ISEA2015
Proceedings of the 21st International Symposium on Electronic Art
ISSN: 2451-8611
ISBN: 978-1-910172-00-1

Esolangs as Experiential Art

Daniel Temkin

New York, NY
daniel@danieltemkin.com

Abstract

Esolangs (for "esoteric programming languages") are a class of languages made by programmers at play, not intended for practical use. Ben Olmstead (creator of the *Malbolge* language) describes them as "pushing the boundaries of programming, but not in useful directions." [1] This paper looks at how these strange languages function as experiential art pieces, with similarities to Oulipean systems and Fluxus event scores.

Keywords

Esolangs, Esoteric Programming Languages, Language, Concept Art, Fluxus, Ouipo, Code Art

Introduction

Esolangs have been described as jokes, parodies, impractical research, scenarios of the improbable, or what can only be defined as programming languages by butchering the very definition of a programming language. [2]

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
DO ,1 SUB #3 <- #112
DO ,1 SUB #4 <- #0
DO ,1 SUB #5 <- #64
DO ,1 SUB #6 <- #194
DO ,1 SUB #7 <- #48
PLEASE DO ,1 SUB #8 <- #22
DO ,1 SUB #9 <- #248
DO ,1 SUB #10 <- #168
DO ,1 SUB #11 <- #24
DO ,1 SUB #12 <- #16
DO ,1 SUB #13 <- #162
PLEASE READ OUT ,1
PLEASE GIVE UP
```

Figure 1. *Hello World* in INTERCAL (author unknown)

There are three strategies esolangs commonly take to express an idea. The first, and perhaps more obvious given that most code is a form of text, is through their vocabulary. The other two we can think of as logic-oriented esolangs and concept languages.

Languages like *LOLCODE* and *INTERCAL* both make strange use of keywords, although with very different approaches. INTERCAL, created in 1972 (and generally considered the first esolang), functions as a parody of languages of the time. It asks the programmer to beg and plead with the machine, personifying the compiler as a capricious autocrat who allows programs to compile or not based on how much groveling is done. Its language is cryptic and confusing, a puzzle for the programmer. [3]

```
HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

Figure 2. *Hello World* in LOLCODE (Adam Lindsay)

LOLCODE (Adam Lindsay, 2007) personifies the "lolcats" meme. Starting every program with "HAI" and ending with "KTHXBYE", the script in between is full of the familiar mix of baby talk and internet slang of lolcats. [4] These languages ridicule the authority of the machine through the actual text of the programs created within them. However, the code is still intuitive: using "visible" for "print" and "kthxbye" for "exit." The lolcats concept is added to the code without obscuring the code's function.

Programming languages are formal systems; they are self-contained and closed, apart from references to computer operations. They are made up of commands which must compile down to machine code, a purely denotative space with no place for nuance: when we communicate with the machine, we can't insinuate or gesture; any ambiguity in the language is wiped out. Both these languages use elements of human language and expression to add back some of that gesture and nuance at the top level, even if it is stripped away in its conversion to machine instructions. Where commands in INTERCAL are confusing, they are only confusing to us, not the machine; likewise, at the machine level, LOLCODE commands are the same as in many other imperative, procedural languages.

Befunge and brainfuck

Although nearly all esolangs rely on vocabulary as part of their method of expression, many esolangs are less vocabulary-oriented than those two examples. In 1993, Wouter van Oortmerssen designed *FALSE*, a language for the Amiga. The objective of FALSE was to support the smallest possible compiler (written in assembler, it was 1024 bytes). To reach this objective, Wouter used single letters for commands. [5]

This obfuscated syntax was then picked up by two other languages the same year, both also written for the Amiga and using single-letter commands: *brainfuck* (usually spelled lower-case) and *Befunge*. Where FALSE was essentially a tiny version of the Forth language, brainfuck and Befunge began to explore the programming language space as systems of thought. These languages rely on logic more than vocabulary.

```
>                  v
v  ,,,,,"Hello"<
>48*,              v
v,,,,,,"World!"<
>25*,@
```

Figure 3. *Hello World* in Befunge (author unknown)

Brainfuck (Urban Müller, 1993) is Turing Complete, meaning that it can theoretically be used to write programs to do anything that can be written in, say, C++, only it is even tinier than FALSE, consisting of eight punctuation marks alone (resulting in a 240 byte compiler). In brainfuck, we can't write code that looks like "let x = 36" because all of those characters, from the word "let", to the spaces between words, are ignored. *X* means nothing to the language, and neither does 3 or 6. Instead, we move back and forth through memory using the angle brackets (< and >) until we get to a place we might think of as *x*, followed by 36 plus signs, each incrementing that memory cell from zero.

A more succinct alternative is to loop six times through the operation of adding six, like so:

```
++++++[>+++++++<-]>
```

Alternately, one could count down from zero, as in 8-bit brainfuck, subtracting one from zero gives you 255. Or one could use nested loops to reach any number which, modulus 256, leaves 36. There are a great many ways of producing that single number, and choosing one over another becomes a matter of personal style or a chance to find a clever solution. [6]

The complexity of brainfuck does not arise artificially; each brainfuck command maps directly to a command in assembly code, the substrate of most languages. Rather than making these commands more human-friendly (like how, say, the C compiler does), it refuses to ease the translation, both with its odd syntax and its tiny selection of choices, thereby directly exposing the conflict between human thinking and computer logic.

Figure 4. *Hello World* in Piet (James Dessart)

Befunge (Chris Pressey, 1993), similarly, builds on complexity that arises from a simple idea. It uses 2D code rather than the ordinary formatted strings of text. Befunge programs run up and down the page (or screen), crossing itself in horizontal and vertical lines, like a maze. Because of its snaking structure, the same command can be read multiple times in different directs, in completely different contexts and with different results. Also, it ignores any code which is not in its execution path, meaning comments and non-executed code can appear amongst real code, making it hard to differentiate what is or is not part of the program.

Building on the 2D design of Befunge, *Piet* (David Morgan-Marr, 2001) gets rid of characters entirely, using images as source code. The compiler takes a similarly serpentine path through the image, only here the change in color from one

group of pixels to the next determine commands. Both changes in hue and in lightness have meaning; because it's delta-based (the change rather than symbol itself is the signifier), correcting a mistake in the code means rewriting all the pixels that fall after it. The language was named for Mondrian, and so much of the source code mimics Mondrian's aesthetic, although the rules of the language favor a shattered and sometimes lumpy, pixelated look.

Piet unifies the vocabulary-oriented impulse with logic-oriented code. Like INTERCAL, Piet creates a space between the look of the code and its actual function, making language visible in a way it isn't in traditional languages (which opt for clarity of style to emphasize code's function over its appearance). Where INTERCAL brings in an overflow of human expression that ordinarily has little place in code, Piet encodes it into an entirely different system with its own signifiers and style, whether it's used for simple abstractions or representational images. However, Piet also has the strange logic borrowed from Befunge; like the logic-oriented languages, it is experiential in nature, a challenge for the programmers using it. This impulse to mix systems is reiterated in other work of David Morgan-Marr's, such as his language *Chef*, which uses (often extremely odd) recipes as code.

Esolangs as Performance Scores

It is tempting to compare Piet programs with Generative works, as some have a similar appearance to computer-generated images. Only here the look of the image is determined by rules which run on the *programmer*, not the machine. Geoff Cox, in his *Speaking Code*, looked at running code as performative. He sees it as a special type of performance, in that the machine always "performs" the same piece of code the same way: the speaking of the code and performance of it become flattened. [7] We can contrast this with work such as the Fluxus event scores, which leave enormous space for interpretation. Yoko Ono's *Fly Piece* (with the single instruction "Fly"), evokes many different things, leaving nothing specific for the "performer" of the score — that performer might be a reader, for whom an imaginary flight is invoked, or perhaps someone actually trying to physically interpret it. [8]

Esolangs in the tradition of brainfuck and Befunge however, re-open this possibility in the score. Because they are open-ended systems, the writing of programs within the language becomes the space for this interpretation. They are experiential — you have to program in a language in order to understand it, it's not to be passively received. And even in brainfuck, code by different programmers may feel very different. An esolang with no esoprogrammers is a sad thing, a score with no performer. A language is a prompt.

The esolanger ais523, who co-runs the esolangs.org wiki, puts it this way: "it's much more interesting if the point of view of the language is one that you can think in independently." Ais523 singles out LOLCODE as falling short of this:

> I should also mention LOLcode, which has become pretty relevant as an esolang in the non-esolang community recently, much to the annoyance of most actual esolangers. It doesn't have much intrinsic interest for most of the reasons people are interested in esolangs, being mostly a simple imperative language derivative with the keywords swapped out... However, it appeals for things like its visual appearance and general attitude, which although are IMO the least important aspect of an esolang, are one of the most immediately noticeable. [9]

To actually code in such a language does not lead to a greater understanding of its system; we essentially get LOLCODE by looking at sample code. Richer languages, even vocabulary-oriented ones, may reveal more interesting ideas about language and code.

Conceptual Languages

Before a compiler builds a program, it has to parse the code. Some languages focus on this step. Like unperformable Fluxus scores, they produce no functional programs at all, serving only to verify source code, turning a language into a system of inclusion or exclusion.

The compiler for the language *Unnecessary* (Keymaker, 2005), when run on any file at all, fails with an error message. An empty document, an image, a Word document, each is rejected as not Unnecessary. Only a file which can't be found succeeds in compilation—and it succeeds in creating an empty program, one which simply opens and closes. A single instruction—NOP (pronounced "no op" for "no operation") is the whole of the program. As Keymaker puts 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 null-quine but that was more of a by-product of the main idea. Fitting nonetheless! [10]

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, producing no output. Unnecessary is a language with no keywords, no input, that can only make empty programs.

Is it possible to have esolangs that go even further than Unneccesary, requiring no machine to run? Chris Pressey, creator of Befunge and creator of the mailing list where much of the early esolang discussions took place sees it this way:

> [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. [11]

A language is just a set of rules for symbols and their behavior. In a sense, they are an even more immaterial form than software; more like fields of possibility for potential software. Making this field narrow enough, we can get languages that have no usability, like Unnecessary. Even with Turing Complete languages like brainfuck, the practical use of the language is never the point: they are designed for the experience of digesting their rules. What makes an esolang interesting is that it rewards this investment of thought.

While these three strategies are distinct, they are often mixed together, as in Piet (and, to some degree, INTERCAL), and in hundreds of others which have been written since that time. Each of these three approaches makes language visible in making programmers type strange things, or think through irrational logic systems. At their best, they create a space for human impulses of communication to overflow constraints of logic.

References

1. Ben Olmstead. Interview. "Interview with Ben Olmstead," esoteric.codes (Daniel Temkin), November 3, 2014
2. Chris Pressey, "Esolang", approx. 2011, accessed Dec 12, 2014
3. Lev Bratishenko, "Technomasochism." Cabinet Magazine, issue 36 Winter 2009/2010
4. lolcats.org, accessed Dec 12, 2014
5. Wouter van Oortmerssen, Interview "Interview with Wouter van Oortmerssen," esoteric.codes (Daniel Temkin), upcoming
6. Daniel Temkin, "Brainfuck," Media-N Journal, Spring 2013
7. Geoff Cox and Alex McLean. Speaking Code: Coding as Aesthetic and Political Expression (Software Studies) (Cambridge: MIT Press, 2012), 22.
8. Ken Friendman, Owen Smith, Lauren Sawchyn. Fluxus Performance Workbook. (Performance Research e-Publications, 2002), 86.
9. Ais523. Interview. "Interview with ais523," esoteric.codes (Daniel Temkin), February 2011
10. Keymaker. Interview. "Interview with Keymaker," esoteric.codes (Daniel Temkin), January 2011
11. Chris Pressey. "The Aesthetics of Esolangs," June 2013 (accessed Dec 12, 2014) http://catseye.tc/node/The_Aesthetics_of_Esolangs

Bibliography

Geoff Cox and Alex McLean. Speaking Code: Coding as Aesthetic and Political Expression (Software Studies) (Cambridge: MIT Press, 2012)

Matthew Fuller, ed. Software Studies; A Lexicon (Cambridge: MIT Press, 2008)

Chris Pressey. "The Aesthetics of Esolangs," June 2013 (accessed Dec 12, 2014) http://catseye.tc/node/The_Aesthetics_of_Esolangs

Michael L. Scott. Programming Language Pragmatics, Third Edition (Burlington: Morgan Kauffman, 2009)

Daniel Temkin, esoteric.codes (blog), accessed December 18, 2014

The Esoteric File Archive, esolangs.org, accessed December 12, 2014

Archive of lang@esoteric.sange.fi mailing list, accessed December 12, 2014

Author Biography

Daniel Temkin (b. 1973) was recently awarded the 2014 Creative Capital / Warhol Foundation Arts Writers Grant for esoteric.codes, his exploration of programming languages as art. He has presented at conferences including Media Art Histories, GLI.TC/H, and hacker conferences such as NOTACON. His papers have been published in World Picture Journal, Media-N Journal and others, and have been taught at schools such as Bard College, Penn State, and NYU.

His work has been a critic's pick in ArtNews, the New York Times and the Boston Globe, shown at Mass MoCA, American History Museum, and galleries across North America and Europe.