, was the first optimizing compiler: a compiler that inspects the source code for opportunities to minimize the resulting machine code. This was an important step in the acceptance of high level languages, as programmers used to working with machine code were skeptical that high level languages could match its performance [4]. Use of

Fortran has continued uninterrupted to the present day, with the language receiving a series of added features, such as the object-oriented paradigm, as technology changed. Since the language has adapted to changing demands, it is still used for scientific computing on supercomputers [5].

Lisp, invented by John McCarthy at MIT in 1958, was the first language to eliminate the distinction between data and code: a distinction that most languages still make to this day. Both code and data are defined as nested expressions, and are interchangeable. The advantage of this is found in its application to artificial intelligence: because data and code are the same, a program can generate new code to simulate learning [6].

In the same year, ALGOL introduced the concept of code blocks: procedures with declared signatures, and code bound between begin and end statements [7], instead of being an arbitrary point in the code that is jumped to when required. As the language that was the first to introduce this feature, found in succeeding languages, ALGOL continued to be used as the de facto standard for algorithms by the ACM long after its use was replaced by its successors [8].

Until the advent of COBOL, programming languages were used for numerical and scientific calculations, and designed with this requirement in mind. In 1959 a committee of individuals with diverse interests, including Grace Hopper, met and discussed the viability of a general-purpose programming language [9]. The evidence for the success in their efforts is found in the number of businesses and governments that continue to use COBOL to this day. Even if more modern languages provide advantages, they are often insufficient to warrant the expense of replacing legacy COBOL code that works fine [10].

ALGOL led to the advent of BCPL (Basic Combined Programming Language) in the 60s and is noteworthy for the addition of two features found in modern languages. First, it introduced

braced code blocks as an early predecessor for C-style syntax. Secondly, it introduced separation of language parsing and machine code generation: a feature found in all modern compilers. This separation allowed greater portability: rather than having to create an entire new compiler per platform, only the machine code generator had to be rewritten [11]. In 1969 Bell Labs produced B, a more expressive form of BCPL [12] that directly led to the advent of C [13].

C was developed at Bell Labs in 1973 and in many ways, was a step back from the principles that led to COBOL: the C language has fewer English statements and has braced code blocks. Instead, it was designed to be very light weight and close to the hardware. C provides direct access to memory and has statements that map closely to machine code. Therefore, the language provided a means of writing code that accesses hardware without having to resort to machine code [13].

Simula was a series of languages developed in the 1960s which introduced novel features such as the object-oriented paradigm, including distinction between object and class, inheritance, and virtual functions [14].

C++ is an evolution of C, developed by Bjarne Stroustrup starting in 1979. Stroustrup was inspired to do so by his dissatisfaction with existing languages: languages were either high level but too slow, or fast but too low level. He selected C as a starting point because it was already in wide use, was fast, and was portable. In short, as a full discussion is outside of the scope of this paper, Stroustrup sought to incorporate the novel features of Simula into C. He did so at first by adding new features to existing C compilers: a “C with classes”. In 1983 it became a language in its own right when it was renamed “C++” and its own compiler, Cfront, was implemented. Even then, Cfront used C as a bridging language: Cfront converted the C++ code into C, then compiled that [15].

The Python language was first released in 1991 as a general-purpose language incorporating a wide range of language features: it is dynamic type, delineates code blocks with white space, has a garbage collector, and supports multiple programming paradigms. Python brought these features together into one language. Python is also interpreted: rather than loading machine code or an intermediate language from storage, the original source code is loaded and converted to machine code, "on the fly" [16]. The fact that Python is interpreted, combined with the overhead of late-binding caused by dynamic typing, means that there are performance penalties to be paid through use of Python [17][18]. This is offset by a more concise syntax and greater portability.

Sun Microsystems released Java in 1995 and Microsoft released C# in 2000. Because of their similarities in the context of this paper, they are discussed together. Both are high level languages with very similar C-style syntax. Both languages compile to an intermediate byte-code language that runs in a virtual machine. The virtual machine compiles the byte-code to machine code "just-in-time", with the result that ideally, the byte-code produced by the compiler should be able to run on any platform for which a virtual machine has been implemented [19][20].

An interesting development is recent renewed interest in compilers that compile directly to machine code rather than an intermediate byte-code language ("ahead-of-time" rather than "just-in-time"). Examples include various libraries that compile Java into machine code to run on the IOS platform [21]. Similarly, Xamarin uses the LLVM compiler to generate machine code from C# [22][23], and the Unity game engine uses the IL2CPP compiler to convert C# to C++ [24].

Discussion

The evolution of programming languages highlights a few facts. First, it was experimental. The first generation of high level languages (Fortran, ALGOL, COBOL, and Lisp) each differed, and each had their own unique features. Much as biological evolution culls the weak, while two are still in use today (Fortran and COBOL), one was supplanted by new forms (ALGOL), and one is in danger of extinction (Lisp). The selective forces were the changing requirements of the time: while Fortran is still used for scientific computing, COBOL came about due increased interest in business applications.

Secondly, the evolution was at times regressive. Many of the first generation of languages were created with the goal of making them as close to English as possible. Subsequently, languages reverted to forms which were more concise and which mapped more closely to machine code. This catered to the needs of programmers for whom verbose English syntax was a hindrance rather than a benefit.

Third, choice of programming languages is a matter of personal preference. Bjarne Stroustrup implemented C++ because he was unhappy with the languages available at the time: the lack of a language that matched his needs drove him to create his own. This mirrors my own experience that led to the creation of Pie: I enjoy Python, but find it slow, and dislike that both Java and C# force the user to adhere to the object-oriented paradigm.

The history of programming languages also highlights one important fact that must be considered in the creation of a new language: each new language that gained wide acceptance introduced something new. This could be a novel language feature, as with Simula and the object-oriented paradigm, or a compiler feature, as with the separation of parsing and machine

code generation in BCPL. This is the question of greatest concern to my graduate project: what novel features must the language introduce to set it apart from the diverse array of languages already available? Without a good answer to this question, implementation of Pie is pointless, other than as an intellectual exercise.

In surveying the history of programming languages, I have realized one possible avenue that may lead to an answer to this question. Since C became successful by abandoning English language syntax in what, on the surface, seems to be a regression, can the same be done with Pie? What are the reasons for which COBOL, Fortran, and Lisp are still in use today? Which features of these older languages, if any, would be of interest in a new programming language? Perhaps the unification of data and code, without the esoteric syntax of Lisp? While COBOL is largely only extant for legacy reasons, Fortran is still used for scientific computation on supercomputers. Why is this? Detailed investigation of these sorts of questions may lead to discovery of ways to make Pie unique and interesting.

The most recent version of the Pie programming language used C# as a bridging language: the compiler converts Pie to C#, then compiles the C# code using the .NET compiler. This had three advantages. First, it simplified an already very challenging project by removing the need to generate .NET byte code. Second, this takes advantage of optimizations already carried out by the C# compiler. Third, any compiler errors that the Pie compiler does not detect are detected by the C# compiler. For future versions of Pie, I had planned to compile to machine code, either directly or through a byte code language. This survey has suggested that this may not be a necessary step for the language to be accepted. It is yet to be seen whether the increase in the number of compilers using a bridging language is a lasting backlash against the popularity of just-in-time interpreted byte code, but it may be an interesting avenue of research. Perhaps Pie

could use C++ as a bridging language? This would allow Pie to take advantage of the speed and low level access to memory provided by C++, while being a safer and friendlier language to work with. It may also be feasible to embed C++ code into Pie code.

The viability and relative merits of these different options will need to be carefully considered.

Conclusion

Study of the history of successful programming languages is an important part of creation of a new one. What made these languages successful? What can a new language do to duplicate this success?

While this paper is a broad overview of the history of a subset of languages, it suggests that further study of some of those languages will be valuable. Are there reasons, aside from legacy code, that COBOL is still widely used? Why is Fortran still used for scientific computing on supercomputers? Is this a cultural tradition of those programmers, or does Fortran have qualities that make it uniquely suitable?

Further study is required, but it is apparent that in creating a new programming language, looking backwards is as important as looking forwards.

Citations

[1] D. Gtirer, *Women in computing history*. inroads (SIGCSE Bulletin), *34*(2). P 118.

[2] C.W. Adams, S. Gill, and D. Combalic, *Digital Computers Advanced Coding Techniques*. Cambridge, MA: MIT, 1954, sec 6-7.

[3] A. Reiter. (2004). *UNIVAC I Computer System* [Online]. Available: http://univac1.0catch.com/

[4] J. Backus, *The History of Fortran I, II, and III*. San Hose, CA: ACM, 1978, pp 166-169.

[5] E. Loh, *The Ideal HPC Programming Language* [Online]. Available: http://queue.acm.org/detail.cfm?id=1820518

[6] J. McCarthy, *History of Lisp*. Stanford, CA: ACM, 1978.

[7] H.T. deBeer, *The history of the Algol effort*. M.S. thesis, Ph.D. dissertation, Dept. Mathematics and Computer Science, Technische Universiteit Eindhoven, 2006, p. 34.

[8] ACM. (2016). *Collected Algorithms* [Online]. Available: http://calgo.acm.org/

[9] D. Gtirer, *Women in computing history*. inroads (SIGCSE Bulletin), *34*(2). P 119.

[10] R. Mitchell. (2006). *Cobol: Not Dead Yet* [Online]. Available: http://www.computerworld.com/article/2554103/app-development/cobol--not-dead-yet.html

[11] M. Richards. *BCPL: the language and its compiler*. Cambridge, MA: Cambridge University Press, 1981.

[12] K. Thompson. (1978). *Users' Reference to B* [Online]. Available: https://web.archive.org/web/20150611114427/https://www.bell-labs.com/usr/dmr/www/kbman.pdf

[13] D. Ritchie. (1993). *The Development of the C Language* [Online]. Available: https://www.bell-labs.com/usr/dmr/www/chist.html

[14] P. Sylvester. (2011). *Simula* [Online]. Available: http://www.edelweb.fr/Simula/#7

[15] B. Stroustrup. (2016). *Bjarne Stroustrup's* FAQ [Online]. Available: http://www.stroustrup.com/bs_faq.html

[16] Python Software Foundation. (2017). *Python Frequently Asked Questi*ons [Online]. Available: https://docs.python.org/3/faq/

[17] Python Software Foundation. (2008). *Why is Python slower than the xxx language* [Online]. Available: https://wiki.python.org/moin/Why%20is%20Python%20slower%20than%20the%20xxx%20language

[18] J. Vanderplas. (2016). Why Python is Slow: Looking Under the Hood [Online]. Available: https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

[19] T. Todorov. (2013). *Understanding .NET Just-In-Time Compilation* [Online]. Available: http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[20] S. Oaks. (2014). *Java Performance: The Definitive Guide* [Online]. Available: https://www.safaribooksonline.com/library/view/java-performance-the/9781449363512/ch04.html

[21] Bad Logic Games. (2016). *RoboVM is no more, what now?* [Online]. Available: http://www.badlogicgames.com/wordpress/?p=3925

[22] Xamarin Inc. (2017). *Compiling for Different Devices* [Online]. Available: https://developer.xamarin.com/guides/ios/advanced_topics/compiling_for_different_devices/

[23] C. Lattner. (2017). *The LLVM Compiler Infrastruct*ure [Online]. Available: http://llvm.org/

[24] Unity Technologies. (2017). *IL2CPP* [Online]. Available: https://docs.unity3d.com/Manual/IL2CPP.html