Language Without Code: Intentionally Unusable, Uncomputable, or Conceptual Programming Languages

Daniel Temkin
Independent Artist,
609 Kappock St., #3E, Bronx, NY 10463
USA
---
daniel@danieltemkin.com

ABSTRACT

The esoteric class of programming languages, commonly called esolangs, have long challenged the norms of programming practice and computational culture. Esolangs are a practice of hacker/hobbyists, who don't primarily think of their work as art. Most esolangs are experiential works; we understand the languages by writing code in them. Through this action, the logic of the language becomes clear. However, a smaller subset of esolangs make their point not through actively writing code, but instead by simply contemplating their rules. We can think of these esolangs as conceptual rather than experiential. Some are designed in such a way that they don't allow any code to be written for them at all. By stepping away from usability, the conceptual esolangs offer the most direct challenge to the definition of programming language, a commonly used term which is surprisingly unspecific, and usually understood through utility, despite the fact that programming languages predate digital computers. This paper delves into the conceptual esolangs and looks at their challenge to the idea of programming languages.

KEYWORDS

Esolangs; Programming Languages; Esoteric Programming Languages; Fluxus; Oulipo.

ARTICLE INFO

Received: 15 November 2017
Accepted: 27 November 2017
Published: 22 December 2017

https://dx.doi.org/10.7559/citarj.v9i3.432

1 | INTRODUCTION

Esolangs (for "esoteric programming languages") are a subversive practice within computer science, taking programming language design to places far outside of practical utility. Some ask programmers to give up control of which command will fire first (David Morgan-Marr's Whenever) or to encode commands into images (Piet, also by Morgan-Marr) or express commands across a 2D plane, to be triggered by lines of code running up, down, to the left or right, or off one side of the page to the other (Chris Pressey's Befunge). Each of these languages investigates programming by postulating what-if scenarios and then designing a language around them.

Most esolangs are experiential works; to understand the language (e.g. what is it like to encode commands into colour changes across an image?), you write code—in the case of Piet, a visual aesthetic emerges from the language's rules—but it takes a fair amount of effort, and usually a community of programmers, to find it. Scott Feeney, founder of the largest wiki and archive of esolangs, esolangs.org, puts it this way:

> *"I think what's most interesting about esolangs is the conversation between languages, which ask questions, and programs written in those languages, which answer the questions. When you build a new esoteric language with a weird set of constraints, you get people thinking: I wonder if I can do X in this language? I wonder if there's a way to do Y? And figuring that out, by writing programs that do X and Y, can be a fun challenge."* (Feeney, 2015)

However, not all esolangs have concepts that lead to working code. Since esolangs are an experimental form, it is common for the limits of the language to not be immediately clear. For instance, take the language Three Star Programmer, created in 2015 by ais523. This language asks programmers to write code with three levels of indirection: pointers to pointers to pointers to memory cells. A program in Three Star Programmer is in the form of a string of numbers; each can be thought of as both raw data and as a pointer to another location in memory, where that memory is also the code itself. The numbers are consecutively read, each dereferenced three times (meaning the interpreter jumps to the location corresponding to the number in that cell), until we get to a final number which is then incremented. However, that final location is also a pointer (to a pointer to a pointer), meaning where it points has just changed. On the esolangs wiki, ais523 (the creator of the language) says "it's very hard to actually write anything in the language, because of the fundamental nature of the language, in which everything affects everything else and no change is really reversible." Despite this, at the time of writing, it was not yet known how powerful it was, in terms of the language's potential to represent algorithms. Between the time of writing and final publication of this paper, ais523 has reported that the language has indeed been proven Turing Complete. ("Three Star Programmer - Esolang," 2015).

Such potential is measured through a classification called computational complexity. Many esolangs strive for the most complex category, Turing Completeness, understood as tantamount to what computers can do, and which most mainstream languages belong to (Kandar, 2013). The reason for this is aesthetic: showing that a strange idea is also very powerful. For example, the highly influential esolang brainfuck (typically spelled lower-case) expresses all code in eight commands, each represented by a punctuation mark. What makes brainfuck interesting is how such a minimal language with such odd logic is provably as powerful a language as Python or C, despite having no built-in representation of the number 2 or of the action of multiplication. With Three Star Programmer, we have an example of a new idea, something that is so odd that it remained unclear exactly how to write performative code in the language. It could have taken years to be proven Turing Complete or a lesser computational class (e.g. a Finite State Machine), which would itself be a fascinating result. At this point, TC is proven, but a path to practical coding in the language is still elusive. (Temkin, 2013).

So why would an esolang designer (or "esolanger") set out to deliberately design a hobbled language, one with no potential for Turing Completeness, or one not even runnable on current technology? Where we can think of brainfuck and Three Star Programmer as experiential esolangs—languages we understand through the experience of writing code in them—the nonprogrammable programming languages have a different agenda. Experiential esolangs are run on the machine, but the nonprogrammable ones are conceptual works: they can only be run in our heads. This paper explores these languages.

2 | WHAT IS A PROGRAMMING LANGUAGE?

Conceptual languages, like their experiential cousins, are a radical challenge to what programming languages are: how they are designed and how they can function. However, in their lack of codability, they perhaps more directly challenge how the very term *programming language* is defined. Surprisingly, this concept is hard to pin down. It is usually defined through utility: a programming language is used to express commands to a computer. The Merriam Webster definition, Wikipedia's definitions (both the longest-posted one and the one currently posted at the time of writing) are all variations of these, sometimes with "formal language" mentioned, which at least points to a substantial difference from natural language.

Wikipedia helpfully notes that the first programming languages were used for automation before existence of the digital computer, which points to perhaps the biggest issue with the term: what is a computer and do we need it to execute code? Before the first modern digital computer, we had the Turing Machine, a purely theoretical automaton used for mathematical proofs about computation. Are we defining programming languages in terms of computers as they are today? As we imagine them tomorrow? Or always in terms of the theoretical machine, as the first languages were designed? Microsoft, IBM, and others have designed quantum computing languages for computers that don't yet physically exist and perhaps won't, at least as they are currently conceived (Simonite, 2017). While we have a system of computational complexity that shows the algorithmic potential of a language, as we

will see through example, there is no established lower boundary of what we expect programming languages to be capable of in order to consider them languages. As the (possibly apocryphal but wholly-in-character) quote from Edsger Dijkstra goes, "Computer science is no more about computers than astronomy is about telescopes" (Dourish, 2017).

Chris Pressey, creator of the esolang Befunge and its mailing list, where much of the early discussion around esolangs took place, makes a similar point about esolangs:

> *"[T]hey're made up of concepts, and these concepts would exist even if our computing equipment wasn't electronic, or wasn't digital, or if we didn't have computing equipment at all. It's just that having computing equipment makes it a lot easier to design and experience these programming languages."* (Pressey, 2013)

But if we remove the computer from the programming language, what are the other aspects of programming languages left to explore? Technical books on compilers get more exact about what a programming language is, the qualities that, while secondary in terms of the ordinary definition, get more to the heart of what programming languages actually are. The popular textbook *Programming Language Pragmatics* describes languages this way:

> *"Unlike natural languages such as English or Chinese, computer languages must be precise. Both their form (syntax) and meaning (semantics) must be specified without ambiguity, so that both programmers and computers can tell what a program is supposed to do."* (Scott, 2006)

Programming languages as logical systems lacking in ambiguity, along with their relentlessly imperative tone (even for the non-imperative languages, which are different in form but not in mood)—are perhaps what most clearly differentiate programming languages from natural language. Esolangs like brainfuck add the semblance of ambiguity (to us human readers or programmers) through complexity, but the language is still clearly defined in both its syntax and semantics; each brainfuck program still has only one semantic interpretation to the machine. However, even this definition fails to separate a language like C++ from the form of English we use to speak to natural language processing systems like Alexa, which likewise translate to computer instructions, but which we would not likely want to categorize as programming language.

Another potential objection is that we expect the semantics of such languages to be opcodes (individual machine instructions): writing to memory, moving data in memory, adding two numbers, etc. As many a creative coding teacher has illustrated by having students enact a virtual computer, impersonating the CPU and other parts of the machine, there is nothing about "copy the value from memory cell 103 to register A" that we can't capture symbolically and effectively by transferring a piece of paper from one student's hand to another.

2.1 LANGUAGE SCHISMS

Malbolge, created by Ben Olmstead in 1998, was designed to be the most difficult language to write code in. Each program runs in a giant loop, requiring a special operation to indicate the end of program. Each command self-encrypts after it runs, mutating from one command to another. All math is done using a counter-intuitive operation called the "crazy" operator, which uses base 3 math, a particularly non-intuitive base for programmers.

When Olmstead put Malbolge out into the world, he released no programs for it; he has to this day never written a Malbolge program. It took three years before its first program, a *Hello World* program, was written for it. While this fact is often mentioned in terms of Malbolge, Olmstead sees it as a bit overblown, as it took a while for Malbolge to be noticed by those who took a serious stab at writing code for it (Olmstead, 2014). This program was not written by hand, but by another program, essentially treating Malbolge as a cryptanalysis problem. This does away with the programming language as a form of human/computer interaction; it is a language entirely generated by one program for another to read. Such languages do exist in mainstream computing: an example is PostScript, a document layout language (and ancestor of the PDF format) which began as human-written mark-up, but is now nearly always generated by a layout program based on designs made using a visual interface (Weingartner, 2006).

Although we can't internalize the logic of Malbolge as we do with most esolangs, the sign of its experiential status is the way that this language, developed in a

single afternoon, has inspired hundreds of hours of coding, with analyses of the cycles of Malbolge's commands, and studies of Malbolge algorithms (Scheffer, 2015).

Less noted than Malbolge's general difficulty is the fact that the documentation and the actual compiler created by Olmstead are not entirely in agreement. From an interview for esoteric.codes:

> *I know there is a mismatch between the documented and implemented tables. I have noticed that some people decide that the bug is in the specification, and others decide the bug is in the implementation; it certainly makes Malbolge harder to use, and fragments the user community (such as it is).*
>
> *If I were to make a Malbolge 2000, I would definitely make the documentation subtly wrong.* (Olmstead, 2014)

We expect a language to have a code processing tool (a compiler or interpreter) to transform it into machine code (or other machine-friendly formats). This tool also serves as a gatekeeper, enforcing the syntax of the language; if we try to compile FORTRAN code as C, the C compiler will reject it as invalid: a syntax error on every line. The compiler (and runtime system if it exists for the language) is seen as the materialization of language itself; it's what we interact with when we write code. However, these executors are not informational: they do not reveal the rules of the language. If we are given a clearly expressed syntax for the language (in a formal notation such as BNF or Backus-Naur form), along with its semantics and specifications of its runtime if needed (garbage collection, special optimizations), we could write our own compiler for the language. This has been done many times for widely-used languages such as C, with each compiler introducing its own quirks and minor differences, obscure enough that they have not been fixed; perhaps best known by the example of the null program; the completely empty file, which is valid in some versions of C and not others (Montfort, 2014).

By pitting the performance of the only available interpreter against the formal definition of the language, Malbolge undermines the sense that either one is the true language.

INTERCAL (somehow short for "Compiler Language With No Pronounceable Acronym"), generally considered the first esolang (it was created in 1972), included a set of documentation filled with nonsensical diagrams and misleading statements. Some aspects of the language were left to be discovered, or intentionally ambiguous. INTERCAL required the keyword PLEASE scattered throughout the program. Not enough PLEASEs and the entire program would be ignored, as the interpreter found the program too rude. Too many PLEASEs and the interpreter saw the programmer as simpering and also ignored the entire thing. The correct proportion of PLEASEs to commands was not in the documentation, leaving the programmer to discover on her own how to get the program running (Bratishenko, 2009; Smith, 2007). The PLEASE command also brings our attention to the one aspect that nearly all programming languages have in common: the relentlessness of their commanding tone.

When INTERCAL was rewritten as C-INTERCAL (by Eric S. Raymond in 1990), making it available to a wider audience, he had to choose which features were critical to maintain and which to modernise. He chose to better document the language (spoiler: it's "approximately 3 non-polite identifiers for every polite identifier used") (Raymond, 2015).

2.2 UNCOMPUTABLE LANGUAGES

Brainfuck has been an inspiration for hundreds of derivative languages, in part due to its simplicity of design: an easy way to get to Turing Completeness. According to the esolangs wiki, Chris Pressey has called it the "twelve-bar blues of esolangs" (“Brainfuck - Esolang,” n.d.).

Lenguage embraces the minimalism of brainfuck, and uses the same command set, with a different encoding of signifiers. Lenguage’s name is a play on words; LEN() is the command in many languages that reports the length of a string. In Lenguage, the length of the program in characters is the only thing that matters. A C program could be a Lenguage program as well, if its length were correct to correspond to a series of commands (“Lenguage - Esolang,” 2014).

Lenguage asks the question: do we need both 0 and 1? If we're going for pure minimalism, why not just one symbol? With a vocabulary of undifferentiated symbols (such as 1s), we could represent code with anything: the length of a line, or enumerating each in a pile of rocks.

In Lenguage, this is performed by translating each of brainfuck's commands into a binary sequence: 000 for + (increment), 001 for - (decrement) etc., and set in order, to produce a single number. A program with the length of that number will be read by the Lenguage interpreter by translating that number into binary and reading the sequence, giving us the program.

The *Hello World* program for Lenguage is any file with 17,498,005,810,995,570,277,424,757,300,680,353, 162,371,620,393,379,153,004,301,136,096,632,219 ,477,184,361,459,647,073,663,110,750,484 characters. At 1.75 * 10^102, it's more than a Googol characters. This means the *Hello World* program, if stored at the atomic level (counting individual atoms to determine the program), would be larger than the size of the known Universe.

If Lenguage had adopted the approach of the language Spoon, another binary-based brainfuck derivative, it would be in a somewhat more usable state. Spoon, created by S. Goodwin in 1998, took brainfuck and represented each of the commands with a binary sequence, similar to Lenguage. Where Lenguage took a minimalist approach to variety in input, Spoon allows us to simply write the number, rather than use the length of the sequence of the number. Furthermore, Spoon uses Huffman-encoded binary sequences, meaning the most commonly used commands (+ and -) are represented with the shortest sequence in binary digits; + is a single 1, - is 000. Had Lenguage used Huffman-encoding, its Hello World program would be only nineteen quattuorvigintillion, 10^76, only the informational content of a one-solar-mass black hole.

Chris Pressey created a derivative language of Spoon, called You are Reading the Name of this Esolang (pronounced "You are Hearing the Name of this Esolang"). It is Spoon with two additional symbols; opening and closing brackets. Code held in the brackets are read as complete Spoon programs in themselves and executed first. If they complete, they are translated to 1s and dropped back into the original sequence. If they do not halt (get stuck in an infinite loop), they are translated into 0s (“You are Reading the Name of this Esolang - Esolang,” n.d.).

While some trivial infinite loops can be detected, Alan Turing proved that there is no generalized solution to determining whether a piece of code will halt; this is known as the Halting Problem (Turing, 1937). You are Reading the Name of this Esolang has taken a fundamental computational problem and moved it from the performance of code into the lexical analysis of code. While some You are Reading the Name of this Esolang programs may be validated by a human reader or the compiler, it has been proven definitively that the machine has no general way to validate a sequence as being a You are Reading the Name of this Esolang program. It could take exponential time, or possibly forever, to compile such a program. Rather than being larger than the universe, You are Reading the Name of This Esolang is beyond the reach of any currently conceivable technology.

Traditional programming languages try to remain unobtrusive, to let us see how the code will function as clearly as possible, rather than drawing attention to its actual structure as symbols on a screen, esolangs frequently bring our attention back to the surface layer of the language. With a language like You are Reading the Name of This Esolang, the name alone is a constant reminder that we are dealing with something very different, where the language is not something we can easily see through, but a structure to be wrestled with, or a puzzle for us to ponder and consider in its own right.

2.3 LANGUAGE AS PURE DOCUMENTATION

Immateriality is a returning theme in esolangs, perhaps drawn from the fact that languages are already almost nothing: sets of rules, with no particular implementation.

The best known of these is Whitespace, a fully functional language you code with just three whitespace characters: space, tab, and return. A Whitespace program can be a file that looks entirely empty. While Whitespace is Turing Complete, it's a language we can consider conceptual in the sense that we experience it by considering its aesthetic. We don’t learn a lot by actually creating programs; the language is a fairly typical procedural language; what’s exciting about it is its strange concept and vocabulary.

Incidentally, C++, a particularly whitespace-ambivalent language (unlike, say, Python, where indentation has syntactic meaning), nearly had meaningful whitespace. Its creator, Bjarne Stroustroup, suggested allowing the overloading of whitespace, meaning C++ programmers could assign actions to it, such as to multiply two numbers, in the

interest of formatting multiplication closer like how mathematicians do, without the * symbol (Stroustrup & Park, 2000).

It would not make much sense to design the Whitespace language as less than a Turing Complete language. A language written with whitespace characters is interesting because of the surprise of it being functional ("Whitespace Tutorial," 2004). Whitespace shows that the signifiers for a language are not meaningful to the machine; it is only of limited signification for us, not for the machine, for which all symbols are essentially interchangeable.

When we take the gesture toward immateriality into the language definition itself, we get smaller and stranger languages, less capable of expression, and often severely limited in usability. Most of the following languages are treated as joke languages. The esolangs wiki has them listed as such explicitly, adding:

> *"This is a list of esoteric languages that are not of any interest except for potential humour value. Generally speaking, they are completely unusable for programming even in theory, trivial and less interesting variations on existing esoteric languages, or too underspecified to determine any potential usability.*
>
> *For esoteric languages that are potentially interesting in some way, or that are actually capable of running programs and producing a useful output, see the normal list of esoteric languages."* ("Joke language list," n.d.)

This, I believe is unfair; Whitespace itself was taken as a joke when it first launched with an announcement to *Slashdot* (to be fair, on April Fool's Day) in 2003, but has remained in public consciousness and an inspiration for embracing the immaterial in esolangs. Whitespace is generally more respected because it was a new idea at the time — although perusing the original Slashdot thread shows that, even then, there were naysayers exclaiming that it had been done before ("New Whitespace-Only Programming Language - Slashdot," 2003).

Part of this dislike comes from the so-called "theme" languages; ArnoldC and LOLCATS, where one writes code that sounds like Schwarzenegger or the lolcat meme (O HAI etc.). There's a Trump version and at least six distinct emoji languages. The problem with these languages is that they are very ordinary apart from their vocabulary. This makes it easy to dismiss the great number of interesting vocabulary-oriented languages, such as Whitespace, or, as I argue with the next few examples, some of the extreme minimalist pieces such as Unnecessary and Καλλίστη.

The legendary compiler book known as the Dragon Book describes a compiler as "a program that reads a program written in one language—the source language—and translates it into an equivalent program in another language—the target language." It also explains the Recognizer, the part of the compiler that affirms that a piece of code is legitimate in a language (Aho, Lam, Sethi, & Ullman, 2006).

The esolang Unnecessary (created in 2005 by Keymaker) can be thought of as a pure Recognizer. It reads only code that doesn't exist, and has only one possible program; a program which does nothing. Since there is no code to write, the author helpfully describes the language as "easy to learn" (Keymaker, 2005).

When one attempts to compile any file at all as an Unnecessary program, it fails with an error message. An empty document, an image, a Word document, each is rejected as insufficiently Unnecessary. Only a file which can't be found (a file location that doesn't exist on disk) succeeds to compile. The result is the creation of an empty program made up of a single instruction, the NOP (pronounced "no op" for "no operation"). This is the minimal command to generate the program as an executable. As Keymaker describes it:

> *"The main idea was that the language could not have programs, other than the kind that don't exist. (Can it have those then if they don't exist?) Then I noticed that every valid program (whatever that is) is a/the nullquine but that was more of a by-product of the main idea. Fitting nonetheless!"* (Keymaker, 2011)

A Quine is a program which prints its own source code to the screen. The Null-quine is a program with empty source code that prints its source (which is nothing) to the screen.

The idea of codeless language goes much farther than one might think. Each has its own attitude toward why nothing happens. It can be useful to think of these in terms of the null program; the program

without code, which, as explained in Nick Montfort's *No Code: Null Programs*, can still instigate activity despite its lack of content, such as logging by the compiler (Montfort, 2014). The null language is never completely without attributes; since a language is a set of rules, the refusal to enforce rules always has some reasoning, revealed in the documentation or (un)implementation.

The language Καλλίστῃ (or Kallisti), a collaborative project from 2007 led by The Prophet Wizard of the Crayon Cake and the Seven-Inch Bread, was inspired by Discordianiam, the Dada-like fake religion once popular with programmers, that plays with meaning and nonsense. Its list of rules is:

- Obey as many rules as possible
- There is plenty nothing
- Everything is true
- Everything is false
- There is only nothing
- Obey as few rules as possible

It also includes BNF notation, which shows the language is all-accepting. Unlike Unnecessary, which rejects all data, Καλλίστῃ accepts it all—but because of this, it doesn't value one type of data over another. Καλλίστῃ turns everything back into what it already was. Its transformation is from source code back to itself ("Καλλίστῃ - Esolang," n.d.).

If the "joke" languages are so disliked by much of the esolang community, it's interesting that they are not simply deleted from the wiki as being not programming languages at all. But what is a language other than a formal system? Is Καλλίστῃ's refusal to specify a signifier as something other than "anything" make it no longer a language? Does the ruleset have to be self-consistent (formal) to be a programming language? How small a gesture can one make toward programming or language for a system to qualify?

While Καλλίστῃ and Unnecessary might seem like the conclusion of how small a language can be, there are actually many others that have essentially no code. The language 2014 only worked in its name year; announced on Dec 31 of that year, no code was written for it. Since any code written after that time is invalid, it has not only no programs, but no defined grammar.

I am personally responsible for several languages that are only programmed in by accident. Inspired by work from the Fluxus movement (based in NYC in the early 1960's), these languages are two of the most commonly coded in the world, although nothing has been written for either intentionally; they take texts or events created for other reasons and interpret them as code.

2.4 ACCIDENTAL TURING MACHINES

One potential objection to these non-programming programming languages are their lack of Turing Completeness. Some esolangs belong to somewhat more limited categories; Malbolge is a Finite State Machine, like many other systems e.g. some implementations of calculators. While we are unlikely to call Malbolge not a programming language because of this, perhaps it's reasonable to set a minimal complexity, below which we would not consider a system to truly be a programming language.

A counter-argument to this is the number of systems that are actually Turing Complete that were created with no intention of using them this way. It was accidentally discovered that C++'s templating system is Turing Complete, which is a problem, as it means there is no way to know that a C++ compilation will complete, due to the Halting Problem (Veldhuizen, 2003).

The card game Magic: The Gathering has been shown to be have complex enough rules to achieve Turing Completeness. We can play Magic: The Gathering in a way that is effectively a computer.

> *"If the new token had been a Zombie rather than an Ally, a different Kazuul Warlord and a different Noxious Ghoul would have triggered, as well as the same Aether Flash. So the same would have happened except it would be all the Zombies that got +1/+1 and all the Allies that got -1/-1. This would effectively take us one step to the right."* (Churchill, n.d.)

The movement to the right hints that this implementation is a simulation of the Turing Machine. Minesweeper has been proven to be Turing Complete, at least if played on an infinite board. A very strange paper announced that the human heart has the capacity to function as a Turing Machine, which is of interest because, due to the Halting Problem, it proves that it is not possible to absolutely predict cardiac tissue's behaviour (Ostrovsky, 2009).

3 | CONCLUSION

As Deleuze showed in his study of Spinoza, a logical system is not necessarily rational (Lapoujade, Rajchman & Jordan, 2017). This is well dramatized by the experiential esolangs like Malbolge and Brainfuck. The challenge of conceptual languages, which don't ask us to write code, is quite different. A useful analogy is the difference between Oulipo practice (the group of writers centred in Paris, beginning in 1960) and that of the Fluxus event scores.

Most esolangs are Oulipian in nature. The Oulipians were writers who created constraint sets which were explored by writing works within those constraints: for example, George Perec's novel A Void followed the constraint of "write a novel without using the letter e". Similarly, the esolanger designs a language for herself or others to then figure out how to program. Like the Oulipians, the esolangers are the rats who build their own maze.

The Fluxus event scores are a bit different; they are performance scores that "merge art and every-day life," often sitting at the border of what can even be called a performance. George Brecht's 3 Lamp Events, a performance where one clicks on and off a lamp several times, is performed far more often by accident than on purpose. If one is aware of the work, the accidental combination of events can be read as a performance, carried out quite unintentionally (Maciunas, 1966).

These event scores more strongly resemble the conceptual esolangs: the languages for which we don't write code. Many of Yoko Ono's early works cannot be physically performed at all, but are meant to be contemplated, much like the conceptual esolangs. For example, her Earth Piece:

Listen to the sound of the earth turning.

1963 Spring (Ono, 1964)

The conceptual languages emphasize the immaterial nature of computation. More than bits or circuits, the materiality of software is logic, running on theoretical and virtual systems, sometimes embodied in circuits—but which tomorrow could be embodied in quantum qubits or another technology not yet dreamt of. The conceptual esolangs twist that logic and turn it against itself in poetic gestures that continue to challenge the sensibility and the limits of code.

REFERENCES

Aho, A. V, Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools.

Brainfuck - Esolang. (n.d.). Retrieved September 29, 2017, from https://esolangs.org/wiki/Brainfuck

Bratishenko, L. (2009). Technomasochism. Cabinet.

Churchill, A. (n.d.). Magic: the Gathering is Turing Complete. Retrieved September 28, 2017, from http://www.toothycat.net/~hologram/Turing/HowItWorks.html

Dourish, P. (2017). The Stuff of Bits. Cambridge: MIT Press.

Feeney, S. D. T. (2015). Interview with Scott Feeney. Retrieved September 25, 2017, from http://esoteric.codes/post/116998438745/interview-with-scott-feeney

Joke language list. (n.d.). Retrieved September 28, 2017, from http://esolangs.org/wiki/Joke_language_list

Kandar, S. (2013). Introduction to Automata Theory, Formal Languages and Computation. https://doi.org/10.1145/568438.568455

Καλλίστη - Esolang. (n.d.). Retrieved September 28, 2017, from https://esolangs.org/wiki/Καλλίστη

Keymaker. (2005). Unnecessary (another esoteric programming language). Retrieved September 28, 2017, from http://yiap.nfshost.com/esoteric/unnecessary/unnecessary.html

Keymaker. (2011). Interview with Keymaker - esoteric.codes. Retrieved September 28, 2017, from http://esoteric.codes/post/84939008828/interview-with-keymaker

Lapoujade, D., Rajchman, J., & Jordan, J. D. (2017). Aberant Movements.

Lenguage - Esolang. (2014). Retrieved September 28, 2017, from http://esolangs.org/wiki/Lenguage

Maciunas, G. (1966). Fluxfest Sale. Film Culture - Expanded Arts, (#43), 6–7.

Montfort, N. (2014). No Code: Null Programs, (December). Retrieved from http://dspace.mit.edu/handle/1721.1/87669

New Whitespace-Only Programming Language - Slashdot. (2003). Retrieved September 28, 2017, from https://slashdot.org/story/03/04/01/0332202/new-whitespace-only-programming-language

Olmstead, B.. (2014). Interview with Ben Olmstead - esoteric.codes. Retrieved September 28, 2017, from http://esoteric.codes/post/101675489813/interview-with-ben-olmstead

Ono, Y. (1964). Grapefruit.

Ostrovsky, I. (2009). Human heart is a Turing machine, research on XBox 360 shows. Wait, what? Retrieved September 18, 2017, from http://igoro.com/archive/human-heart-is-a-turing-machine-research-on-xbox-360-shows-wait-what/

Pressey, C. (2013). The Aesthetics of Esolangs. Retrieved September 25, 2017, from https://github.com/catseye/The-Dossier/blob/master/article/The Aesthetics of Esolangs.md

Raymond, E. S. (2015). Interview with Eric S. Raymond - esoteric.codes. Retrieved September 29, 2017, from http://esoteric.codes/post/130618094278/interview-with-eric-s-raymond

Scheffer, L. (2015). Programming in Malbolge. Retrieved September 29, 2017, from http://www.lscheffer.com/malbolge.shtml

Scott, M. L. (2006). Programming Language Pragmatics. Analysis (Vol. 20). https://doi.org/10.1016/B978-0-12-374514-9.00008-2

Simonite, T. (2017). Microsoft's Nadella Wants to Help Coders Take a Quantum Leap | WIRED. (2017). Retrieved September 29, 2017, from https://www.wired.com/story/microsofts-nadella-wants-to-help-coders-take-a-quantum-leap/

Smith, A. (2007). C-INTERCAL 0.29 Revamped Instruction Manual. Retrieved September 28, 2017, from http://catb.org/esr/intercal/ick.htm

Stroustrup, B., & Park, F. (2000). Generalizing Overloading for C ++ 2000.

Temkin, D. (2013). Brainfuck. Media-N. Retrieved from http://median.s151960.gridserver.com/?page_id=947

Three Star Programmer - Esolang. (2015). Retrieved September 29, 2017, from https://esolangs.org/wiki/Three_Star_Programmer

Veldhuizen, T. L. (2003). C++ Templates are Turing Complete. URL Http://osl. Iu. edu/tveldhui/papers/2003/turing. Pdf. Unpublished Manuscript, 1–3. https://doi.org/10.1.1.14.3670

Weingartner, P. (2006). A First Guide to PostScript. Retrieved September 28, 2017, from http://www.tailrecursive.org/postscript/postscript.html

Whitespace Tutorial. (2004). Retrieved from http://compsoc.dur.ac.uk/whitespace/tutorial.php

You are Reading the Name of this Esolang - Esolang. (n.d.). Retrieved September 28, 2017, from https://esolangs.org/wiki/You_are_Reading_the_Name_of_this_Esolang

BIOGRAPHICAL INFORMATION

Daniel Temkin makes images, programming languages, and interactive pieces that use the machine as a place of confrontation between logic and human thought. His esoteric.codes, 2014 recipient of the ArtsWriters.org grant, documents the history of programming languages as an art medium. He has published in journals such as Leonardo and World Picture Journal and has presented at conferences including xCoAx, SXSW, GLI.TC/H, SIGGRAPH, and Media Art Histories. He regularly performs readings from his Internet Directory project, a 37,000+ page loose-leaf book of all the .COM domains in alphabetical order; he received a commission from the Webby Awards to build an online version, a scroll of domains that takes two years to watch. He is expanding the esoteric.codes project as a 2017-18 member of NEW INC, the New Museum's incubator.