Coem: A Poetic Programming Language

Katherine Yang
University of Southern California
May 2, 2022

Abstract

Coem (a portmanteau of “code” and “poem”) is a multicoded (Mateas and Montfort 2005) esoteric programming language, or “esolang” (Temkin 2020), that seeks to explore ways that poetry can be made purposeful, and ways that code can be made emotional. With roots in codeworks and electronic literature (Cayley 2002; Mez 2001), aesthetic programming (Soon and Cox 2020), and critical code studies (Marino 2020), the language strives to bring into conversation the fields of programming, poetry, linguistics, and typography. In feminist opposition to general-purpose technology that privileges efficiency and clarity, the language is an experiment in personal computing that foregrounds ambiguity, emotion, metaphor, and typographical design.

The poetic programming language builds upon the aesthetic exemplified by similar works in:verse (Aneja 2016) and Esopo (Hicks 2016), which both transpose the aesthetics of poetry into the structures of code. However, instead of mapping words to mathematical or computational operations, Coem is intended simply as a meditative exercise in truth and expression. Thus, it challenges conceptions of the purpose of language within the contrasting contexts of programming and poetry, producing texts both as a means to an end and as objects bearing an inherent wealth of information and emotion. Furthermore, through the interface of an online editor, the work is presented, in collaboration with the compiler, as an integrated text of source code, output logs, and error messages.

The language is a self-taught exercise in language and computation, beginning from the practical handbook *Crafting Interpreters* (Nystrom 2021) and working towards exploring the possibilities of poetic programming. Through this work, I hope to invite conversation and collaboration on the topic and to offer my work as an example for similar opportunities for self-designed esolangs and codeworks.

1 Introduction

"generative art is interested predominantly in the results created by generative processes"
Inke Arns, "Code as Performative Speech Act"

"poetry makes nothing happen"
W. H. Auden, "In Memory of W. B. Yeats"

From the perspective of the two epigraphs above, code and poetry stand diametrically opposed. Code is seen as a means to an end of producing output or effecting change in a computer system. We write code because we want to make things happen. In contrast, poetry does not effect any visible change in the "real world", and certainly we do not expect it to. Following the above quoted line, Auden's poem continues, "it survives, / A way of happening, a mouth." Auden describes poetry as something that "survives", transcending action in expression, emotion, and dissemination. Indeed, it is "a way of happening", a reflection or exhibition of happenings rather than a specific happening itself.

How, then, might a new poetic programming language find meaning in this fundamental conflict?

Coem (a portmanteau of "code" and "poem") is a multicoding (Mateas and Montfort 2005) esoteric programming language, or "esolang" (Temkin 2020), that seeks to explore ways that poetry can be made purposeful, and ways that code can be made emotional. The language aims to be a medium in which a poet-programmer may engage in a meditative exercise in truth and expression.

The work spans a range of entities: so far, it consists of the Java source code, a JavaScript translation, grammar and language support for the Lezer parser system and the CodeMirror code editor, the documentation website, and the interactive web editor, all of which can be found as individual repositories in the Coem organisation on GitHub.

This paper will primarily describe the features of the language and how they map to overarching philosophies of the language, then close with concluding thoughts about the current state of the project and future plans.

2 Features / Philosophies

2.1 Tokens / Metaphors

Coem is inspired by similar esolang projects such as in:verse (Aneja 2016) and the Esopo family of languages (Hicks 2016). Both languages demonstrate multicoding by encoding programmatic functionality in words from natural language. In in:verse, the poet-programmer defines their own lexicon, where each word maps to a variable, operator, or mathematical function; then, a poem-program can be written where the combination of words is compiled into graphical shader code that produces a piece of generative art. In AshPaper, an Esopo language, poetic features such as syllable count, capitalisation, rhymes, similes, punctuation, alliteration, and whitespace are mapped to low-level operations that ultimately can be used to implement any algorithm.

In contrast, Coem cannot be used to perform mathematical or computational operations and cannot be used to implement algorithms. Though Coem does not map words or linguistic features to

computational operations as regularly as the two examples above, it is still proposed as a multicoding language by the basic definition of having text be "read", or parsed, by both the poet-programmer and the compiler. In consequence, words and symbols are open to a looser re-interpretation and blurring, in consideration of their uses in both programming and literary contexts.

The tokens, or keywords and operators, of the language have each been considered deliberately. A selection of them are discussed here through a critical code studies lens (Marino 2020) as they relate to general philosophies of the language.

- **Optative let.** In grammar, the optative mood is a grammatical mood that indicates a wish or hope regarding a given action. To declare a variable and to assign it an optional value, the poet-programmer wishes it, as if in a prayer or manifestation.

```
let me be true
```

- **Em dashes as argument holders.** In code, argument holders are typically parentheses. Here, parentheses are reserved for RegEx variable names; instead, em dashes are used, evoking the sense of a break in thought or in the sense of providing additional information.

```
know—I am true—
know—I am false—
know—I am nothing—
```

- **Infinitive marker as function keyword.** Liu and Lieberman describe a "programmatic semantics of natural language", observing that natural language syntax and semantics often map neatly to programming constructs (2004). For example, action verbs map to functions (e.g. "eat", "chase") while noun phrases map to classes (e.g. "the maze", "dots"). This relationship is encoded here by using *to*, the infinitive marker, to indicate the definition of a new function, or verb. However, the word *to* is widely versatile in the English language and can be repurposed to give function definitions a variety of different pragmatic framings. Three examples can be seen in the function definitions below: *to* as an infinitive marker indicating the action of breathing; *to* as a preposition identifying Matilda as the recipient of something; and *to* as a preposition expressing motion in the direction of the sea.

```
to breathe——:
  † something here
```

```
to matilda——:
  † something here

to sea——:
  † something here
```

- **Dot as block terminal.** In (object-oriented) programming, the dot is traditionally used as an operator indicating ownership. Here, the literary dot is transposed into a programming context, indicating not the end of a sentence but the end of a block (the group of statements comprising an if statement, while statement, or function declaration).

```
to breathe——:
  say—“one two three”—.
```

- **Ampersand as return operator.** In code, the ampersand is typically used as the AND operation—one ampersand for bitwise AND and two ampersands for logical AND. In this language, the word “and” is instead used for the logical AND operation, freeing up the ampersand for another purpose. In natural language, the ampersand imparts a certain sense of symmetry and completion by joining two parts of a whole. Here, it’s used to return a value from a function, thus completing the purpose of the function.

```
to breathe——:
  say—“one two three”—
  & “release”.
```

- **Dagger as comment.** The dagger is a specialised character typically used to begin a footnote in literature, often when the asterisk has been used for a previous footnote on the same page. Here, it’s used in a similar context of denoting additional information as a comment in the code.

```
† a comment here
```

2.2 Regular expressions / Ambiguities

A regular expression (or RegEx) is a text pattern that uses operators to capture a range of possible strings. The pipe operator (`|`) matches either the preceding or proceeding token; the question mark operator (`?`) matches either 0 or 1 of the preceding token; the asterisk operator (`*`) matches 0 or more of the preceding token; the plus operator (`+`) matches 1 or more of the preceding token, and parentheses group characters together. Regular expressions are often used in search and replace operations, where searching for a pattern rather than a fixed string can return many different strings that match the query.

In Coem, variable and function names (or identifiers) can be written in RegEx.

One practical use of this feature is to enable the poet-programmer to provide inflected forms of a word to express grammatical case, which can smooth out potential awkwardness in the readability of the word in different contexts. For example, the poet-programmer can create a variable to refer to the entity of themselves by writing `I|me|myself`, which can then be referred to in the following lines of code using any form of `I`, `me`, or `myself`.

```
let I|me|myself be true † creates a variable that can be referred to using I, me, or myself
if—I am true—: † using a variant of the variable that happens to be its subject case
  let me be false † object case
  know—myself—. † reflexive case
```

Another use of this feature is to enable the poet-programmer to express multiple poetic possibilities. In Dan Waber's "Regular Expressions as a System of Poetic Notation", he writes about the poetic potential of regular expressions. He describes the potential of RegEx to be "a system of notation to augment, to build upon, to multiply the possibilities of language to make nets for catching truth." Strikingly, he writes, "When reading about regular expressions substitute the word 'means' wherever the word 'matches' occurs and the text will become about poetics" (Waber 2008). In the following examples, the patterns can match, or mean, multiple words, giving rise to a wealth of subtly different meanings.

```
let s(e|a) be "blue"
let mis(t|sed) be "thick"
let mou?rning be "dark"
```

Using the RegEx operators, each pattern produces two different words: `s(e|a)` produces `see` and `sea`; `mis(t|sed)` produces `mist` and `missed`; and `mou?rning` produces `mourning` and `morning`. Each of the two words, placed in context with the accompanying adjective, produces a different interpretation: the speaker sees blue and the sea is blue; the mist is thick and the subject is thickly missed; the morning is dark and the emotion or act of mourning is dark. Finally, in the way that the two words are so tightly coupled—sharing letters and visibly overlapping so indeed as to appear simultaneously as one word—the pattern gives rise to a strong conflation of the two

co-occurring concepts. The reader is led to imagine that, say, not only is the morning dark and the emotion or act of mourning is dark, but the two ideas are inextricably related.

Recalling one of the primary applications of regular expressions, this feature gives rise to a poetic sense of using RegEx patterns to search for meaning. Programming languages typically value precision and disambiguation: in order to carry out the programmer's instructions, the compiler expects to know exactly what is meant by the given sequence of characters. Conversely, poetry often leaves untold space for interpretation in the meanings of individual words and their relationships with each other. The concept of multicoding (Mateas and Montfort 2005) describes how we, as readers of code, can already bring our linguistic associations into interpretations of code as a text—even more so in esolangs, where keywords may be chosen to deliberately invoke those associations with domains of, say, cooking (Morgan-Mar 2002) or stage plays (Wiberg and Åslund 2001). Through the uniquely technical conceit of regular expressions, Coem further opens up the space for interpretation in programming and encodes the search for meaning inherent in the writing and reading of poetry.

2.3 Identifiers as strings / Code as text

If a word is found that is neither a keyword in the language nor a previously defined identifier in the current environment nor a previously defined identifier in the enclosing environment, Coem does not throw an error stating that the variable name is undefined. Instead, it interprets the word as a string without quotation marks. A one-line program demonstrating the concept is given below.

```
let me be alive
```

This line of code creates a variable named `me` and sets it to the value of the variable named `alive`. A typical compiler might throw an error: it is unable to find any variable named `alive` in the code context and thus cannot retrieve its value to pass to the new variable. However, in this esolang, we take poetic license to assert that `alive` is, in fact, a concept that is defined, albeit highly imprecisely and idiosyncratically and, well, not in the code. Just as we bring to natural language our real-world knowledge about how words map to real-world entities and concepts, we easily and immediately bring to this line of code our own definitions of `alive`, especially as applied to a variable named `me`.

2.4 Directives / Customisability

Borrowing from the tradition of preprocessor directives in languages such as C and C++, the Coem language features directives, denoted by the pound symbol, which allow the poet-programmer to customise how the machine interprets them and interacts with them. The directives range from the functional, such as the *#as palimpsest* directive that retains a variable's history, to the emotional, such as the *#be gentle* feature that softens the tone of error messages. An example of the palimpsest directive is given below.

```
† based on Emmy Meli's "I Am Woman"

#as palimpsest

let I|me|myself be "woman"
let me be "fearless"
let me be "sexy"
let me be "divine"

know—myself— † woman fearless sexy divine
```

By giving the compiler the directive to treat variable definitions as if writing on a palimpsest, a variable can become much richer. Over the course of the program, the variable `me` has been redefined with multiple values. Traditionally, its history of shifting identities would be discarded and the variable would retain only its last definition. However, using the palimpsest directive, we are able to query the variable after it has been redefined multiple times and see that its journey has led it to contain multitudes.

2.5 One-panel editor / Source code as primary text

Most code editors are typically composed of two parts. The first is the editor, where the source code lives. The second is the console, where the program's output is printed, as well as any error messages. In some cases, the console only prints error messages and the program's output is instead printed to another area of the editor. In many cases, though most of the programmer's time is spent writing and reading source code, the source code is in fact not the primary object of attention, especially after the writing of the program has been completed. The source code serves as a means to an end, the hidden machinery inside a black-box mechanical system.

In Coem, the source code, output, and error messages are all integrated as one text, presented together as one consolidated piece of code.

In the web editor, the primary action that the poet-programmer can enact on the code is not labelled as "run" or "execute" but "reflect". When this button is clicked, the program is tokenised, parsed, and interpreted, as expected. However, rather than directing output into a separate area, output is reflected back into the source code as a comment to the side of the associated line. Similarly, errors are printed on a new line beneath the associated line of code. This approach of privileging the source code as the primary text reframes both output and errors as parts of the text that the poet-programmer is writing. The compiler is a collaborator in producing those elements, but by folding them back into the source code, they are presented more as natural elements of the written object than as incidental side effects or accidents. Indeed, the poet-programmer may choose to intentionally write an erroneous line of code to produce an error message that helps achieve a particular effect in their poem-program.

Thus, the compiler is to the poet-programmer as the editor is to the author: it "reads" the text and passes it back, having added its own notes in the margins. The nature of this collaboration or conversation can be further characterised through the use of directives: a proposed feature might be *#be judgmental*, by which the compiler is given permission to identify lines of code that are redundant, such as a line that defines a variable that is never used afterwards, and delete them from the text, thus "optimising" the program. As a project positioning the compiler as a co-producer of the text, this design choice reflects an interest not necessarily in the ability of the computer to produce random or stochastic effects, but in its ability to validate the poet-programmer's agency and intentions by providing a structured system in which play can occur.

Another aspect of this approach is the attention it draws to the constructs around (non-)linearity and the presence or lack of context in programming.

Programming languages with control flow are non-linear in nature, as built-in structures such as goto statements, conditions, and loops cause lines of code to be executed out of order. However, when the compiler parses the source code, the source code is flattened into a stack of calls, which is then reflected in the output as isolated lines of printed messages. The output is thus disconnected from the source of the text. If a program contains a line that prints the value of a variable, the output displays only that value, without no context for, say, the variable name, where it was born, where it dies, and how it relates to other variables. Specialised debugging tools such as breakpoints and stepping commands can help provide some context, but they are not always available nor deemed necessary by default.

By tying output and errors directly to the sides of the lines that produced them, Coem emphasises the context from which these messages originate. Using the metaphor that the compiler is "echoing" messages back to the poet-programmer, especially using know or say, aliases of the print function, this paradigm creates a more immediate and intimate sense of a collaboration or conversation between poet-programmer and compiler.

2.6 Syntax highlighting / Code as display object

In Coem, the source code not only functions as the primary text but is also presented as a display object to be viewed and appreciated. Today, aesthetic programming (Soon and Cox 2020) is most popularly seen in an increasing number of syntax highlighting themes, code editor themes, coding fonts with specialised ligatures, and even tools such as Carbon that produce pretty-looking screenshots of pasted code snippets. These products reflect the growing sense that, as we read more code, we are more interested in or open to the idea of giving code the same typographical treatments that we give to text in other design-oriented contexts.

In the Coem web editor, code is set in the IBM Plex font family, with different fonts and limited monochrome colours signifying and differentiating types of tokens. Specifically, keywords are set in italic serif; operators are set in roman serif; identifiers are set in monospace; strings are set in sans-serif; and comments and directives are set in italic serif, in grey.

Typographical detail is also encoded in the grammar of the language: argument holders are em dashes rather than hyphens; quotation marks are curly quotes rather than straight quotes.

These design choices draw attention to the purpose and effect of syntax highlighting. Traditionally, syntax highlighting is a feature of text editors that displays text in different colours and fonts to help with scanning, comprehension, and error identification. When writing code with

syntax highlighting enabled, there is a satisfaction in writing words that are instantaneously and successfully parsed, as reflected in the visual transformation of the words in the editor. Conversely, when writing poetry, there is no sense of having written the "right" words: with the exception of works of constrained writing, the onus is on the poet to decide if there should be a word, which word it should be, and where it should be placed. The syntax highlighting of this language and editor imbues the experience of writing poetry with the sense of satisfaction and assurance derived from the visual manifestation of a successful parse.

3 Conclusion

As a learning process of exploration and experimentation, Coem continues to be a work in progress as I noodle on expanding ideas about more or different features and philosophies. An aspect of the project that offers significant value to both myself and others is to create more sample works that demonstrate my intentions behind certain design choices and about how the language could be used, which means letting myself take more time to play with the language. Over time, I hope to develop the language and editor robustly enough that other poet-programmers may feel inspired to create their own works in the language, especially if their intentions and interpretations differ from mine. Most generally, I hope to invite conversation and collaboration on the topic of poetic programming and to offer this work as an example for similar avenues of exploration in esolangs and codeworks.

References

Aneja, Sukanya. 2016. 'in:verse || A Poetic Programming Language'. 2016. https://inverse.website/.

Arns, Inke. 2005. 'Code as Performative Speech Act'. *Artnodes* 0 (4). https://doi.org/10.7238/a.v0i4.727.

Auden, W. H. 1940. 'In Memory of W. B. Yeats'. In *Another TIme*. Random House. https://poets.org/poem/memory-w-b-yeats.

Cayley, John. 2002. 'The Code Is Not the Text (Unless It Is the Text) › Electronic Book Review'. 2002. http://electronicbookreview.com/essay/the-code-is-not-the-text-unless-it-is-the-text/.

Hicks, William. 2016. 'Esopo'. William Hicks. 6 July 2016. https://wphicks.github.io/esopo/.

Liu, Hugo, and H. Lieberman. 2004. 'Toward a Programmatic Semantics of Natural Language'. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 281–82. https://doi.org/10.1109/VLHCC.2004.59.

Marino, Mark C. 2020. *Critical Code Studies*. MIT Press.

Mateas, Michael, and Nick Montfort. 2005. 'A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics'. In *Proceedings of the 6th Digital Arts and Culture Conference*, 144–53. IT University of Copenhagen. https://nickm.com/cis/a_box_darkly.pdf.

Mez. 2001. '_the Data][h!][Bleeding Texts_'. 2001. http://netwurkerz.de/mez/datableed/complete/index.htm.

Morgan-Mar, David. 2002. 'DM's Esoteric Programming Languages - Chef'. 2002. https://www.dangermouse.net/esoteric/chef.html.

Nystrom, Robert. 2021. 'Crafting Interpreters'. 2021. http://craftinginterpreters.com/.

Soon, Winnie, and Geoff Cox. 2020. *Aesthetic Programming: A Handbook of Software Studies*. Open Humanities Press. https://aesthetic-programming.net/.

Temkin, Daniel. 2020. 'The Aesthetics of Multicoding Esolangs'. In *Proceedings of the Electronic Literature Organization Conference 2020*, 9. University of Central Florida. https://stars.library.ucf.edu/elo2020/asynchronous/proceedingspapers/17/.

Waber, Dan. 2008. 'Regular Expressions as a System of Poetic Notation'. *P-QUEUE* 5 (August). https://pqueue.files.wordpress.com/2017/10/p-queue-volume-5.pdf.

Wiberg, Karl, and Jon Åslund. 2001. 'The Shakespeare Programming Language'. The Shakespeare Programming Language. 2001. http://shakespearelang.sourceforge.net/.