THE AESTHETICS OF MULTICODING ESOLANGS

Daniel Temkin

bzzz@danieltemkin.com

To bake a Hello World Souffle requires exactly 114 g sugar, 111 ml beaten eggs, 54 ml double cream, and 32 g cocoa powder. (Worth) We might expect a recipe to measure by the number of eggs, rather than their volume in milliliters. Having the level of specificity appropriate more for a chemistry lab than a kitchen is a clue that this recipe is not only for cooking, but also a computer program in the Chef language.

Chef is what we might call a multicoding esolang: a programming language with lexical and grammatical rules such that their programs hold a second meaning when read in another context.

An esolang (for “esoteric language”) is any language not designed for practical programming: thought experiments, art pieces, jokes for programmers, or any language that challenges conventional ideas of computing.

The term multicoding was coined by Michael Matteas and Nick Montfort (as “multiple coding”). A multicoded text holds different meanings within different interpretive systems. An example they use is a sentence that is grammatical in both English and French: "Jean put dire comment on tape" which in French means "Jean is able to say how one types." (Montfort and Matteas)

An example in code is the polyglot, a program that runs successfully in two different languages, like Python and Java, often by employing lesser-used features of those languages. There are polyglots that run in over 200 different languages. (Temkin, “There’s Still a Chance to Participate in This Insanely Large Polyglot”)

Multicoding esolangs bring these two things together: they are specifically designed to hold double meanings. Languages of this type first appeared around 2001 but sporadically enough since then that they do not yet have a term to refer to them collectively. As a designer of these languages, I find the lack of discussion of these languages as a group makes it hard to discuss what is at stake in this form. By grouping them into this category, I hope to flesh this out.

EARLY MULTICODING ESOLANGS

The language Shakespeare was created in 2001 by Karl Hasselström and Jon Åslund. Each program is a play, written by an Elizabethan with an unfortunate fondness for adjectives. Comparisons (equal, less than) are represented as analogies, and math is performed with long strings of qualifiers considered positive or negative within the lexicon. Here is a typical example of what that leads to:

> You are as brave as the sum of your fat little stuffed misused dusty old rotten codpiece and a beautiful fair warm peaceful sunny summer's day. (Hasselström and Åslund)

The aesthetic of Shakespeare does not lead to great writing, nor was it intended to. It was designed as a joke and provocation. In Edsger Dijkstra's "The Humble Programmer," the landmark paper that reflected and solidified the dominant style of code, we are warned that code should avoid personal style in favor a neutral voice favoring clarity. (Dijkstra) Shakespeare challenges this very directly. It was not the first to do so, with precursors dating back (at least) to the parody language INTERCAL of 1972, but Shakespeare did this in a humorous way that helped cement esoteric languages as a challenge to traditional programming.

What is the aesthetic range of Shakespeare programs? This might be a ridiculous question given the intent of the piece, but should be considered in comparison with other approaches to multicoding. The Oulipian movement's study of limits might be helpful here. Oulipian writers experimented with just how constrained their constraint sets can they be and still allow for expressiveness. In his "Three Cases of Pushing Things to the Limit," Le Laonnais said "I fear that the reduction of a poem to a single letter may lie on the far side of the acceptable limit;" comparing the poem "T" to the poem "U." (Mathews and Queneau) However, in his "four-legged m" poem of 1965, Aram Saroyan showed that even with a single letter, poetry is possible. (Stephens)

Shakespeare has a limited expressive range due to the opposite: the length of its pieces, with a typical program running several pages (at least those with textual output). In combination with the small vocabulary its programmer can draw from and the inflexible rules of the language, all Shakespeare programs end up saying more-or-less the same thing.

Contrast this with the language Piet, also created in 2001, by David Morgan-Mar. In Piet (pronounced "Pete" although often mispronounced), images are code. It was inspired by Piet Mondrian, yet its programs do not typically resemble Mondrian's signature style. Piet does not require primary colors; in fact it is difficult to write Piet code with only red, yellow, blue, white and black. Code is interpreted by change in pixels in both hue and brightness, meaning that paler shades of color might be required to make a program work. We can see the challenge in pieces that attempt to appear Mondrian-like.

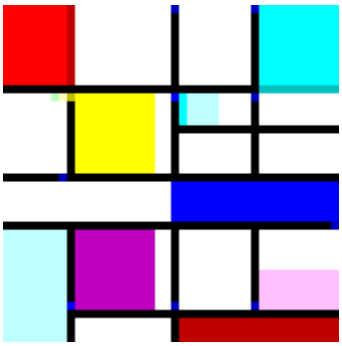

Figure 1. "Piet" by Thomas Schoch

The most widely shared Piet programs are those of Thomas Schoch, including the piece on the cover of the book *Speaking Code* and the program in Figure 1, which prints “Piet” to the screen. (Cox and McLean) Schoch is the virtuoso of Piet, who manages to build a distinct personal style within Piet's rules. If it seems like overkill to use the “v” word for someone responsible for only a few images, the language become synonymous with those images. In my interview with Morgan-Mar in 2015, he described first coming across Schoch’s work (Morgan-Mar, David; Temkin):

> Up until that point, I’d thought that every Piet program would inevitably end up looking like essentially random blocks of pixels of various sizes, with no really detectable rhyme or reason to it.

Those pixel-blobby examples in some ways more straightforwardly explore the aesthetic of Piet as a system. They look like computer-generated images, perhaps a more complex version of Webdriver Torso. However, in reality they are the reverse of generative art: the work is hand-made to fulfill the rules of the language.

Figure 2. "Towers of Hanoi" by Sylvain Tintillier

In Schoch's work, that hand-made aspect becomes more apparent. Piet’s aesthetic proves flexible enough to allow programmers their own distinct style while still operating within the language. They still look like Piet programs. While they may vary in size, they share the same palette, and for the most part the same density of voxels and in similar distributions and clusters, just in different shapes.

It is yet to be seen if another Piet programmer will find their own unique style within the Piet framework as Schoch has.

David Morgan-Mar created Chef the following year (2002). While a cooking recipe might sound as arbitrary a system as other esolangs use, it differs in that recipes *already are* algorithms. Recipes discretize the continuous activity of cooking. They break it down into a set of repeatable steps with hopefully predictable outputs, given the correct inputs. It is a brilliant central metaphor for a programming language.

Every program in Chef is carried out in two different ways: in the kitchen and in the computer. The aesthetics of the language are determined by the mapping between those two activities. In Chef, the variables are “ingredients,” the input is the “refrigerator,” the output and memory are a series of “mixing bowls” and “baking dishes,” and the instructions are the steps of the recipe itself.

The ingredients added to a mixing bowl correspond to data or variables, and the mixing bowl to a stack. This works as an analogy as it is easy to picture, and when you are reading the recipe, it is easy to follow as code.

Other aspects are less clear. If a recipe asks us to remove three eggs from the fridge, we’re creating a variable called eggs, assigning it an initial value of three, but then prompting the user to input a number to replace that three. From that point on, there are any number of eggs in the program, but still only three in the cooking recipe. This is because variability is needed in code but we don't want the number of eggs to be arbitrary or we'll end up with something inedible. In this aspect, the readings as code and a cooking recipe diverge in a way that can be confusing.

It should be noted that Morgan-Mar never cooked a Chef program himself, and only designed one recipe which is almost certainly inedible (as is typical for esolangs of that era). He did not expect anyone to be “masochistic enough” to actually cook and eat a Chef recipe. (Morgan-Mar, David; Temkin)

Ian Bogost is just that masochist. He has been eating Chef recipes for fifteen years. Bogost has been teaching Chef to his creative coding students for that long, often leading to actual cooked dishes. His commentary speaks to how the rules of Chef translate culinarily:

> In some ways, I wish Chef didn’t do text, because it invites a few major issues. The first is that a computer program that outputs a short string is just not that interesting. The second is that Unicode values require very high numeric data, which either results in insanely large quantities of ingredients or requires complex looping to multiply small values up. And the third is that text output requires liquid ingredients or “liquefying” a mixing bowl, which just produces a slurry that probably nobody wants to eat if prepared.

He added:

> The mixing bowl/baking dish paradigm definitely impacts the viability of Chef recipes in a negative way. As a culinary counterpart to the stack data structure, it’s just not a great match for many types of recipes—that, plus the liquefaction problem—makes Chef mostly good for baked goods. So long as that constraint is acknowledged and respected up front, the results can be really interesting. (Temkin, “Chef and the Aesthetics of Multicoding”)

That said, Bogost and his students find the challenge of Chef productive and compelling and he continues to teach the language. Part of this is in the greater challenge of Chef, and generally of working with food as a medium. In Piet, just about any combination of pixels might have defenders in terms of its visual aesthetic. To make a program both edible and executable is challenge enough, let alone tasty.

One must keep in mind that Chef, Shakespeare, and Piet were all created when there were very few examples to model the languages on, and little expectation of engagement with such languages. The term multicoding had not yet been coined, and when such languages were grouped together, it was under a larger category of "thematic" languages. Many are still tagged this way on the esolang wiki. Thematic languages includes ArnoldC and TrumpScript, simple procedural languages dressed up with silly quotes from Arnold Schwarzenegger or Donald Trump (created when Trump was only an amusing sideshow to the 2016 race). Thematic is often used pejoratively, to indicate gimmicky esolangs, mostly designed around fleeting issues or memes of the day (emoji, or lolcatz) and do not challenge computational norms on a deeper basis. (ais523 and Temkin)

It might be easy to confuse the two; after all, only the text of code is unique about Chef and Piet. Behind their unusual texts, they are ordinary stack-based languages. Compare these with esolangs of the era that garnered more respect at the time, such as David Madore's 1999 language Unlambda, offering alternative modes of computation.

This is where I believe it is essential to consider the multicoding esolangs by the strength of their central metaphors and by the programs written in them, in terms of the range of texts and the commitment to that metaphor. An ArnoldC program can be understood by looking at a single example; there is little variation between programs. They are all filled with Schwarzeneggerisms like TALK TO THE HAND; a far cry from the field of programs created for Chef and Piet. While Shakespeare has a more limited range, it incorporates its premise into the structure of the code itself. It can't be turned into conventional code with a simple find/replace like ArnoldC can; no other language has its distinct flow of adjectives. Shakespeare was also innovative in allowing for a large list of (computationally) equivalent words, which had not been attempted before in a programming language.

NEW APPROACHES

The early group of multicoding esolangs can be differentiated from a second wave, including work from around 2005 on (it is a very slow-moving second wave). Beginning when the first group of multicoding esolangs were well established, this second group learns from and explicitly responds to the first. They also are more likely to be made by people from a wider community; hacker/hobbyists but also digital artists and poets, many of whom discovered the promise of esolanging through the multicoding languages.

In 2012, I created Light Pattern, where code is a folder full of jpegs, considered by the interpreter in alphabetical order. Changes in the dominant color of the photo, along with camera settings like aperture and shutter speed, define commands. In Piet, we can look at an image and understand by sight what the program is. In Light Pattern, we don't necessarily know why a picture is darker than the previous one: is it a change in shutter speed, in aperture, or did the sun go behind a cloud? The photograph is evidence of a process—but it is that process of creating the photograph—not the appearance of the photo—which is the vocabulary of Light Pattern. Light

Pattern also doesn't say anything about the content of the photo or what story the series of photos should tell, which are the first, most visible aspect of the work to most viewers. (Temkin, “Light Pattern: Writing Code with Photographs”)

Light Pattern, then, challenges the programmer to create a series of photographs whose content relates to its execution as code.

In the Unmaking series, I do this by performing instructional works by artists like Vito Acconci and photographing them in such a way that those photos will then print the original instructions. This plays off the idea of the Quine program: the program that prints its own code; only here it becomes self-perpetuating only through the continued performance of the work. In “Three Lamp Events,” George Brecht’s event score commanding us to turn on and off lamps are followed, but with lamps holding red, green, and blue bulbs. They are photographed in such a way that the combination of lamp and camera write Brecht’s score in the Light Pattern language. (Temkin, “Light Pattern: Three Lamp Events”)

Figure 3. Still from "Three Lamp Events" by the author

In 2016, Will Hicks created a family of languages called Esopo (a portmanteau of "Oulipo" and "esolang"). Hicks, a poet, responded directly to the Shakespeare esolang:

> When I originally encountered Shakespeare, one of my disappointments with it was how much the algorithmic aspects of the language intruded on its possible artistic expression. It tends to become tiresome after the third “Speak your mind!” (Temkin, “Esopo: Turing Complete Poetry”)

Hicks took a very different approach from Shakespeare. There is not a single Esopo language, but a family of languages, each with its own rules. The first, AshPaper, uses alliteration, rhyming, and capitalization for commands. Instead of asking programmers to write strings (like Shakespeare's infamous "Speak your mind!") to print to the screen, AshPaper draws from how poets already work.

> I developed a lexicon in which particular tools from the poet's toolbox would correspond to algorithmic operations, but I also tried to ensure that there were multiple ways of building any particular operation. A "for loop" might be constructed with end-rhyme and a particular meter or with an appropriate sprinkling of similes, for instance. (Temkin, "Esopo: Turing Complete Poetry")

The second Esopo language is an epistolary one called Correspond. It has two registers, You and Me, turning all computation into reflection on oneself and their relationship with their correspondent. Like with Light Pattern, the Esopo languages create a larger interpretive space, bringing other aspects of the code to the forefront, so it is not entirely defined by the constraints of the language.

The 2009 language bodyfuck by Nik Hanselmann is based on the classic esolang brainfuck, written in all punctuation. It multicodes brainfuck commands with physical actions so that programmers leap in front of their webcam in order to write code. Most importantly, there is no backspace in bodyfuck, meaning that if you make a mistake, you have to begin again. This emphasizes the compulsiveness of code, already there in the original brainfuck language, while making the labor of programming visible and physical. It uses multicoding, not in an Oulipian fashion as a constraint set for possible programs, but instead to dramatize the act of programming itself.

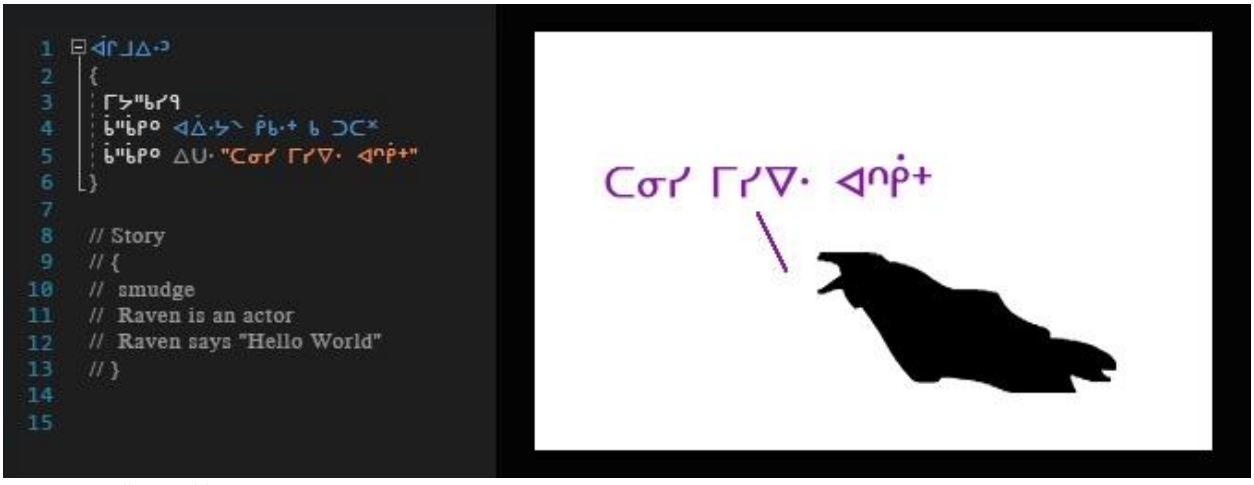

Figure 4. Hello World in Cree#

Jon Corbett's Cree# brings the language and story-telling of Cree tradition into programming.

> [T]he graphical output can't exist without the story that the code is telling… In the above example, we have Raven (the Trickster), a character from Cree folklore who is as well known for his mischievousness as he is known for his role as a teacher and a guide. The

> program initializes when Raven flies on to the screen and lands on a tree and ends when the Raven flies away. But while sitting on the tree, Raven can perform a number of tasks. In the above example, the Raven says “Hello World” (Corbett)

According to Corbett, Piet showed him that such a language was possible, but Corbett was careful about which aspects of the early esolangs to adopt. He eschewed the complexities of stack-based programming used in Piet and Shakespeare. Stack approaches appealed to the early esolangers for the technical ease: for one thing, it means no need for variable names, which can be awkward especially in non-textual languages. Corbett opted for logic more in tune with mainstream programming and so more intuitive for beginning programmers. Corbett goes so far as to offer a friendly IDE for his language, much like the corporate C# for which it is named.

This allows him to bring the focus to the story-telling aspect. Unlike Chef, where the two readings of the code diverge (such as in taking the eggs from the fridge), Corbett unifies. He asks us not to read the Cree# code simultaneously in different and sometimes conflicting systems, but to let the story tell us what the code does. Only by understanding the story can we see how the code runs.

Corbett’s work points out that all programming languages are multicoding to some extent. Code, after all, already has two readings: that by the programmer and that by the machine. Languages that attempt to represent code with great clarity (in the Dijsktrean tradition) still make cultural choices not only in what language they borrow keywords from, but also in how they explain the behavior of that code. Corbett not only draws from Cree tradition for the central metaphors of his language; he also has developed an Indigenous Digital Media Toolkit to enable others to create such languages for their own stories and communities.

By building on the promise of the early languages, the second wave further refines how meaning is expressed in the constraints of multicoding languages. While some might be fully understood through example programs by the language designed (e.g. Shakespeare and bodyfuck), most are sites for collaboration with esoprogrammers, who materialize the languages into specific programs. It is only through that concretization of the language into actual code that the premise of the language is tested and explored. However, most still have relatively small sets of programs; a language of this type might take many programs before we get a sense of it as a constraint set, let alone exhausting its possibilities.

After twenty years of such languages, this is still a very sparse category of work. I hope that, by naming and grouping these languages into a common category, it will help encourage new esolangs with provocative ideas, and perhaps more esoprogrammers to take on those already out there; they are essential to the collaborative creation of meaning.

REFERENCES

ais523, and Daniel Temkin. “Interview with Ais523.” *Esoteric.Codes*, 2011.

Corbett, Jon. “Week 2, Cree#.” *CCS Working Group 2020*, 2020, http://wg20.criticalcodestudies.com/index.php?p=/discussion/71/week-2-cree.

Cox, Geoff, and Alex McLean. “Speaking Code.” *Speaking Code*, 2013, doi:10.7551/mitpress/8193.001.0001.

Dijkstra, Edsger W. “The Humble Programmer.” *Communications of the ACM*, vol. 15, no. 10, 1972, pp. 859–66, doi:10.1145/355604.361591.

Hasselström, Karl, and Jon Åslund. *The Shakespeare Programming Language*. 2001, http://shakespearelang.sourceforge.net/report/shakespeare/.

Mathews, Harry, and Raymond Queneau. *Oulipo Compendium*. 2005, http://books.google.pl/books?id=s-UbAQAAIAAJ.

Montfort, Nick, and Michael Matteas. *A Box Darkly*. 2005, p. 10.

Morgan-Mar, David; Temkin, Daniel. “Interview with David Morgan-Mar.” *Esoteric.Codes*, 2015, https://esoteric.codes/blog/david-morgan-mar.

Stephens, Paul. “Absence of Clutter.” *Absence of Clutter*, 2020, doi:10.7551/mitpress/12603.003.0004.

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

---. “Esopo: Turing Complete Poetry.” *Esoteric.Codes*, 2018, https://esoteric.codes/blog/esopo-turing-complete-poetry.

---. “Light Pattern: Three Lamp Events.” *2015*, p. http://lightpattern.info/Basics/ThreeLampEvents/.

---. “Light Pattern: Writing Code with Photographs.” *ACM SIGGRAPH Art Papers, SIGGRAPH 2015*, 2015, pp. 375–81, doi:10.1162/Leon-a-01091.

---. “There’s Still a Chance to Participate in This Insanely Large Polyglot.” *Esoteric.Codes*, 2018, https://esoteric.codes/blog/theres-still-a-chance-to-participate-in-this.

Worth, Mike. *Baking a Hello World Cake*. 2013, http://www.mike-worth.com/2013/03/31/baking-a-hello-world-cake/.