2017.xCoAx.org

Lisbon
Computation
Communication
Aesthetics
& X

UNUSABLE FOR PROGRAMMING

DANIEL TEMKIN
daniel@danieltemkin.com

Independent, New York, USA

Abstract
Esolangs are programming languages designed for non-practical purposes, often as experiments, parodies, or experiential art pieces. A common goal of esolangs is to achieve Turing Completeness: the capability of implementing algorithms of any complexity. They just go about getting to Turing Completeness with an unusual and impractical set of commands, or unexpected ways of representing code. However, there is a much smaller class of esolangs that are entirely "Unusable for Programming." These languages explore the very boundary of what a programming language is; producing unstable programs, or for which no programs can be written at all. This is a look at such languages and how they function (or fail to).

Keywords
Esolangs
Programming languages
Code art
Oulipo
Tangible Computing

final draft

1. INTRODUCTION

Esolangs (for "esoteric programming languages") include languages built as thought experiments, jokes, parodies critical of language practices, and artefacts from imagined alternate computer histories. The esolang Unlambda, for example, is both an esolang masterpiece and typical of the genre. Written by David Madore in 1999, Unlambda gives us a version of functional programming taken to its most extreme: functions can only be applied to other functions, returning more functions, all of them unnamed. It's a cyberlinguistic puzzle intended as a challenge and an experiment at the extremes of a certain type of language design. The following is a program that calculates the Fibonacci sequence. The code is nearly opaque, with little indication of what it's doing or how it works. Even people who have spent time with the language need to piece together the program's structure and content.

```
```s``s``sii`ki
`k.*``s``s`ks
``s`k`s`ks``s``s`ks``s`k`s`kr``s`k`sikk
`k``s`ksk
```
(Madore 2003)

While Unlambda is bizarre, it is still falls firmly within most people's intuitive definition of a programming language: something along the lines of "a formal system for expressing algorithms to be carried out by a computer." Esolangs such as $tonePits challenge this. An esolang concept encoding computation into the movement of stones across a Manqala board, $tonePits reminds us that algorithms can be carried out in any procedural, repeatable, and exacting process. When art students learning about computation carry cardboard numbers across a room, this is not a metaphor for computation, but an actual implementation (however crudely) of an algorithm (as the pseudonymous Mahagugu puts it in the $tonePits wiki entry: Do you feel like a paleolithic caveman programmer now?). In fact, algorithms, named for a 9th century Persian mathematician, long predate computers. (Mahagugu 2015)

Turing Completeness (TC) is a computational class, a measure of complexity useful to understand what can be expressed in a language. According to the Church-Turing Thesis, any algorithm is computable in a TC language. Often sought as a goal for esolangs, TC status is the proof that the strange approach to logic taken by a language like Unlambda still allows a programmer to create algorithms as complex as any that can be expressed in a language like Java. (Turing 1937)

TC status has been proven in systems that were never intended to be used as programming languages. The Windows game *Minesweeper* is Turing complete, at least if played on an infinite board. (Kaye 2007) The human heart is TC. This research is done not because we want to encode programs in cardiac tissue, but because it proves that the heart is unpredictable, useful for studying heart arrhythmias. (Scarle 2009; Ostrovsky 2009)

The focus of this paper is not on the accidentally Turing Complete, but their opposite: languages that go further than Unlambda in unusability. These are the Unusable for Programming languages. To be unusable for programming (UFP), a language might be:

First Draft
```

1) Too unusual or strangely conceived to (yet) know their computational class;
2) A language computers cannot currently process and perhaps never will due to physical limitations of computational hardware;
3) Unreliable: the same code produces inconsistent programs;
4) Uncomputable: there is no way to compile or interpret a program written in the language due to its logical design (as opposed to limitations of the physical machine, which would fall under #2)

So why design languages that break from the most rudimentary task we expect languages to perform? Any answer, in part, will extend the challenge to programming norms that has been a part of esolanging throughout its history. Esolangs from the start have worked against received programming; working against the neutral tone of code, to the indifference of the compiler and the Western bias of programmatic tools. INTERCAL was the first esolang twice: when it was first created in 1973 and then when it was revived in 1990, several years before FALSE, the language that kicked off the esolanging movement. INTERCAL parodied the opaque coding style of its era, such as FORTRAN. (Raymond 2015) While esolangs expand on how programming languages function, UFP languages challenge the definition of what a language is. In this paper, we will look at examples of languages that fit each of the descriptions above, each breaking some expected element of programming language design.

As a creator of such languages myself, this paper will include some examples from my own practice alongside those of other esolangers. The research includes statements written by language designers about their work alongside original interviews for my esoteric.codes research project.

2. LANGUAGES OF UNKNOWN COMPUTATIONAL CLASS

Three Star Programmer is a language that fits the first category listed above; a language too unusual or strangely conceived to know its computational class. It is also a One Instruction Set Computer (OISC), meaning the language has only one command, one possible instruction. Imagine a version of BASIC which only had the command to print, except that OISC's command allows more complex behaviour than printing something to the screen repeatedly. To better understand how this type of language functions, it may help to first look at a simpler variation called Subleq.

Subleq's programs are lists of numbers. As Subleq has only one command, there is no reason to state it over and over in the program; instead, these numbers are the arguments passed to this one nameless command. It takes three arguments, so it reads three numbers from the program file at a time. The command takes the first number, let's call it A, and subtracts it from the next number, B. If B is zero or less after the subtraction, Subleq's execution jumps to the location specified in C. This number is a location within the program of the command to jump to. In Subleq, the memory addresses correspond to the source code of the program itself; if the program were to read "2 1 0", it would first subtract 2 from 1, changing the program in memory to "2 -1 0". Since -1 is less than zero, it will jump to the location listed in C, which is 0; back to the begin-

First Draft

ning of the program. This program is an endless loop. Amazingly, Subleq has been proven Turing Complete, so we cannot claim it as unusable for programming. But in what sense is Subleq an OISC? We can argue that Subleq's single command is actually a composite performing several manipulations under the hood (copy, subtract, conditional jump). However, it's treated in an atomic way; the programmer has to fire each of these commands in the same order every time. Other OISCs have tried to further simplify Subleq's behavior while maintaining Turing Completeness, such as BitBitJump which is similar in functionality but even more stripped down. (Mazonka 2011)

Three Star Programmer is not just an OISC but an experiment in programmatic indirection. In Subleq, each number has potentially two meanings: a literal value used in subtraction, and a memory address pointing elsewhere in the program; that is the value pointed to by the third argument (C). In Three Star Programmer, each number points to a location within the program, which points to another place, which points to another place; there are three levels of indirection. These are called pointers, and demarcated by stars in languages like C, hence the name Three Star Programmer. The name refers to a certain type of brilliant or arrogant programmer who would regularly do three star programming. In NASA's coding conventions, any level of indirection more than one is prohibited. (Holzmann 2006) Running this many levels of indirection breaks from the "clarity above all else" coding standard dating back to Dijkstra's *The Humble Programmer*. (Dijkstra 1972) Esolangs commonly challenge these programming norms, allowing for programmatic play and hackery.

When Three Level Programmer programs run, they dereferences each of these pointers, like a maze of arrows through memory, until finding the final location, then moving it one space one to the right of where it was. With each loop through the program, it prints the corresponding ASCII value to the screen. There is currently one working program in the language (with the source code "0 1 2"), which loops through the ASCII characters, printing each three times to the screen.

Three Star Programmer is not designed to be a UFP language, but because it is a new idea, and a very strange one, we do not yet know the power of its approach. It very well may be proven Turing Complete, or a simpler computational class, such as a Finite State Machine, allowing a subset of calculations to be carried out. The most common approach to prove Turing Completeness in such a language is to simulate a known TC language in it; if we were to recreate each of the commands of a simple TC language—the most common example is brainfuck, a very small language that fulfils Turing Completeness—we will know that Three Star Programmer is in fact very much usable for programming. This will be a difficult task for this odd language; as creator ais523 puts it, "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." (ais5232005)

3. LIMITATIONS OF THE MACHINE; TWO VARIATIONS ON SPOON

In 1998, Steve Goodwin created Spoon, a variation of the most notorious of esolangs, the eight-command language brainfuck. Brainfuck has just eight commands, each represented as a punctuation mark. It was created in 1993 as an experiment in minimalism; its compiler is just 256 bytes, a quarter of 1k. (Pressey

Final Draft

2015) Spoon encodes each of brainfuck's commands into binary using Huffman encoding, a compression method. The idea of Spoon was to represent brainfuck—already an extreme minimalist language—in the smallest programs possible. Brainfuck uses the command + to increment a value, - to decement, [to begin a loop,] to end one, and so on. Spoon uses 1 in the place of +, 000 in place of -, 00100 in the place of an opening bracket, etc. While Spoon uses just 0s and 1s, this is a reduction in the number of tokens (the alphabet of the language), not the number of commands (its lexicon), which remains at eight. It is simply a change of vocabulary to allow for smaller file sizes.

Goodwin mentions a suggested next step in development of the language, to try to go to a lexicon of just a single symbol, perhaps using all 1s. He says this is not possible, because we would still need to separate each group of 1s from each other; if 1 represents + and 11 means -, how do we know whether two consecutive 1s is a single + or two -s? (Spoon 1998)

However, this problem is overcome in another language, called (for some reason) Lenguage. Lenguage starts with its own binary representation of brainfuck, but then represents the number with only a single number. So if we were to pick "1" as our character, and our program were 110 (6 in decimal), we would represent it with six 1s: "111111". The fact that it's a 1 is immaterial, only the length of the string is considered. We could represent the same program with six stones, or a line six meters long.

If we were to use a Huffman-encoded version of Lenguage, essentially Spoon represented in a single value (to make the language a bit more compact) and a leading 1 to not drop the 0s which might start a program, we can write a Hello World program with the number 10849434748690940448822161841167224730538154893520896610033783669489699200976, or approximately nineteen quattuorvigintillion, 10 to the power of 76 (without the compacting, we need another 24 digits). To represent this with all 1s means using roughly the informational content of a one-solar-mass black hole. A hard drive capable of storing data at the atomic level holding this Hello, World program would be 333,000 times the size of the earth. While this is a practical concern, it is a hardware problem, not a theoretical limit; so whether our Spoon / Lenguage hybrid is truly Unusable for Programming is an open question. If we were to limit languages by hardware, many languages in use today would have been UFP forty years ago.

For a less ambiguously UFP language that goes beyond the limitations of the machine, there is Chris Pressey's 2007 variation of Spoon, called You are Reading the Name of this Esolang. Chris Pressey is a central figure in the esolang community; he created the highly influential esolang Befunge and started the mailing list where many of the early esolang discussions took place.

You are Reading the Name of this Esolang is Spoon with two additional symbols; opening and closing brackets. Code held in the brackets are read as Spoon programs and executed first. If they complete, they are translated to 1s and dropped back into the original sequence. If they do not halt (e.g. get stuck in an infinite loop), they are translated into 0s. The problem of course is that it is not so easy to determine if a program will ever halt. While, in some cases, an infinite loop 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 inserted it into the lexing step of the 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. If a sub-program were to take input from the user, the program itself may be valid or invalid within the language based on the user's behaviour, meaning that user input can make the entire program invalid.

4. UNRELIABLE LANGUAGES

The most dramatic of unreliable languages is perhaps Nora O'Marchu's cat++. To use the language, we write code that generates cats, who appear in a visual representation in front of a field of stars. While calculations can be performed via the cats, we don't have direct control over how they behave; we have to motivate them by creating other cats for them to interact with, giving them food, or otherwise interacting with them, all through code, and hoping they respond as we wish. Cat++ was designed as a system for live-coding visuals, drawing more from that tradition than esolanging, but it brings to code an interesting undecidability. The visuals are its main output. The cats are highly pixelated, with just enough detail to read as cat-like, as much from their catty movements as their static representation.

While we might see cat++ as a set of algorithms for performance, O'Marchu situates the work as a language: "Developing new uses for code as a medium for aesthetic or political expression allows for the dissemination and development of new understandings of the use and influence of code beyond technical domains." (Palop 2016) This is an esolang; a language designed for something other than practical coding.

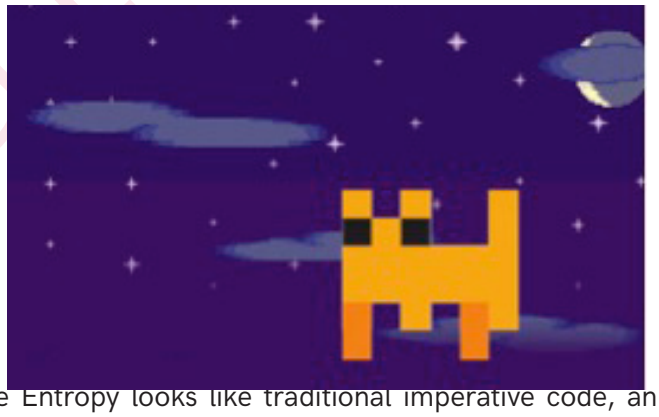

Fig. 1
Still from Nora O'Marchu's cat++

My language Entropy looks like traditional imperative code, an intentionally conservative mix of C and Pascal syntax. However, in Entropy, all data is like a natural resource with a time limit. Each time the data is accessed, there's a possibility that it will decay, go off by a small value, becoming more approximated over time. This is achieved by storing everything, even strings, as floating-point numbers or sequences of floats.

An Entropy Hello, World program run in a loop will look approximately like this:

```
Hellp, World!
Heklp, Wosld"
Hellq, Wntle"
Ielmq+ Voule!
```

Joseph Weizenbaum, creator of the Eliza chatbot, wrote about the compul-

Final Draft

sive aspect of programming; how the code is always buggy yet seems just a step away from being perfected:

> **Indeed, the compulsive programmer's excitement rises to its highest, most feverish pitch when he is on the trail of a most recalcitrant error, when everything ought to work but the computer nevertheless reproaches him by misbehaving in a number of mysterious, apparently unrelated ways. It is then that the system the programmer has himself created gives every evidence of having taken on a life of its own and, certainly, of having slipped from his control. (Weizenbaum 1976)**

Entropy cuts into this compulsive cycle by making the impossibility of achieving perfect code an explicit feature of the language itself, rather than an incidental (if inevitable) factor of programming. The programmer can express an idea to the user of her program quickly, before her data falls apart; but this is the most she can hope for. Another language of mine, Time Out, intervenes in the lexing of the language—the decoding of programmatic text—rather than its performance. In this way, it is similar to You Are Reading the Name of This Language. A Time Out program is a list of "time out" statements pausing execution for some number of milliseconds. These commands to pause are not the code of Time Out; the language is one step removed from this. It is actually the length of the pause between each line of code which serves as the actual text of the language. The language is written in pauses. This means that anything else which causes the interpreter to pause will affect what command is executed in Time Out.

Time Out is run in the browser; its interpreter is written in JavaScript; other JavaScript commands are allowed and will be executed by the JS interpreter directly, but not interact with the Time Out VM. Whether one invokes pauses with explicit "time out" commands or using JavaScript commands that take some time to run, it is the pauses themselves which are the tokens—the alphabet—of this language. To get a Time Out program to run as written, one must allow the program to run in the active browser tab. To click to another tab will deprioritize the activity of the Time Out program and possibly cause it to fire the wrong commands. Using another application will most likely affect the program as well; the only way to be sure it will run perfectly is to walk away from the machine or to sit and wait while it runs. While the units of speed corresponding to tokens in Time Out are configurable, to get it to run without error, a slow enough setting needs to be selected that the computer will not run it too slowly when resources are used elsewhere. On my machine, it takes seven minutes to run the Hello World program.

Of the three languages listed here as Unreliable, Time Out is provably Turing complete, if run in the correct context. It is possible to use Time Out in a predictable way which will accurately carry out algorithms; it just takes a lot of patience.

Most esolangs dramatize the gulf between programmer and machine and draw our attention to the act of programming itself. To make a language truly unpredictable might feel like a broken promise. Unlambda succeeds because its bizarre logic still allows us to write predictable code. These unreliable languages make the act of coding performative. This is most explicit with cat++ where this performance is graphical and designed for public display, but the other two languages likewise draw our attention to programming itself as an activity.

Final Draft

5. DEMATERIALIZED LANGUAGES

> The artist in algorithmic art creates an entire class of individual works. He or she is an artist insofar as she works in the realm of possibilities and potentials, not of realities and facts. The work of art in algorithmic art is the description of an infinity of possible works. They all share some common features that the mind can discern, even if the eye cannot see any similarities. The description is a sign of signs. (Nake 2010)

In algorithmic art, the material output serves as evidence of the art piece; the algorithm which generated them. While algorithms designed to output content of this kind can be described as fields of potential material manifestations, an esolang is similarly dematerialized and perhaps even more so: a field of potential algorithms that can be written in the language.

The list of rules for a language, the signifiers it recognizes and how those can be combined, is the closest we have to an embodiment of that language. Individual compilers that translate from code in that language into machine code might be evidence of the logic of the language, but individual compilers also might have quirks or bugs inaccurately enforcing the logic of that language. Furthermore, to use the compiler, we have to write code, which means referring to some kind of language description, in prose or in a formal notation such as EBNF. While most esolangs were created outside the context of art and so do not actively refer to dematerialization from the art-historical perspective, esolangs have their own history of conceptual experimentation around the native non-materiality of programming languages as a form; the fact that they are but lists of rules.

It perhaps begins with the Whitespace language, created in 2003. This language is written with only spaces, tabs, and returns, making programs appear blank. Whitespace, however, is perhaps less about materiality and more about encryption. While Whitespace programs can look like a blank page, they don't have to. From the original Whitespace website:

> Most modern programming languages do not consider white space characters (spaces, tabs and newlines) syntax, ignoring them, as if they weren't there. We consider this to be a gross injustice to these perfectly friendly members of the character set. Should they be ignored, just because they are invisible? Whitespace is a language that seeks to redress the balance. Any non whitespace characters are ignored; only spaces, tabs and newlines are considered syntax. (Brady and Morris, 2003)

Since all non-whitespace characters are ignored (the opposite of most languages), it's possible to use the spaces between words in a C program to write a functioning Whitespace program. The program file is now two programs: as read as C, and as read as Whitespace. This is known as a polyglot. While Whitespace has programs that look blank to us, to the machine, it is just a different a different character set; this is a play on the vocabulary of language; Whitespace is dematerialized only in a metaphorical sense; it is still a fully formed language, and Turing complete.

Things get more interesting when we go further down the line of dematerilization, toward languages that are missing key elements of that allow programs to

First Draft

be written as well. We can think of these as Conceptual languages, as they are impossible to materialize in the form of a sample program, a compiler, existing only as sets of rules.

According to esolangs.org, Unnecessary is a programming language "where the existence of a program file is considered an error." Keymaker, the creator of the language, describes it this way:

> **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. (Keymaker 2011)**

Unnecessary has no parsing step at all; it has an evaluator, which tests for just one condition: the existence of the file. If it succeeds in finding no file, its code generator spits out a file with a single instruction: NOP for no operation (this allows us to run it and see nothing happening). Unnecessary is the all-rejecting language. When you tell it to compile a program, it only succeeds when it can't find the source code, when it's given a bad path. So, like Keymaker says, the only programs that can exist for it are the ones that don't exist.

Keymaker mentions the null quine. A quine is a program where the source code and the output of the program it builds are identical. The null quine is a special quine that prints its own code but, since it has no code, it prints nothing. Unnecessary is like a language equivalent of the null quine itself. After looking at OISCs like Three Star Programmer earlier, we might think of Unnecesary as a Zero Instruction Set Computer.

Καλλίστῃ ("Kallisti", 2007, created by someone who calls himself "The Prophet Wizard of the Crayon Cake and the Seven Inch Bread") has instructions that are deliberately contradictory. Kallisti is a name drawn from Discordianism, an anarchic, Dadaist religion. The full set of Kallisti rules are:

- **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**

Where Unnecessary is clear and simple, Καλλίστῃ is dedicated to disorder and confusion. However, it includes pseudo-BNF notation, which says that Καλλίστῃ accepts anything and spits it back out unchanged. Where Unnecessary is the all-rejecting language, Καλλίστῃ is all-accepting. A C++ program, your resume, or a JPEG sitting on your desktop from last month's vacation: each one of these files is also a Καλλίστῃ program. But, because it's all-accepting, it can't favour any one piece of data over another, and can't make any decisions based on it. Instead, "computations arise from modifications to these anythings" (the anything of the source code and the anything of its output). This is done according to a syntax that "is very difficult for humans to understand."

Unnecessary has one or perhaps zero choices for a program, depending on one's perspective; Καλλίστῃ has many choices for programs, but all these choices are rendered equivalent. Most esolangs de-privilege the author (creator of

First Draft

the esolang) through their collaborative spirit; to create a language is ask esoprogrammers to explore it. Their discoveries describe the shape of the language, its potential algorithmically and expressively. Many esolangs start out with unknown computational class (such as Three Star Programmer) and are later proven through simulation of another language by programmers working with it. This group of conceptual language, in taking a step away from collaboration, ironically move back toward a singular vision, even if there are infinite (or zero) programs that are part of this system.

6. CONCLUSION

While esolangs like Unlambda question our approach to code, the UFP languages' interventions test the very definition of programming languages. The most common definitions of programming languages stress that 1) they are formal languages, lacking in the ambiguity of natural language semantics, and 2) that they are intended to send commands to the machine.

The lack of ambiguity of machine level semantics are hard to work around as the machine is not capable of interpreting a truly ambiguous message: "copy the value of memory cell 243 to register A" has only one meaning. Natural language is more slippery. In the unreliable languages (cat++, etc), this ambiguity slips into the language at the syntactic level: our communication through the language is filled with uncertainty (filtered through cats!), before the final translation down to machine instructions.

Most of the other UFP languages challenge the second part of the definition: their use in communicating with the machine, by abolishing the machine, requiring machines that can never exist, or constructing languages where little communication between person and machine is possible. While UFP languages can still be seen as far-out experiments in code, their rejection of the most basic principles of design allow them to go the furthest in questioning what it is we're constructing when we write code.

Final Draft

REFERENCES

ais523, 'Three Star Programmer' *esolangs.org wiki* (2015), accessed January 28, 2017

Edwin Brady and Chris Morris, 'Whitespace' (2003) archived June 23, 2015

Geroge Brecht, 'Events: A Heterospective' (Köln: Verlag der Buchhandlung Walther Konig)

Geoff Cox and Alex MacLean, 'Speaking Code' (Cambridge: MIT Press, 2009)

Edsger W. Dijkstra, 'The Humble Programmer' ACM Turing Lecture (1972)

Steve Goodwin, 'Spoon' (1998) archived February 28, 2014 https://web.archive.org/web/20140228003324/http://www.bluedust.dontexist.com/spoon

Gerard J. Holzmann, 'The Power of 10: Rules for Developing Safety-Critical Code' *IEEE Computer* (2006)

Richard Kaye, 'Infinite versions of minesweeper are Turing complete' (2007) http://web.mat.bham.ac.uk/R.W.Kaye/minesw/infmsw.pdf

Keymaker and Daniel Temkin, 'Interview with Keymaker' *esoteric.codes* (2011) http://esoteric.codes/post/84939008828/interview-with-keymaker

Mahagugu, '$tonePits' *esolangs.org wiki* Last Updated November 2015

Henry Matthews and Alastair Brotchie, editors, 'Oulipo Compendium' (London: Atlas Press, 1998) 177-178

Michael Matteas and Nick Montfort, 'A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics' *Proceedings of the 6th Digital Arts and Culture Conference* (2005) 144-153.

Oleg Mazonka, 'Bit Copying: The Ultimate Computational Simplicity' *Complex Systems Journal* Vol 19, N3 (2011)

Catherine Morris and Vincent Bonin, editors, 'Materializing Six Years: Lucy Lippard and the Emergence of Conceptual Art', MIT Press (2012)

Frieder Nake, 'PARAGRAPHS ON COMPUTER ART, PAST AND PRESENT' CAT 2010 London Conference (2010)

Igor Ostrovsky, 'Human Heart is a Turing machine, research on Xbox 360 shows. Wait, what?' (2009) http://igoro.com/archive/human-heart-is-a-turing-machine-research-on-xbox-360-shows-wait-what/

Benoit Palop, 'Cat++ is a Visual Live-Coding Language Based on File Behavior' *The Creators Project* (2016) http://thecreatorsproject.vice.com/en_uk/blog/cat-visual-coding-language

Chris Pressey and Daniel Temkin, 'Interview with Chris Pressey' *esoteric.codes* (2015) http://esoteric.codes/post/118780138572/interview-with-chris-pressey

Chris Pressey, 'You are Reading the Name of this Esolang' *Cat's Eye Technology* (2007) accessed January 31, 2017 http://catseye.tc/node/You_are_Reading_the_Name_of_this_Esolang

Eric S Raymond and Daniel Temkin, 'Interview with Eric S. Raymond' esoteric.codes (2015) http://esoteric.codes/post/130618094278/interview-with-eric-s-raymond

Simon Scarle, 'Implications of the Turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: Cardiac arrhythmias, re-entry and the Halting problem', *Computational Biology and Chemistry* Volume 33, Issue 4 (2009)

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

Seven Inch Bread, 'Καλλίστῃ' *esolangs.org wiki* (2007) accessed January 28, 2017

Alan Turing, 'On computable numbers, with an application to the Entscheidungsproblem' *Proceedings of the London Mathematical Society* Series 2, Volume 42 (1937)

Joseph Weizenbaum, 'Computer Power and Human Reason: From Judgment to Calculation' (London: W H Freeman & Co, 1976)

Final Draft