DHQ: Digital Humanities Quarterly
2023
Volume 17 Number 2

The Less Humble Programmer

Daniel Temkin <d_at_danieltemkin_dot_com>, Bard College

Abstract

Esoteric programming languages (esolangs) break from the norms of language design by explicitly refusing practicality and clarity. While some go even further and make code impossible to write (e.g. Unnecessary), others (e.g. Malboge) retains the ability to express functional and reliable code, despite the seeming disorder of the language. To understand the conversation these languages are having, we must look at how they challenge or re-affirm wider ideas in programming culture and in how computer science is taught: specifically the sometimes-contradictory aesthetics of Humbleness and Computational Idealism.

The “esoteric” class of programming languages break from practicality. An esolang (a portmanteau of “esoteric” and “language”) is not designed to solve practical issues, even esoteric ones. Instead, esolangs ask the programmer to puzzle through counterintuitive rules that challenge them or invite reflection on the act of programming. 1

In their refusal of practical use, esolangs are primarily cultural objects. Apart from this, they can seem at first glance to have little else in common. Among the thousands of esolangs are programmer in-jokes, critique of computational culture, experiments with logic, and experiential art pieces. Esolanging is a medium: programming language design as a creative practice. I would no sooner propose an all-encompassing aesthetic theory of esolanging than I would of painting. But it is also a young medium, first explored by a movement that gained traction in the 1990s and 2000s. This movement (which has no name other than esolangs) has a common set of concerns setting the tone for the languages that follow. 2

The aesthetics of this movement often parallel those of conceptual art and digital art more broadly: esolangs are open-ended systems, natively collaborative, and distanced from any single materialized form. However, the early esolangers did not think of their work in an art context and were not concerned with art history. They were reacting to and building on the aesthetics of commercial coding and the often unstated values of computer science. These disciplines, which are sometimes at odds with each other, are both driven by a pragmatism that esolangs actively eschew. In rejecting practicality, esolangs carve out their own aesthetic and make clear the contradictory factors at work in mainstream code aesthetics. 3

More specifically, esolangs play off a tension that goes back to the beginning of computing. The dominant approach to code is set out explicitly in Edsger W. Dijkstra's 1972 paper “The Humble Programmer.” Dijkstra calls for clarity and neutrality of style, in the service of “professionalizing” programming as a discipline. This aids collaboration between developers, allowing code to become modularized widgets that can more easily interact, for both corporate and free software enterprises to function at scale [Dijkstra 1972]. But in so doing, Dijkstra had to push back against the “free-wheeling” hacker styles of the 1950s, which prioritized optimization and clever hacks, necessary when clock cycles were at a premium. That spirit has not been quashed even after many decades of professional programming, but instead sublimated into hacker activities outside of commercial coding that allow for personal style and virtuosity, such as code golf and obfuscated coding. While these two examples have elements of competition, esolangs are instead collaborative and experimental, a space to collectively explore strange and absurd logic. 4

The neutrality Dijkstra prescribes not only suppresses l33t hacker skills, but also elegance, a conflicted idea in 5

mainstream code aesthetics. Elegance is at once a near-mythic objective smuggled into computer science from mathematical Platonism but downplayed in day-to-day coding, where it conflicts with practical considerations like timelines, compatibility, and the now-unassailable neutrality of style. Esolangs, associated more with disorder, might seem like a strange place to seek elegance. However, the pursuit of elegance comes back repeatedly in esolangs, often to vindicate ones' virtuosity. Esolangs provide a way of engaging with this idea from mainstream code aesthetics that paradoxically is given little space in production code.

This paper explores this aesthetic of esolangs through its history, looking closely at examples from early in esolang practice. Late in the paper, we will look at esolangs that break from this style or apply it in new ways, pointing to future possibilities for the medium. I approach this both as a researcher, who has interviewed dozens of esolangers over ten years for esoteric.codes and also as a creator of such languages. 6

Clarity and Humbleness

Dijkstra's 1972 Turing Lecture Award "The Humble Programmer" begins with personal history. In 1957, applying for a marriage license, Dijkstra was asked to state his profession. At the time, "computer scientist" and "programmer" were not yet recognized by the Dutch government. Years earlier as a PhD student, he had approached his then-advisor, the great computer scientist and co-creator of ALGOL, Adriaan van Wijngaarden, with his frustrations that programming was not yet "an intellectually respectable discipline;" van Wijngaarden encouraged him to help shape it into one [Dijkstra 1972]. 7

These anecdotes show what "The Humble Programmer" is up to. Not a technical paper, it explains how a programmer should organize their thoughts into code in a way that could finally bring respect and "competence" to the activity. 8

It builds on a letter that competes with this one as Dijkstra's best-known piece of writing, 1968's "Go To Statement Considered Harmful" [Dijkstra 1968]. Gotos are unconditional jumps that move the activity of the program to any arbitrary line of code without regard for structure or scope. They are easy to abuse; by their very nature, they skip across the surface of the program, bypassing containment meant to group related activity together. He argued that we "wise programmers are aware of our own limitations" and so should avoid the temptation of the command, and instead embrace the "structured programming" of blocks and subroutines fully, even when this is less efficient [Dijkstra 1968]. The technology had reached a point that some efficiency could be given up in order to build more larger programs that remain readable. This idea is now the unassailable model of coding and baked into the design of most languages: modern languages have no goto statement (e.g., JavaScript) or constrain them to work only within a block of code (e.g., C#). 9

"The Humble Programmer" takes the simple prescription of banning gotos, a push toward complete adoption of structured programming, and builds an entire design philosophy around it. Clarity is the goal at every level of the program: from the high level of program structure down to clearly written individual lines of code. Whether one learned to code as a computer science student, in a creative computing program, on the job or on one's own, the style of code Dijkstra calls for is the norm; "good code" shortens the cognitive distance between the text and its ultimate performance. This begs the question of who Dijkstra is talking to; who needed to receive this message. If "clever tricks" was not enough of a hint, Dijkstra spells it out in describing programmers' one-liners: 10

> [O]ne programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question "Can you code this in less symbols?" — as if this were of any conceptual relevance! — or he just asks "Guess what it does!". From this observation we must conclude that this language as a tool is an open invitation for clever tricks … I am sorry, but I must regard this as one of the most damning things that can be said about a programming language. [Dijkstra 1972]

The "New Programming" Dijkstra advocates is very much at odds with what programming looked like in the previous era. John Backus, director of the team that developed FORTRAN beginning in 1954, describes what coding looked like before structured programming was the norm:

> Programming in the early 1950s was really fun. Much of its pleasure resulted from the absurd difficulties that "automatic calculators" created for their would-be users and the challenge this presented. He had to fit his program and data into a tiny store, and overcome bizarre difficulties in getting information in and out of it, all while using a limited and often peculiar set of instructions. He had to employ every trick he could think of to make a program run at a speed that would justify the large cost of running it. [Backus 1980]

"Limited and peculiar set of instructions" also characterizes many esolangs. Backus describes the downside of this era:

> Just as freewheeling westeners developed a chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals. This feeling is noted in an article by J. H. Brown and John W. Carr, 111, in the 1954 ONR symposium: ". . . many 'professional' machine users 'strongly opposed the use of decimal numbers' . . .. To this group, the process of machine instruction was one that could not be turned over to the uninitiated." [Backus 1980]

Machine instructions are written in the 1s and 0s of pure binary, and often represented in code as hex (base 16), which maps more easily to binary than decimal numbers. Neither binary nor hex are especially difficult to learn with practice, but their use essentially locks out people who know only decimal, those who have not spent significant time thinking and calculating with machinic systems of that era.

11 In her "Sorcery and Source Codes," Wendy Chun delves into this priesthood of early machine coders guarding what they saw as "esoteric knowledge" and the way their legacy still lives in tension with moving code from a craft to an industrialized practice that Dijkstra's humble approach made possible:

> Much disciplinary effort has been required to make source code readable as the source. Structured programming... sought to rein in "goto crazy" programmers and self-modifying code. A response to the much-discussed "software crisis" of the late 1960s, its goal was to move programming from a craft to a standardized industrial practice. [Chun 2013]

Industrialization involved a level of abstraction: creating units of code as building blocks that could more easily function across different machines and contexts. Through it, a deskilling turned "programmers into users," although as Chun describes, some programmers craved a space that allowed the free-wheeling and unencumbered "really fun" programming of the 1950s that was no longer considered acceptable. Chun explicitly mentions "self-modifying code," [Chun 2013] where programs edit their own source code as they run. It makes code harder to read and maintain, as the instructions in the source program are not necessarily the instructions that will be executed. While it has uses in machine learning and DRM software, it is generally discouraged in modern programming, but common in esolangs.

12 Spaces for programmers to program outside the humble style have emerged repeatedly since the 1970s. Nick Montfort and Michael Matteas's "A Box Darkly" describes obfuscated coding contests, where programmers compete to create the hardest-to-read programs. The existence of these contests "throws a wrench into the simplified theory of coding" that clarity is the primary aim [Montfort and Matteas 2005]. Often obfuscated code contests use constraints, such as a maximum program size, to justify obfuscation: packing maximal functionality into a short program can't be easily done with clear code. These constraints mimic the conditions of early computing that led to the esoteric nature of code that Dijkstra objected to. Demos, programs that show technical skill through tiny audio-visual programs, also have contests that reward based on the brevity and technical brilliance of packed binaries, associated with the 80s and 90s of the Amiga era.

13 In code golf, a recreational form of programming contest, programers compete to accomplish a common task in the fewest number of characters, often with explicit additional constraints. A recent example from the Code Golf section of the popular programming resource Stack Exchange, is to write a program that prints the letter "a" without using any "a" in the source code, but also avoiding "X" and digits like "0", "1", or "4", among others, which can be used to build the

numeric equivalent of “a”. One competing program, below, finds the truth value of 3 < 3, which is False, and then grabs the second letter of the resulting string “False.” We would ordinarily grab the second letter using the number 1 (Python is zero-based). However, since 1 is a banned symbol, it works backwards, grabbing the third from the back:

```
print`3<3`[~3]
```

Example 1. “The Letter A without A.” Codegolf.Stackexchange.Com. https://codegolf.stackexchange.com/questions/90349/the-letter-a-without-a.

This program shows how even a very short program can show great proficiency, while becoming nearly unreadable to most programmers. [Stackexchange 2016]

For each of these practices, the abandonment of “good code” toward an anti-Dijkstrean aesthetic recalls some of the conditions of early computing, but they are not retro forms. Rarely are programmers writing 1950s machine code in contests. Instead, they create conditions where esoteric computing skills become necessary, freeing the programmer to use every clever trick they can come up with. 14

Virtuosity and Esoteric Knowledge

The constraints in esolangs are created by the grammar of the language, rather than explicitly set like in rules like in code golf. There is also no set objective; the programmer (referred to as an esoprogrammer) writes code of their own devising within an esolang. They might discover something new about the language, create a program of great complexity to show what is possible, or extend the idea of the language to places the esolanger had not foreseen as possible. This is a decidedly collaborative form. 15

Looking at the early esolangs reveals how this break from Dijkstra’s Humbleness happened in the earliest languages. But before the esolangs, there were the beloved oddball languages like Forth and SNOBOL, which showed that idiosyncratic programmatic paradigms are possible. The Forth language encourages heavy use of a data structure called a stack, where values (e.g., strings and numbers) are stored and manipulated without ever assigning them to variables or otherwise naming them. To become a great Forth programmer requires adopting a very different kind of thinking, visualizing the state of this stack, the location of various values that move up and down through it. This was inspirational to esolangers, who saw just how deep one can go into a very different model of thinking. However, Forth is not an esolang, despite its quirks. Its stack-based model works very well in heavily constrained machines such as microcontrollers, and Forth found wide use in the space program [FORTH, Inc. n.d.]. 16

Forth helped inspire the language which kicked off the esolang movement, FALSE, in 1993. Its creator, Wooter van Oortermsen, had already created a popular language for the Amiga, called Amiga-E, and would go on to get a PhD in programming language research. However, at the time, he was a hobbyist in language design. He created FALSE — named for his favorite truth value and capitalized ala FORTRAN — to learn how to write a compiler in assembly code [van Oortmerssen and Temkin 2015]. 17

FALSE was not designed as an esolang, as the concept did not yet exist. Van Oromerssen could not foresee the reaction the language would ultimately receive or its long legacy — or really that anyone would take an interest in his language at all. He created it as an intellectual challenge for himself: to write as small a language as possible — ultimately creating a compiler of only 1k — but pack as much functionality as possible into those 1024 bytes. It is expansive within its small size: it has named variables (26, one for each letter), and sophisticated features like lambdas, which later “minimal” esolangs would not bother with. 18

Its esoteric qualities come from the measures van Oortmerssen used to get it down to 1k. First, the single-letter commands, chosen because of the ease of parsing, lead to programs as dense thickets of information, along with the somewhat exotic stack-based model of computation borrowed from Forth. The result was a language very challenging to read (called a “write-only language”). Here is a prime number generator in FALSE: 19

```
99 9[1-$][$@$@$@$@/*=[1-$$[%1-$@]?0[$.' ,]?]?]#
```

Example 2. A prime-number generator in FALSE, Wouter van Oortmerssen.

This style of esolang, holding great algorithmic potential but with little regard for readability, became known as a Turing Tarpit. Tarpits have Turing Completeness, sufficient complexity to represent any algorithm expressible on a conventional computers. This term came from Alan Perlis's warning of tarpit languages "in which everything is possible but nothing of interest is easy." Their extreme minimalism make obfuscation nearly impossible to avoid [Pedis 1981]. 20

FALSE had an immediate response: two answer languages the same year, each expanding on one aspect of FALSE's design, and each created for the Amiga. The first is brainfuck, which has become the prototypical example of the Turing Tarpit. Where van Oortmerssen wanted to pack as much functionality as possible into his minimal language, Urban Müller's brainfuck removes everything except the bare minimum necessary to reach Turing Completeness [Müller 2017]. Where FALSE is 1k, brainfuck is less than a quarter of that size. And where FALSE has strange, difficult-to-read code, brainfuck is so minimal that there can be little confusion about what each of the symbols mean, and yet in use, so many of the symbols are necessary to accomplish anything, that they inevitably create programs that obscure their function. 21

In brainfuck, one moves back and forth in a line of memory cells. There is no way of directly writing an integer in brainfuck, so if we need the number 46, the simplest way to get there is to write a program with 46 +s in a row (each one adding a single number): 22

```
++++++++++++++++++++++++++++++++++++++++++++++.
```

Example 3. Assigning 46 to a memory cell in brainfuck.

The period at the end prints the value in ASCII; 46 happens to be a period, so that is printed to the screen. However, this is not an especially readable brainfuck program; 46 +s can easy be mistaken for 45, which would print a minus sign instead. There are countless ways to get to the number 46 in brainfuck, many more interesting than the above program; we'll look at more examples in the next section. In this language, even assigning an integer, one of the most basic commands, is not offered directly and instead requires creative choices by the programmer. It annihilates the possibility of neutral code. 23

The other language created in 1993 in response to FALSE, is Befunge. Chris Pressey had been experimenting with the idea of a language that drew on the goto, Dijkstra's forbidden command, but to make it *more so*: "what if you had BASIC, but instead of having line numbers, you drew an arrow to the line you wanted to GOTO?" FALSE's single-letter commands gave him a way forward. In a monospaced font, letters align not only horizontally but vertically as well; with single-letter commands like FALSE's, Pressey could now create a program that flowed up and down, right-to-left, and back and forth, using the symbols ^ < > v to change direction. Other symbols allowed for conditions: if a statement is true, it would go one direction, and if false, the opposite. Befunge abolishes the concept of lines of code. 24

While Befunge is similar to FALSE in its brevity of text and its heavy use of the stack, it is not a Turing Tarpit. Its complexity is lexical, front-loaded. Befunge code can run off the screen in one direction, to emerge on the other side, a program as labyrinth where conditional branches are alternate paths through the text. In Befunge, the typical Hello World program has "Hello, World!" written backward in the string, as each letter is added to the stack and is printed in reverse order: "!dlroW ,olleH". However, this is assuming it is currently running left-to-right, and being Befunge, the opposite might be true. Or perhaps the "Hello," and "World!" cross at the "o", one flowing vertically and the other horizontally. There are many arbitrary approaches to Befunge, encouraging creative exploration of its rules. 25

```
                              060p070
p'O80v
                             pb2*90p4$4>
$4$>v>
                            v4$>4$>4$>4$>#
ARGH>!
                           <{[BEFUNGE_97]}>
FUNGE!
                           ##:-:##   #####*          4$*>4$      >060p>    60g80g -!#v_
60g1+     60p60v
                           #vOOGAH               **>4$>^!!eg    nufeB^    $4$4$4 $4<v#
<<v-*2a::   v7-1g<
                           #>70g>90g-!          #@_^Befunge!!   123456    123456
VvVv!#!>Weird!   >0ggv*
                            ^$4$4p07+1g07      ,a<$4<   <$4$4<  <$4$4<    <$4$4< <<#
<*-=-=-=-=-*  -=-=v*
                             ::48*-#v_>,4$>    4$4$4     $4$4$  4$4$4$    4$4$4$
4$^*!*   XXXXXX   XXX>
                               BOINK>$60g1-7  0g+d2*     %'A+,1 $1$1$1    $1$1$1 $>^<$
HAR!!!  8888
                                  Befunge_is  such_a     pretty langua    ge,_is n't_i
t?_It_  8888
                                     looks_so much_l     ike_li ne_noi    se_and it's_
STILL_  '88'
                           Turing-    Complet e!_Cam     ouflag e_your    code!! Confu
se_the
                           hell_out  of_every one_re     ading_ your_co  de._Oh, AND_y
ou.:-) ,o88o.
                           Once_this_thing_i   s_code   d,_rea  ding_it_back_ver
ges_on   the_imp 888888
                            ossible._Obfusc     ate_the_obfus    cated!_Befunge_
debuggers_are__  888888
                             your_friends!       By:_Alexios     Chouchou las... X-X-
X-X-X-X-X!   888888
                                -=*##*=-           \*****/         9797*  -=97=- !@-*=
*****     '"88P'
                                                                              *!@-*
                                                                              =*!@-
                                                                              -=*!@
                                                                              @-=*!
```

Example 4. The full code of the program “Soup!”. If the code does not appear to spell out the word “Soup!” as ASCII art in your browser, it can be viewed at this website: https://www.bedroomlan.org/coding/soup/

The above Soup! program by Alexios Chouchou prints the word Soup, similarly rendered as its output. Most of the text in this program is never run as code, as the instruction pointer flows around it (e.g., see the text beginning with “Befunge

is such a pretty language, isn't it?") [Chouchou 2011].

Pressey's Befunge mailing list would foster the first community devoted to esoteric language design practices. Pressey also first coined the term "esoteric language" to mean hyper-obscure. While it was not coined in reference to early computing, the connection proved apt [Pressey and Temkin 2015]. 26

Not all esolangs use single-letter commands or adopt a puzzle mentality to coding. Multicoding Languages explicitly hold multiple meanings, building on the original use of the computer, that of an encryption/decryption device. Programmers create code that functions simultaneously in both, giving a new avenue for personal expression [Montfort and Matteas 2005]. The Piet language, created by David Morgan-Mar, uses images as code, while Velato, a language I wrote, uses music in the form of MIDI files. To write a working piece of code, the resulting music often sounds computer-generated; the implicit challenge of the language is to write something that works as code and sounds musical rather than random. 27

IRISH CREAM DESSERT SQUARES.
This recipe creates sapid dessert squares perfect for party, pleasure, and it also displays the square of any number.

INGREDIENTS.
1 package yellow cake mix
3 beaten eggs
12 tablespoons Irish Cream Liqueur
5 tablespoons oil
1 can of cream cheese vanilla frosting
61 white chocolate chips
Cooking time: 30 minutes.
Preheat oven to 180 degrees Celsius gas mark 4.

METHOD.
Take beaten eggs from refrigerator. Put beaten eggs into mixing bowl. Combine package of yellow cake mix into mixing bowl. Combine beaten eggs. Mix the mixing bowl well. Put oil into mixing bowl. Fold Irish Cream Liqueur into mixing bowl. Mix the mixing bowl well. Pour contents of mixing bowl into the baking dish. Put can of cream cheese vanilla frosting into 2nd mixing bowl. Combine white chocolate chips into 2nd mixing bowl. Liquefy contents of the 2nd mixing bowl. Pour contents of the 2nd mixing bowl into the baking dish.

Serves 15.

Example 5. IRISH CREAM DESSERT SQUARES, Author unknown.

Chef (also created by Morgan-Mar) asks programmers to code using cooking recipes. Designing a recipe that leads to an edible outcome that also works as code is a challenge. Early Chef programs called for hundreds of pounds of flour in order to have the correct numbers to print a message to the screen. It took years before programmers engaged with the creative possibility offered by these multicoding languages and use them as constraint systems that shaped the "non-code" aspect of their works. This approach to Chef was popularized by Ian Bogost, who has for years assigned his creative coding students to write Chef recipes, with a requirement to attempt an edible result [Temkin 2020]. 28

Chef's central metaphor is algorithm as recipe. A recipe is a continuous process broken down into discrete steps which, when repeated with the same starting state, will produce the same result. In other words, a recipe is algorithmic. Chef, in making this metaphor literal, maps baking activities to computation. This mapping is arbitrary, designed by the esolanger, and creates the aesthetic of the language. Variables are ingredients, but taking an ingredient from the fridge reads from STDIN; if a variable will not be populated from user input, it must already be on the table. Only certain types of programs can be written easily — for instance, strings are a problem as they involve higher numbers than one can get into a recipe, so mathematical programs are preferred. Also, only certain types of cooking works well, mainly baking with many mixing bowls, a necessary instrument to handle data in the program. While elegant Chef programs are 29

possible, they are not the point; the programmer who writes very slow code that's still edible will not have their algorithmic approach criticized, as the challenge is already so difficult [Bogost 2016]. Instead, Chef takes the concept of personal expression in the text of code, as seen in Befunge, into a system of signification with multiple layers of meaning. It is a set of constraints that each programmer can find their own way through, creating a recipe that is not merely about technical brilliance alone, but a challenging creative exercise within the constraints of the language.

Computational Idealism

Esolangs can seem to embrace disorder, with their unconventional, often unreadable programs, especially to the uninitiated. After all, challenging conventions of language design is at the heart of esolanging. However, there is often order within that disorder, drawing from a quality of classical programming which is too often understated, the seemingly ineffable quality of *elegance*. 30

Elegance is perhaps best associated with Donald Knuth, who calls for an aesthetic of code in both his book *Literate Programming* (1992) and the ongoing epic *The Art of Computer Programming*, which has been growing, volume by volume, for fifty years (volume 1 appeared in 1968, volume 4b in 2022). Matthew Fuller, in his essay "Elegance," sums up the concept as Knuth presents it, in these criteria: "the leanness of the code; the clarity with which the problem is defined; spareness of use of resources such as time and processor cycles… Such a definition of elegance shares a common vocabulary with design and engineering, where, in order to achieve elegance, use of materials should be the barest and cleverest." [Fuller 2013] 31

Knuth connects his use of the term art with the Greek word *techne*, the root of both technology and technique: art as applicative, as opposed to science. Despite his wide use of terms like art, aesthetic, and literary, he is not interested in developing an aesthetic theory or in questioning his own underlying assumptions. He speaks frequently of beauty in code, but more in terms of great craftspersonship than in personal expression. He also expects that beauty in code will be self-evident. 32

> Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write grand programs, noble programs, truly magnificent ones! [Knuth 1998]

Beatrice Fazi, in her philosophical treatise *Contingent Computation*, gives a name for the aesthetic Knuth embracing: "computational idealism." [Fazi 2018] For Fazi, it draws not only from design and engineering, but from a Platonism associated with mathematics. She describes it as "the classical age's concern with the supremacy of simplicity over complexity, of order over chaos, and of unity over parts." [Fazi 2018] The most beautiful equations are those that find unexpected and powerful truths concisely and in simple ratios [Fazi 2018].

Fazi illustrates computational idealism with a surprising choice, a malicious program associated more with mayhem and disorder. The program is Jaromil's forkbomb, a shell script written in all punctuation that will crash a Mac built in 2021 as well as any machine from twenty years ago. It is also the most-tattooed program of all time [Roio 2000]: 33

```
:(){ :|:& };:
```

Example 6. ASCII Shell Forkbomb by Jaromil, 2002

The forkbomb is efficient and single-minded: a "small program that does exactly what it's supposed to do." [Fazi 2018] Fazi sees the program as an expression of the programmer's proficiency in the art of programming: "The pace of the fork bomb is relentless, inexorable, inescapable." [Fazi 2018] The program has none of the clarity Knuth calls for, but is the "barest and cleverest," the minimal answer to the problem, in only thirteen characters. 34

It is inescapable because it is purely algorithmic: there is no input or output, no interaction with other systems, nothing to interfere with its progress. A single algorithm is carried out repeatedly in this program. Its primary purpose, to halt the machine, can not be found in the text of its code; it is a side-effect of this relentlessness. 35

The algorithm is at the heart of Knuth's *The Art of Computer Programming*. It begins with a chapter defining the concept and with this message: 36

> "Information processing" is too broad a designation for the material I am considering, and "programming techniques" is too narrow. Therefore I wish to propose analysis of algorithms as an appropriate name for the subject matter covered in these books. [Knuth 1997]

Warren Sack, in his *The Software Arts*, does not see this elevation of the algorithm as accidental. The algorithm, as Knuth defines it, is exactly the aspect of computation that can be described mathematically; placing it at the center of computing allows a mathematical aesthetic to remain dominant ("Only some software can be rendered as mathematics — the software that implements algorithms!" [Sack 2019]). We see evidence of Knuth's sidelining of non-algorithmic thinking in his chapter "Input and Output," one of the very few that focuses on a process that is not primarily algorithmic. It begins with a warning that "many computer users feel that input and output are not actually part of 'real' programming." [Sack 2019] Sack sums up Knuth's points of what marks an algorithm: finiteness ("an algorithm must always terminate after a set number of steps,") definiteness ("each step of an algorithm must be precisely defined,") effectiveness ("all of the operations in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using pencil and paper") [Sack 2019]. This sidelines much the computing we actually do: always-up systems like web servers, trading platforms, etc., are not algorithmic in this sense, nor is code built around communication, which makes up the majority of code running today [Sack 2019]. 37

The purely algorithmic language, allowing for no input or output or interaction with other systems, is a common sign of an esolang; stripping away any utility marks the language as meant for something else. Chris Pressey's SMETANA (Self-Modifying Extremely Tiny AutomatoN Application, 1994) is a language that essentially only offers goto [Pressey n.d.]. Here is a typical program, an infinite loop: 38

Step 1. Swap step 1 with step 2.

Step 2. Go to step 2.

Step 3. Go to step 1.

Example 7. SMETANA Example Program by Chris Pressey, 1994

Line 1 swaps 1 and 2. By the time we get to 2, it has the content of 1, and so it swaps them back. Line 3 sends us back to the beginning. These two commands, swapping and jumping, are the only commands available in SMETANA. It reflects a very common strategy to esolang-building: to put forward an extremely strange set of rules to then discover what is possible to accomplish in the language. It relies on self-modification, one of the features of early computing that is frowned on in Humble programming. While it might seem that nothing can be accomplished in a SMETANA program, that is not entirely true. Comparing the SMETANA's beginning state and end state and interpreting the result gives a way to understand its computation. In fact, SMETANA is proven to be a Finite State Machine, more restricted than a Turing Complete language, but capable of simple logic such as matching a string. It is surprising that anything at all can be accomplished within this language and perhaps that is the point of it [Hopcroft, Motwani, and Ullman 2001]. 39

Brainfuck is a language highly associated with chaos and disorder, where even short programs are nearly unreadable because of the bare simplicity of its commands and the great many that are necessary to accomplish anything of complexity. Yet if we look at the examples of brainfuck programs, they are not sloppy and verbose; to many esoprogrammers, brainfuck is a complex machine like those of early computing, in which they can recreate order. 40

As mentioned in the previous section, brainfuck can't directly represent the number 46. Here are examples of constructing the number 46 taken from the esolangs wiki: 41

```
+++++[>+++++++++<-]>+ (21, 2) non-wrapping
-[>+<++++++]>+++ (17, 2) wrapping
>-[[ - - >]<<-]>- (16, 3) wrapping
+[-[ - <]>> - ]<- (15, 4) wrapping
-[+>+[+<]>+]> (13, 5) wrapping
```

Example 8. (“Brainfuck Constants” n.d.)

While there are many other options for arriving at 46, the ones publicly shared are the most elegant, those that minimize some aspect of the algorithm. Each is followed by two numbers: the first is the number of characters used to write the code, the second is the number of memory cells it uses to perform. 42

The first example is the most straightforward. We add five to a memory cell by using the + operator five times. We use the > operator to move to the right, and add nine to it with nine +s. The square brackets loop us back and forth between the two; we subtract from the left cell repeatedly, each time adding nine to the right, effectively multiplying one number by the other. This is the simplest form of multiplication in brainfuck and is easy to spot with experience in the language. Writing this way means going back to first principles, like building addition and multiplication in formal theory from the successor function. 43

The following examples are marked with “wrapping,” meaning they pass zero. One feature of brainfuck, most likely deriving only from the minimalism of its compiler, is that subtracting 1 from 0 gives the maximum value for that memory cell, 255. Four of the five examples wrap past zero, counting up or down, sometimes many times, in order to eventually arrive at 46. The last example, with a loop within a loop, is nearly unreadable and uses five memory cells, more than the others, but it is the shortest way to express code that arrives at our target number in just 13 characters. 44

Müller creates brainfuck, a language where programs take on complexity far faster than in ordinary language, and esoprogrammers answer by enacting the most elegant possible “solutions” to the language. This challenge and response gives a way for programmers to show their abilities; we recognize their brilliance through this quality of elegance. The seeming chaos of brainfuck is perhaps not a desire to tear down classical design but instead an invitation to recreate it in hostile, esoteric territory. Furthermore, in the case of brainfuck, that hostile territory has clear parallels to the constraints of early computing. 45

Perhaps the most dramatic example of this tension is in a language created by a mathematician, John Conway’s FRACTRAN. FRACTRAN is a purely mathematical language with no obvious parallel in physical machines of today or in early machines. Conway would scrawl his FRACTRAN program called PRIMEGAME on blackboards from memory and walk students through its intricacies. The program is a series of fractions and a single number to represent its starting state, in this case, 2. That state is compared to each fraction in order; if it can be divided evenly by the denominator, it is multiplied by the fraction. Essentially, the program stores state as a factored version of the number, with each factor a variable and its exponent the current value. Then it goes back to the beginning of the sequence. The PRIMEGAME program, when run, produces 2^2, 2^3, 2^5, 2^7, 2^11, and so on, producing each of the prime numbers in order, as powers of 2. 46

FRACTRAN is a programming language run more often by human hands than by machine. Programs that require more than a few variables end up unwieldy. A FRACTRAN interpreter written in FRACTRAN itself uses fractions like 829688452441224767515935732 1 / 5453349840514519345169425873 to handle the great many variables at play. Wildly unintuitive, the language rests on a simple, elegant concept, and, in the hands of a genius like Conway, a complex program can be expressed in thirteen fractions [Beder n.d.]. 47

Esolanging as Idea Art

The dichotomy of Humbleness and Expressiveness dominates esolang aesthetics, with the emphasis on personal style, virtuosity of code, and elegance, all of which are discouraged in mainstream code. By sidelining practicality, the 48

dominant motivation of most language design, they become a space for experimentation and play with pure idea and challenge the seemingly unalterable, unquestionable givens of traditional programming conventions. However, not every language adopts this Less Humble aesthetic. Some esolangs point to other possibilities of the medium.

49 The language Unnecessary by Keymaker has only one possible program: the program that doesn't exist. If the program does not exist, it will run and print its own source code to the screen (which is nothing). If one tries to run a different program, one that exists, the program will end in error. Unnecessary points to possibilities of esolangs as pure idea-art. It is not alone; esolangs, in their challenge to conventional ideas of code, can end up as provocations with no code. At this point, most of these languages are still grouped in the "joke" category of the esolangs wiki, alongside masterpieces that have not yet reached their full recognition. It is as if the esolang community does not yet know what to do with languages that break from Less Humble aesthetics.

50 Even Unnecessary gives consistent results when run repeatedly. Other languages force the programmer to give up control of their code, in what seems like a fundamental betrayal of what a language should do. The language Entropy, which I wrote in 2011, reverses the script from the Turing Tarpits. Entropy programs are written in perfectly legible code, a mix of C and Pascal; algorithms can be clearly stated and written for anyone to read, they break the mold of esolangs as Less Humble. But when they run, data is treated as a limited resource. The more often it is accessed, the more likely it will go "off" by a small percentage. The longer the program runs, the more nonsensical its data becomes, until it fades into pure randomness. This gives the programmer a short time to get their idea across to the user before the program derails [Temkin 2018]. Similarly, cat++, a language by Nora O'Marchu, forces programmers to control the machine through cats who mostly ignore entreaties by the programmer and need to be tempted by rewards and offerings. While computation is theoretically possible, in effect it hardly ever happens, as the cats go about their business. These languages explicitly comment on the drama of programming itself [Palop n.d.].

51 More recently, there have arisen languages that challenge the Western bias (and particularly the nearly complete dominance of English in lexicons) of programming tools. Yorlang, created by Karounwi Anuoluwapo, a Nigerian developer who spent his day writing code for foreign clients, mostly communicating with them in English. H e and his friends wanted to be able to put English aside and write code in a language based in Yoruba, the language they speak to each other. Yorlang rejects the cultural biases of programming. Yorlang draws from 'alb, a language by Ramsey Nasser that set out to become a tool for Arabic speakers to learn to code but instead became an esolang illustrating the Western biases of the underlying tools. Cree# expands what is possible to represent in code, by offering an alternate system of logic that challenges Western defaults. These languages work against aspects of mainstream programming languages, in their de-centering of English. They are esolangs in this rejection, and in not prioritizing practical coding, but otherwise adopt none of the Less Humble principles [Anuoluwapo 2018].

52 Esolangs as a medium began with a series of languages that realize the potential for personal expression and for elegance within chaos. This basis has given newer esolangers a foundation on which to raise new questions, challenge base assumptions of who languages are designed for and how they should be used. As more programmers, poets, and artists move into this medium, the questioning and confrontational spirit of the early esolangs finds new articulation.

Works Cited

Anuoluwapo 2018 Anuoluwapo, Karounwi. (2018) "Yorlang." https://anoniscoding.github.io/yorlang/.

Backus 1980 Backus, John. (1980) "Programming in America in the 1950s- Some Personal Impressions." In *A titlestory of Computing in the Twentieth Century*, 1st Edition, 125–35. Academic Press.

Beder n.d. Beder, Jesse. n.d. "Fractran." Accessed July 1, 2022. https://github.com/jbeder/fractran.

Bogost 2016 Bogost, Ian. (2016) *Play Anything*.

Brainfuck n.d. "Brainfuck Constants." n.d. Esolangs.Org. https://esolangs.org/wiki/Brainfuck_constants.

Chouchou 2011 Chouchou, Alexios. (2011) "Soup!" https://www.bedroomlan.org/coding/soup/.

Chun 2013 Chun, Wendy Hui Kyong. (2013) *Programmed Visions*. *Programmed Visions*.

https://doi.org/10.7551/mitpress/9780262015424.001.0001.

Dijkstra 1968 Dijkstra, Edsger W. (1968) “Letters to the Editor: Go to Statement Considered Harmful.” *Communications of the ACM* 11 (3): 147–48. https://doi.org/10.1145/362929.362947.

Dijkstra 1972 Dijkstra, Edsger W. (1972) “The Humble Programmer.” Communications of the ACM 15 (10): 859–66. https://doi.org/10.1145/355604.361591.

FORTH, Inc. n.d. Forth, Inc. (n.d.) “Forth in Space Applications.” FORTH, Inc. Accessed January 7, 2022. https://www.forth.com/resources/space-applications/.

Fazi 2018 Fazi, Beatrice. (2018) *Contingent Computation*. London: Rowman and Littlefield International, Ltd.

Fuller 2013 Fuller. (2013) “Elegance.” In *Software Studies*. pg. 88. https://doi.org/10.7551/mitpress/9780262062749.001.0001.

Hopcroft, Motwani, and Ullman 2001 Hopcroft, John E., Motwani, Rajeev, and Ullman, Jeffrey D. (2001) *Introduction to Automata Theory, Languages, and Computation, 3rd Edition*. *ACM SIGACT News*. Vol. 32. Pearson. https://doi.org/10.1145/568438.568455.

Knuth 1974 Knuth, Donald E. (1974) “Computer Programming as an Art.” *Communications of the ACM* 17 (12): 667–73. https://doi.org/10.1145/361604.361612.

Knuth 1997 Knuth, Donald E. (1997) *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* Ttitlerd Edit. Addison-Wesley.

Knuth 1998 Knuth, Donald E. (1998) *The Art of Computer Programming. Volume 3: Sorting and Searctitleng.* Second Edit. Addison-Wesley.

Montfort and Matteas 2005 Montfort, Nick, and Matteas, Michael. (2005) “A Box Darkly,” 10.

Müller 2017 Müller, Urban. (2017) “Brainfuck: or how I learned to change the problem.” *Tamedia TX 2017*. Available at: https://youtu.be/gjm9irBs96U?t=8725

Palop n.d. Palop, Benoit. (n.d.) “‘Cat++’ Is a Visual Live-Coding Language Based on Feline Behavior.” *Vice*. https://www.vice.com/en/article/kbn5qe/cat-visual-coding-language.

Pedis 1981 Pedis, A. (1981) “Epigrams on Programming.” *ACM Sigplan Notices* 17 (9): 1–5. http://octopus.library.cmu.edu/cgi-bin/tiff2pdf/simon/box00075/fld05959/bdl0003/doc0002/simon.pdf.

Pressey and Temkin 2015 Pressey, Chris, and Temkin, Daniel. (2015) “Interview with Chris Pressey.” *Esoteric.Codes*. https://esoteric.codes/blog/interview-with-chris-pressey.

Pressey n.d. Pressey, Chris. (n.d.) “SMETANA.” Accessed July 1, 2022. https://catseye.tc/article/Automata.md#smetana.

Roio 2000 Roio, Jaromil. (2000) “ASCII Shell Forkbomb.” https://jaromil.dyne.org/journal/forkbomb_art.html.

Sack 2019 Sack, Warren. (2019) *The Software Arts*. https://doi.org/10.7551/mitpress/9495.001.0001.

Stackexchange 2016 Stackexchange. (2016) “The Letter A without A.” Codegolf.Stackexchange.Com. https://codegolf.stackexchange.com/questions/90349/the-letter-a-without-a.

Temkin 2018 Temkin, D. (2018) “Entropy and Fatfinger: Challenging the Compulsiveness of Code with Programmatic Anti-Styles.” “Leonardo 51” (4). https://doi.org/10.1162/LEON_a_01651.

Temkin 2020 Temkin, Daniel. (2020) “Chef and the Aesthetics of Multicoding.” *Esoteric.Codes*. https://esoteric.codes/blog/chef-multicoding-esolang-aesthetics.

van Oortmerssen and Temkin 2015 van Oortmerssen, Wouter, and Temkin, Daniel. (2015) “Interview with Wouter van Oortmerssen.” *Esoteric.Codes*. https://esoteric.codes/blog/interview-with-wouter-van-oortmerssen.