Lessons to be learned from esoteric programming languages

T. W. of Finity

version 1.00, first published on 2024/02/26
on the *ZedLX.com* website in the section *Articles* / *Programming language design blog*

Abstract. Two popular esoteric programming languages, *Ook!* and *BF* were examined in an attempt to find interesting language design properties that could be applied to the *ZedLX* programming language. Several usable design properties were detected upon closer examination of those two esoteric languages. The detected design properties are discussed in detail in this article, including conclusions from a subjective perspective of the article's author.

1. Introduction

The design of the *ZedLX* programming language was influenced by a wide variety of programming languages, too many for them all to be explicitly mentioned in this article. As the principal designer of the *ZedLX* language, I would like to emphasize one unexpected language that has provided invaluable inspiration and insight for the design of the *ZedLX* language. This essential predecessor of *ZedLX* is the language *Ook!*, a simple esoteric programming language, itself inspired by another simple esoteric language, *BF* [1].

The languages *Ook!* and *BF* are highly popular among programming language hobbyists and enthusiasts, despite their high level of difficulty. [2][3] The Ook! language has itself inspired many other programming languages [4], which indicates that it might be regarded as a very inspiring language. Thus the questions arises: do those languages bear some interesting properties hidden behind their uninviting appearance?

2. Being simple to explain

“Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.”

- Antoine de Saint-Exupéry, Airman's Odyssey

Simplicity is an often overlooked aspect of a programming language. A common evolution path of a popular programming language encourages it to swallow, one by one, every imaginable and popular feature that can be easily integrated into the language [5]. This is a normal consequence of catering to the community of programmers who use the language for a wide variety of purposes. After a few years of development on such a path, the language increases in complexity to the point where it can not be made suitable for programming beginners anymore.

In contrast to such trajectories, simplicity might be one of the key features that contributes to constantly high popularity of languages like *Ook!* and *BF*. These languages are so simple that they can be completely explained to beginners in less than a few hours, while the same task pertaining to some general purpose programming language requires much more time, even when restricted only to the basics of the language.

This important clue from languages *Ook!* and *BF* was taken into account when *ZedLX* language was designed. The amount of effort required on learning basics *ZedLX* before the language becomes interesting and somewhat usable was intentionally kept low; in this aspect the designers of *ZedLX* attempted to make the language more similar to the *Ook!* and *BF* languages.

3. Structure of computer programs

Ever since the Dijkstra's famous letter *“Go To Statement Considered Harmful”* [6], much has been written about improving the structure of computer programs. Many new approaches were put forward, of which two can be clearly distinguished: Python's approach of mandatory indentation, and C-like approach of free-form syntax.

Even in the apparently homogeneous camp of free-form languages, there are significant disagreements regarding the exact form that best indicates structure of programs. For example, there are many conflicting indentation styles in C-alike languages, where each organization, group or author prefers a different indentation style.[7] *Ook!* language also belongs to this group, meaning that it allows any desired indentation style while prescribing none in particular.

ZedLX expands on this widely accepted principle of free-form indentation by allowing an additional syntax for some blocks of statements, called the “statement list” syntax [8]. The main purpose of the statement list syntax is twofold: first, to reduce the number of braces and the number of accompanying indentation problems; second, to provide an alternative to statement block syntax in a way that makes the structure of statement blocks easier to understand.

4. Pronounceability of programming languages

One important advantage of the *Ook!* language, compared to the vast majority of programming languages in common use, is the property of being pronounceable in an obvious way. Pronounceability might be an important feature of a programming language for beginners, as it is likely to lead to easier collaboration and reduced number of misunderstandings [9].

The language *Ook!* appears to have been designed for easy pronunciation not only for humans, but also for some of their closest relatives [10], which is a commendable property [11]. However, the *ZedLX* language is designed for a much narrower set of potential users, namely only for humans who would like to learn computer programming. In this regard, it was deemed more important to make *ZedLX* similar to other popular programming languages, then to make it easily pronounceable across species.

In an attempt to get closer to the high pronounceability of *Ook!*, the design of *ZedLX* strives to reduce the number of unpronounceable characters, compared to other mainstream programming languages. The characters that are commonly difficult to pronounce are many of the non-alphabetic characters, widely used in most programming languages.

In mainstream languages, some of the most problematic characters to non-ambiguously pronounce are the characters “^”, “&” and “`”. Can the reader, immediately and off the top of head, tell what is the correct, easily understandable, and non-ambiguous pronunciation of those characters? Note that the word “and” is ambiguous, especially in a programming language. The non-ambiguous version, now humorously known as “ampersand”, has a humiliating origins, preventing utilization of this word in any serious endeavor. For those reasons alone, we recommend the following non-ambiguous pronunciation of the mentioned characters: “acme”, “grandma” and “tock”, in respective order. Furthermore, the following non-ambiguous pronunciations of other characters are recommend when reading aloud source code of *ZedLX* programs: “#” as “hash”, “!” as “bang”, “.” as “dot”, “$” as “pharma”, “~” as “waver”, “@”as “ater”, “?” as “quer”, “'” as uniquote, “"” as “biquote”, “\” as “fallbar”, “|” as “verbar”, “/” as “risebar”, “_” as “underbar”, “%” as “pers”, “*” as “star”, “=” as “equor”, “<” as “lessor”, “>” as “greator”.

5. Turing-completeness of *ZedLX*

The language *Ook!* is proven to be Turing-complete [12][13], while, at the time of writing this article, *ZedLX* is not yet proven to be Turing-complete. An easy way to prove Turing-completeness of *ZedLX* is to write a*n* interpreter of the *Ook!* language in *ZedLX*. Therefore, one important use of the *Ook!* language is that it makes it easier to prove Turing-completeness of new programming languages, and this has direct implications for the question of Turing-completeness of *ZedLX*.

Since the *Ook!* interpreter for the *ZedLX* is easy to imagine, the authors of *ZedLX* felt that it was not a priority to explicitly write an *Ook!* interpreter in the *ZedLX* language. Instead, it is sufficient to simply discuss how to map features of the *Ook!* language into the features of *ZedLX*. For example, the *Ook!* language allows a statement "Ook? Ook?". While the exact substitute for this statement in the *ZedLX* language is still uncertain at the time of writing this article, the preliminary analysis suggests that the equivalent might be the empty statement. Similarly, as *ZedLX* currently lacks the ability to output characters to a standard output, some substitute for the statement "Ook! Ook." has to be found in order for Turing-completeness of the *ZedLX* language to be easily proven.

6. Grammar-indicated syntax

The syntax of mainstream programming languages leaves much to be desired when compared to the clean, simple, and easy-to-understand syntax of programming languages *Ook!* and *BF*. Almost all common programming languages suffer from the problem of context-sensitive analysis, necessitating ad-hoc style parser hacks involving symbol tables, often multiple.[14]

The syntax of languages *Ook!* and *BF* has been analyzed in detail by designers of *ZedLX*, as an essential guidance for syntax simplification. As a consequence, the grammar of *ZedLX* was designed to not be context-sensitive, and the *ZedLX* parser was implemented without any symbol tables.

At the first glance, it might appear that this improvement is just an implementation detail; however, such a conclusion is almost certainly false. The context-sensitive syntax of most programming languages is probably making those languages more difficult to learn. The syntax of *ZedLX* is in the simpler class of LL(*) languages [15], which should make the language easier to learn, analyze and understand. Therefore, by adopting an important syntactic lesson, primarily based on the *BF* language, the designers of *ZedLX* have also improved its learnability.

Careless accumulation of syntactic elements might be a result of a language designer's hidden desire to quickly occupy the greatest possible share in the competitive market of programming languages. Afterwards, many excuses can be easily produced to purportedly explain why the language has a convoluted syntax. Here I provide two examples of some well-known instances of such behavior: “the lexer hack” [16] and “the most vexing parse” [17] of the C++ language.

I hope that future programming languages designs will recognize the importance of clean syntax, exemplified by the syntax of the *BF* language. The clean syntax is important to facilitate easier language learning, language usage, and language analysis. All mainstream programming languages are designed with an apparent primary goal of quick and easy implementation of numerous features. I advocate for programming languages that are primarily designed for the programmers who will be using them, instead of languages primarily designed for a quick grab of the market-share.

7. Redundancy reduction attempts

While the *Ook!* language strives for multi-species usability, this approach has necessarily led to many compromises when viewed from the human-only perspective. Other species might find it unavoidable to use vocalizations that are barely differentiable by humans, while on the other side the specific vocal apparatus of human beings prefers only an anthropocentric class of vocalizations. When a compromise is made by including only those sounds that are pronounceable and differentiable by multiple species, the end result is unwanted redundancy when analyzed from a perspective of any single species [18]. This highlights common problems of languages designed for highly disparate use cases, which includes designs of most of the general-purpose programming languages.

The humans themselves appear to be in a love-and-hate relationship with redundancies. The most widely used programming languages have ample redundancies, even when viewed from a human-only perspective. The humans are, apparently, willing to tolerate the redundancies in a given programming language as long as they are not detrimental for programming. If a redundancies can somehow be ameliorated, the humans will enthusiastically adopt the solution, probably because removal of redundancies makes it easier to perform modifications of source code. The examples of such behavior include the acceptance of numerous “class wizards”, identifier renaming tools, and the acceptance of type inference mechanisms in programming languages.

The *ZedLX* language expands on the commonly accepted mechanisms of redundancy reduction in several ways. The most important new mechanism, directly inspired by the problem of abundant redundancies of *Ook!* and by the syntactic elements of BF, is the feature of unmarked initializers [19]. This feature relieves the programmer of unnecessary type specifications when the appropriate type can be easily inferred. There are other mechanisms that serve the same or similar purpose. In order to increase the brevity of this article, I will challenge the readers to attempt to find them by themselves.

8. Valid critique is priceless

On an abstract basis, constructive critique has been recognized as an indispensable aspect of science and research. It manifests itself most apparently in the form pre-publication or post-publication peer reviews. While academic processes are apparently designed to allow for constructive critique in theory, I question whether the ensuing effects might often be the opposite in practice.

The essential problem with any critique is than no one likes to be critiqued, and especially not if the person is an established academic or scientist held in high regard. It is not only that people dislike critiques about their own work, but the same is true for groups of people, whether working together in an organization, or being just a loosely connected group sharing similar interests. Consequently, those groups, whether consciously or unconsciously, make it difficult for any substantial critique about them to receive the required attention. Such outcomes can easily be accomplished by various methods in the existing systems, for example by double-standards in the “gray” areas, lack of blinding in reviews, or by simply ignoring the criticisms, to name a few potential methods.

The first esoteric programming language, INTERCAL [20][21], is in many ways an obvious critique of other contemporary programming languages, even though it never provides critiques in explicit expressions. Instead, a person interested in INTERCAL is simply allowed to arrive at own conclusions. Such a situation raises a question: like INTERCAL, do other esoteric programming languages also harbor non-explicit critiques?

The languages *BF* and *Ook!* can be perceived as a critique of academic disciplines of programming language research, computing and computer science; perhaps intentionally dispatched through alternative channels to avoid unfair censorship. If we were to understand those two languages as concealed critiques, then what are the messages they are trying to covertly convey?

Both languages share a striking resemblance to several models of abstract machines proposed by academia in the preceding decades [22]. Those abstract machines have received a fantastically high level of attention in academic circles [23]; those machines have been taught to generations of students [24], and they have been branded as a form of serious and accepted research.

Why is then, by what criteria, the research on *BF* and *Ook!* not a “serious” and “accepted” research? Who is the producer of the so-called criteria, who is the judge, and in which way are they held accountable? Both the producer and the judge of criteria are groups of researchers, invariably as biased as any other human beings are, and held accountable to no one.

The exact same methods in their various forms, by which research on *Ook!* and *BF* has been ignored, can also be employed to silence a vast range of valid critiques directed at the core of the academic system [25]. The academic system itself is sustained by the group of people who are its supporters; this group is just as biased as any other group of people is, and it doesn't like to hear serious critique when it is directed against its core principles. The languages *Ook!* and *BF* might be suggesting that the academic system has failed to provide a viable mechanism by which critiques can be directed against it. In this way, the academic system has become blind to certain kinds of critiques and repeatedly fails to hear suggestions about serious reforms.

In the unconventional words of the *Ook!* language I read a manifest and clairvoyant message, much clearer than any academese [26] that I have ever stumbled upon. The message reads as follows: the academic system is obsolete, biased and broken beyond repair; it has been superseded by the Internet and the variety of other new technologies spawned by the computing revolution; new centers of education and research are being created by those new technologies; the dominant forms of education and research in the future will be those that fully embrace new technologies of the Digital Age.

9. Conclusions

The article discusses several ways in which new programming languages can be improved by lessons learned from esoteric programming languages *Ook!* and *BF*. The enduring popularity of those languages indicates that important new insights might be discovered by analyzing them. This article has discussed several insights discovered upon closer examination of those languages. It was found that the discovered insights were applicable to the design of the *ZedLX* language. This article has also presented and discussed the ways in which several lessons from *Ook!* and *BF* were applied to the *ZedLX* language, one lesson per article section.

The discussion in this article also indicates that current systems of education and research may be inadequate in the rapidly advancing technological conditions of the recent times. The current systems appear to be systematically disregarding important kinds of research and data, and also disregarding new methods of education. The essential malfunctions of current systems may be a consequence of conformism and of many unavoidable human biases, especial the status quo bias. The old systems will be unable to reform themselves and will likely make an attempt to cling to power. The end result is likely to be a fusion of old and novel systems, but only after the old systems are forced to radically reform themselves.

References

1 Haupt, Michael. "Implementing Brainfuck in COLA." (2008). pp. 3.

2 Morr, Sebastian. "Esoteric Programming Languages." (2015). pp. 3.

3 Alan Jay Perlis. “Epigrams on programming.” In: ACM SIGPLAN Notices 17.9 (Nov. 1982), pp. 7–13.

4 Esolang. "Trivial brainfuck substitution." Retrieved February 20, 2024 from https://esolangs.org/wiki/Trivial_brainfuck_substitution

5 Al-Qahtani, Sultan S., et al. "Comparing selected criteria of programming languages java, php, c++, perl, haskell, aspectj, ruby, cobol, bash scripts and scheme revision 1.0-a team cplgroup comp6411-s10 term report." arXiv preprint arXiv:1008.3434 (2010).

6 Dijkstra, Edsger W. "Letters to the editor: go to statement considered harmful." Communications of the ACM 11.3 (1968): 147-148.

7 Broekhuis, Stijn. The importance of coding styles within industries. BS thesis. University of Twente, 2021.

8 T. W. of Finity. "ZedLX - Statement Lists, part 3" (2018). Retrieved February 20, 2024 from https://zedlx.com/basics/statement-list-whitespace/part03

9 Porter, Leo, et al. "A multi-institutional study of peer instruction in introductory computing." Proceedings of the 47th ACM Technical Symposium on Computing Science Education. 2016.

10 David Morgan-Mar. "Ook!" (2002). Retrieved February 20, 2024 from https://www.dangermouse.net/esoteric/ook.html

11 Echeverri, Alejandra, et al. "Approaching human-animal relationships from multiple angles: A synthetic perspective." Biological Conservation 224 (2018): 50-62.

12 Böhm, Corrado, and Giuseppe Jacopini. "Flow diagrams, turing machines and languages with only two formation rules." Communications of the ACM 9.5 (1966): 366-371.

13 Rogozhin, Yurii. "Small universal Turing machines." Theoretical Computer Science 168.2 (1996): 215-240.

14 Laurent, Nicolas, and Kim Mens. "Taming context-sensitive languages with principled stateful parsing." Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering. 2016.

15 Parr, T., Fisher, K., “LL(*): the foundation of the ANTLR parser generator.” In Proceedings of PLDI 2011, pp. 425-436, 2011.

16 Bendersky, Eli. (July 05, 2012). "How Clang handles the type / variable name ambiguity of C/C++". Retrieved February 20, 2024 from https://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc/

17 Meyers, Scott. Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14. " O'Reilly Media, Inc.", 2014. pp.51.

18 Authors of Inside Edition. "Koko the Gorilla's Best Moments: From Sign Language to Meeting Mister Rogers." (2018). Retrieved February 20, 2024 from https://www.youtube.com/watch?v=G4QQ8Mfjb_g

19 T. W. of Finity. "ZedLX - The Type color, part 3" (2018). Retrieved February 20, 2024 from https://zedlx.com/basics/type-color/part03

20 Woods, Donald R., and James M. Lyon. "The INTERCAL Programming Language Reference Manual." (1973). http://3e8.org/pub/intercal.pdf

21 Morr, Sebastian. "Esoteric Programming Languages." (2015). pp. 4.

22 Érdi, Gergő. "From Register Machines to Brainfuck, part 1." (6 September 2010). Retrieved February 20, 2024 from https://erdi.dev/blog/2010-09-06-from_register_machines_to_brainfuck,_part_1/

23 Jekovec, Matevz, and Andrej Brodnik. "Survey of the sequential and parallel models of computation: Technical report LUSY-2012/02."

24 Bruni, Roberto, and Ugo Montanari. *Models of computation*. Springer International Publishing, 2017.

25 Chmutina, Ksenia, Wesley Cheek, and Jason von Meding. "“Critique is not a verb”: is peer review stifling the dialogue in disaster scholarship?." Disaster Prevention and Management: An International Journal 31.4 (2022): 387-397.

26 Pinker, Steven. "Why academics stink at writing." The chronicle of higher education 61.5 (2014).