

202 120 202 000 100 221 210 002 010 221 020 001 212

Daniel
Temkin

FORTY-FOUR ESOLANGS
The Art of Esoteric Code

PREFACE

I first decided to make my own language when I read an academic paper on the game Minesweeper.⁸

In the signature computer solitaire game of the 1990s, you click on tiles, one at a time, revealing a mine—if you are unlucky—or a safe tile indicating how many mines are nearby. The paper proved that an accidental simulation of a computer lurked in the mechanics of the game, written in the partial knowledge of where mines might be. As each mine is revealed, the locations of others resolve, creating flows through the physical space of the board. These flows combine to simulate logic gates and ultimately perform computation. No one needed this system to exist, it had no purpose, and yet it could represent any algorithm written in the eagerly utilitarian languages of Python (van Rossum, 1991) or C# (Hejlsberg, 2000).⁹

Commercial and academic languages, each in their own way, aim to minimize the cognitive work needed to understand code's performance. But this accidental computer showed a radically different path: what a pure code anti-style might look like. It felt like a challenge to see how far one could push the concept of code, reconsider what it is for, what form it can take, and what questions it can ask. Designing a language that similarly broke from computational norms seemed the best way to explore these questions.

I was not the only one interested in the peripheries of language design. A rich hacker culture of esolangs had begun in the early 1990s. The parody language INTERCAL (Woods and Lyon, 1972) was revived as C-INTERCAL (Raymond, 1990), where one literally pleads with the interpreter to run their cryptic code. The minimalist languages FALSE (van Oortmerssen, 1993), brainfuck (Müller, 1993), and Befunge (Pressey, 1993) began as experiments—how small can I make a compiler? what if we could jump to another location in code by pointing to it?—and became systems of thought with their own unique logic, ripe for exploration.¹⁰

Rarely does an esolanger discover the most interesting possibilities of their language on their own; it is others' experiments that reveal the nuance and character of a language. The first esolang I fell in love with was Piet (Morgan-Mar, 2001) (named for Piet Mondrian and so pronounced "Pete"). Piet programs are written as a progression of colors in orthogonally pixelated images resembling mosaics or low-fi graphics. Most programs are immediately recognizable as Piet, with its familiar palette and blocky shapes. Yet others counter this aesthetic, perhaps obscuring the Piet program within a larger image with colors outside the language's palette, or finding their own style within Piet's strict rules. This collaborative spirit between programmer and language designer comes naturally to free and open source software culture. The scenario the esolanger creates is explored through the programs others write for it, bringing their own sensibility and logic to the project. This interplay reminded me of the constraint-based writing of Georges Perec and the other Oulipians. I immediately wrote my own language in response to Piet. It is Language 0 in this book.

I began an interview series with esolangers to learn more of their thinking beyond the technical aspects of the work that tend to dominate esolang discussion, and to document the history of the form. This grew into the blog esoteric.codes, which, for ten years, considered esolang aesthetics alongside code art and code poetry, encouraging a broader context that connects to other computational art.

I noticed that it was common practice for esolangers to keep a to-do list of languages yet to be implemented. These can often be found in the user pages of the esolangs.org wiki, the central archive of these languages. Some stand alone as concepts that don't need a working interpreter to express their idea: simply considering the description gives an experience of the language. It runs in our heads. Others are waiting to be crystallized into definitive sets of rules. Sol LeWitt differentiated between the concept and the idea of a work: a concept is a general direction, and the idea fills in the specifics of that concept. Esolangs, a hacker folk art, developed a parallel to this, created by programmers with little interest in conceptual art but also working in an open-ended, dematerialized

form. It is just one way these two practices have developed similar habits and strategies in exploring systems, logic, and irrationality.

In this book, I share forty-four esolangs I wrote between 2009 and 2025. Each language is presented first as a Prompt, then as a Realization, mirroring LeWitt's division of concept and idea. The realization is summarized; full technical definitions or working interpreters are collected at <https://danieltemkin.com/esolangs> for those who want to experiment further. Sometimes the realization flows naturally from the prompt, but often a single prompt can give rise to many interpretations. Others are left unrealized. This could be because they don't need implementation, because implementation would lead to logically impossible results, or because their best realization has not yet occurred to me. Mine are not the only possibilities, and I invite readers to interpret the prompts differently.

We write code in idioms and metaphor. Each programming language has two aspects. The syntax—its grammar and lexicon—defines what qualifies as code, splitting texts into those that belong to the language and those that don't. Its semantics are embodied through a virtual machine providing a set of behaviors the programmer can set in motion. It is a metaphor, as all computers are. The language might make use of familiar abstractions, giving us stacks and memory cells, but it might take the form of a Turing Machine lurking in a solitaire game. It might map easily to the physical box we use to run our code or it might be impossible for our machines to perform.

All the implementations of the languages are open-source and released under the permissive MIT License. Make your own programs, your own implementations, or bring your own vision to them. They are languages, meant to be used.



Implementations can be found here:
<https://danieltemkin.com/esolangs>

8. It was actually two related papers: Richard Kay, “Some Minesweeper Configurations,” *Boletim Sociedade Portuguesa de Matemática* (2007): 181-189, and Richard Kay, “Infinite Versions of Minesweeper Are Turing Complete,” *School of Mathematics preprint number 2000/15* (2000), University of Birmingham, Updated May 31, 2007.

9. So long as we can endlessly expand the boundaries of the board.

10. “Interview with Chris Pressey,” esoteric.codes, May 12, 2015, <https://esoteric.codes/blog/interview-with-chris-pressey>.

SENTENCES ON CODE ART

(2017, 2024)

0: Computers are logical systems that arise as often by accident as by design.

1: Their core materiality is logic, not pixels or circuits or bits or other features of their physical implementation. Other implementations are possible.

2: Our engagement with logic is irrational because we are irrational beings. We are incapable of fully asserting our agency through a series of logical steps.

3: This central drama of human–computer interaction is experienced most directly at the code level.

4: Bugs are the primary progeny of programmers. We write broken software.

5: Although the machine presents a world of our own making, it rebukes us by not doing what we want, leading to a compulsive cycle of fixing and augmenting code. To borrow the words of Joseph Weizenbaum, we are all computer bums.

6: For any definition of the term “programming language,” there is a language that sits on its border.

7: Programmability is not a requirement for programming languages. Theoretical programming languages existed before practical ones. We can create valid languages whose programs are physically unconstructable or whose executors are logically impossible.

8: The ambiguity of human language is present in code, which never fully escapes its status as human writing, even when machine-generated. We bring to code our excess of language.

9: We don't need an irrational idea to follow logically; our irrationality will pollute any attempt at rigor.

A: Code is code because it conforms to the grammar of a language. The computer is the person or thing that executes code as the language specifies. In this way, the language defines everything else.

PROMPTS

LANGUAGE 4.

each day, a programmer records the paths of clouds across the sky as they

- * drift
- * morph
- * split
- * combine

when the sun sets, they review, to glean the program's purpose

this language exists for a single program as observed by a single person on a single day

LANGUAGE 40.

a programming language written in tangles of lines

commands are inscribed in the turns each line makes as it flows down the page

strands never split: there are no alternate paths and every command always runs

should it leave the program in an undesirable state, another strand undoes its work

LANGUAGE 41.

a programming language whose symbols are homonyms with many meanings

each possible reading plays out in parallel

the program is an accumulation of all possible interpretations

REALIZATIONS

LANGUAGE 4.

each day, a programmer records the paths of clouds across the sky as they

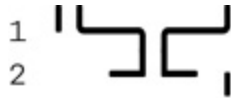
- * drift
- * morph
- * split
- * combine

when the sun sets, they review, to glean the program's purpose

this language exists for a single program as observed by a single person on a single day

PERFORMED AS Cloud Computing (2018, 2020, 2022)

Documentation kept for personal reflection.

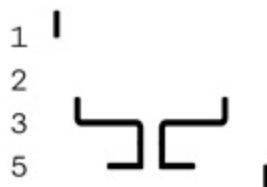


The `|` and `|` characters mark the beginning and end of the glyph, which is a tangle of strands that execute together. The numbers 1 and 2 on the left are line numbers; they have no effect but make it easier to evaluate the strands visually.

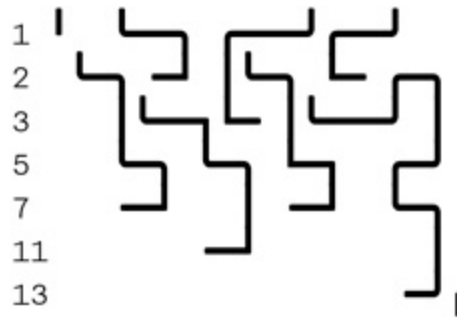
The two strands each have a hook pointing up at their start: `└` or `┘`. These mark them as *value strands*, which add or assign to an element of a list. These begin on line one, so they assign to elements of list one.

Their movements left and right add or subtract from their value. The first strand moves two segments to the right on line one, adding one twice. It then moves one segment to the left on line two, subtracting two. This sums to zero. The second strand does the same in opposite movements: subtracting one twice, adding two once. Since they both evaluate to zero, these are null-strands, placeholders that do nothing on their own.

If the same code moves down two lines, leaving the glyph markers in place, it evaluates differently. They now move across lines three and five, evaluating to 1 and -1 instead of 0, and assigning to the third list, as they begin on line three:



You may notice there is no line four. In Rivulet, all line numbers after the first are successive primes, and these numbers reset to one for each new glyph. Here is a more complex set of null-strands, each evaluating to zero but taking different paths:



Rivulet's visual vocabulary comes from the comforting compactness of mazes, space-filling algorithms, and visual work like Anni Albers's *Meander* series. Having options in how to draw strands is important for concise code, allowing strands to not block other strands in their glyph.

There are four different strand types, governed by different rules on how they flow. A *reference strand* begins with the same type of hook as a value strand but ends with little gap: — -. It refers to two list elements rather than a list element and a constant. These are marked by where the strand begins and ends in relation to where other strands start. Unlike value strands, its route through the glyph does not affect its semantic value.

By default, value and reference strands perform assignment or addition. An *action strand* allows for other commands. It sits directly below the starting hook of the strand it applies to and has a hook pointing down or to the right. It's read in vertical, rather than horizontal, movements, as if it were a value strand turned ninety degrees. Its “line numbers” are in relation to where it begins, not to the glyph as a whole.



The action strand above, regardless of where it begins, resolves to -1; it moves down for one vertical bar, adding one, then to the left

and up one, subtracting two. Negative one is the code for subtraction assignment.

Finally, there is the *question strand set*. A strand with no hooks marks the beginning of a question, and below that, a strand dances back and forth through the glyph, often filling its remaining cavities. Its movements are meaningless apart from where it ends, which indicates whether it tests for equality or positivity, for a list element or an entire list.

Most programming languages have some concept of branching: an `if` statement that marks a set of instructions to run only if a condition is met. In Rivulet, every branch of the program runs every time. Rivulet offers only a conditional rollback. After a glyph executes, its question strand fires. Should its condition be found wanting, the glyph's work is unraveled, returning the program to its previous execution state as if never run. All loops end by rolling back their last iteration. In this way, Rivulet always tests the *outcome* of a scenario instead of what leads up to it. One could say that no thread is ever left hanging.

Here is a glyph with all the strand types together:

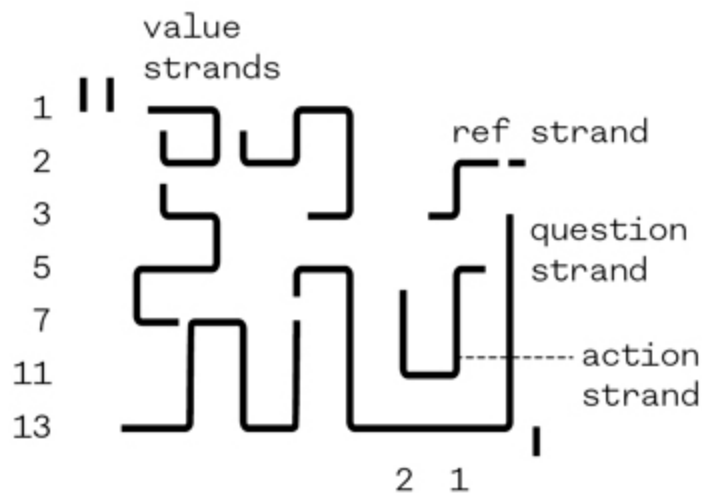


Figure 2.16 An annotated Rivulet glyph

LANGUAGE 41.

a programming language whose symbols are homonyms with many meanings

each possible reading plays out in parallel

the program is an accumulation of all possible interpretations

REALIZED AS Valence (2024)

Valence is a programming language of chance associations and accidental algorithms. It is written with eight homonymic symbols:

⋄ ∩ ∪ ∂ ∫ < ∞ ⊥

These are borrowed from Ancient Greek numbers and measuring signs but have no connection to their original meaning. Each has several unrelated interpretations, disambiguated through context. They are the only symbols recognized in this language; other letters or numbers are completely ignored (with the exception of square brackets as explained below). The sign \int indicates division, the octal digit 2, and the variable \int among its possible readings. It also can force an expression to evaluate to a variable name rather than a value.

Many programming languages use prefix notation, with the command appearing first. In the case of addition, it may look like + A B C. Others use infix, written as A + B + C. Valence uses prefix for a single parameter, infix for two parameters. No instruction takes more than two.

Here are some readings for the sign \perp where A and B stand in for other Valence signs:

0 params, nofix:	: the variable or the octal number 7 or the ratio type
1 param, prefix:	A : receive input and assign to A (when read as a command)
1 param, prefix:	A : dequeue an element from A (when read as an expression)
2 params, infix:	A B : multiply assign: $A = A \times B$ (when A is a variable and B another type of expression)
2 params, infix:	A B : multiply: $A \times B$ (when A is another type of expression, not a variable name)

The structural ambiguity of a command and the multiplicity of meanings for each symbol work together to open up many possible readings for a line of code.

Here is a lexicon for the first few signs:

0: the octal (base eight) digit 0, the variable 0, integer, not, add, addition assignment

1: the octal digit 1, the variable 1, subtract, if, subtraction assignment, a movement toward entropy, read as integer

2: the octal digit 2, the variable 2, divide, goto, read as a variable, repeatedly dequeue elements from a queue until empty, end a block of code

Below is a real example:

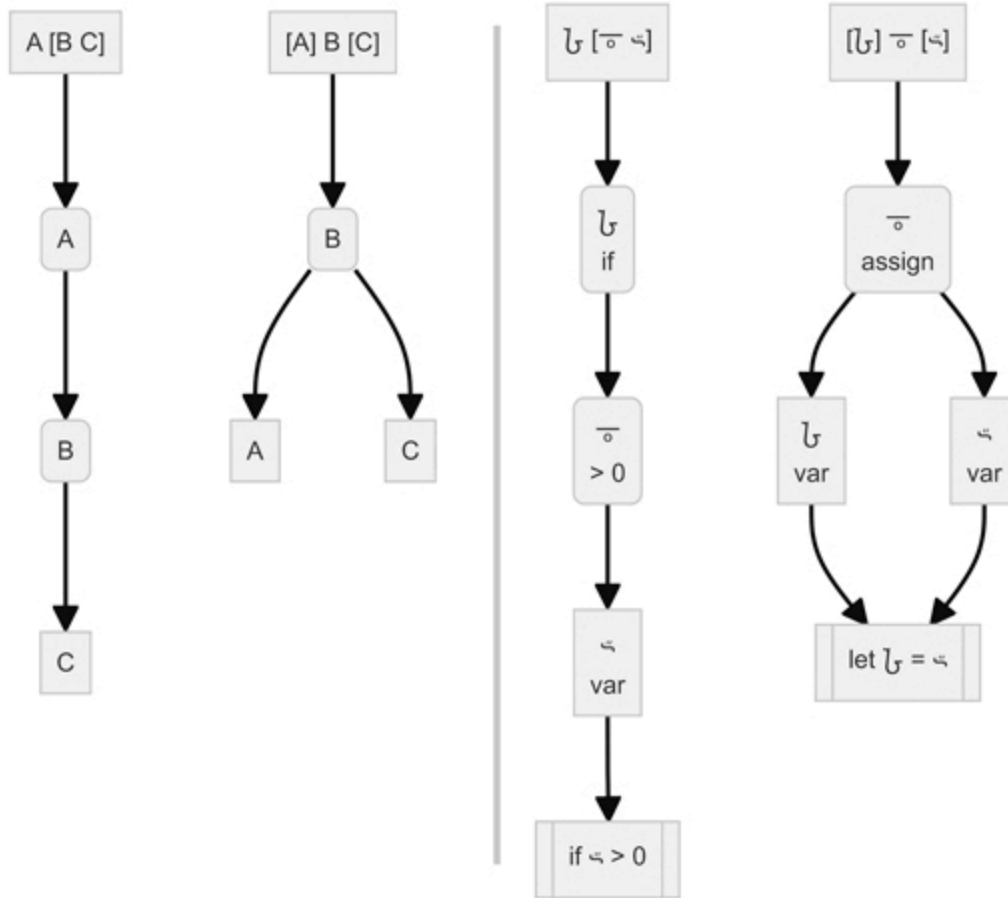


Figure 2.17 Possible interpretations of a three-letter Valence statement

On the left is a breakdown of how three signs might be read, with A, B, and C in the place of Valence instructions. To the right is a specific example. The brackets create groupings to be executed first. In A [B C], [B C] is fired first, with B a prefix function taking parameter C. The alternative, [A] B [C], has B as the function and A and C its parameters.

The example to the right tests whether a variable is greater than zero or assigns a value to a variable. A programmer can impede the proliferation of readings by explicitly marking groupings with square brackets and qualifiers, which force a specific reading. L O S has four interpretations, but $[\text{L}] \text{O} [\text{S}]$ has only two. $[\text{L}] \text{O} [\text{L} \text{S}]$ has only one: adding the second L forces an integer reading of L : as the octal digital 2, rather than the variable L .

When a line of code is left ambiguous, Valence does not pick one over another. Instead, it forks the running program into two, playing out both possibilities side by side. When a second ambiguous line is reached, it splits both programs again. Syntactic correctness is still required, however. The `if 𐌵 > 0` reading must have an `end if` later in the program; without it, only the right reading, `let 𐌶 = 𐌵`, survives.

Valence has several ways to control program flow: one can `goto` a label or `jump` a number of lines away from its current location. Either can be calculated. The label for a `goto` corresponds to the single octal digit a sign also signifies. For example, this line of code has an ambiguous `goto`:

𐌲 𐌵 𐌶 𐌶

It can be read as:

𐌲 [[𐌵]𐌶] [𐌶]
 meaning goto label 𐌶

Or:

𐌲 [𐌵[𐌶]]
 meaning goto label 𐌵

Both use the prefix reading of 𐌲 to mean `goto` but the first version, 𐌲 [[𐌵]𐌶] [𐌶], uses three symbols to arrive at the number one while the second uses only two. This leaves an extra sign in the second case, a not function, transforming one to zero. The digits zero and one then translate to their corresponding signifiers, 𐌶 and 𐌵. The program splits to follow both possibilities.