Let's Take Esoteric Programming Languages Seriously

Jeremy Singer
University of Glasgow
Glasgow, United Kingdom
jeremy.singer@glasgow.ac.uk

Steve Draper
University of Glasgow
Glasgow, United Kingdom
steve.draper@glasgow.ac.uk

Abstract

Esoteric programming languages are challenging to learn, but their unusual features and constraints may serve to improve programming ability. From languages designed to be intentionally obtuse (e.g. INTERCAL) to others targeting artistic expression (e.g. Piet) or exploring the nature of computation (e.g. Fractan), there is rich variety in the realm of esoteric programming languages. This essay examines the counterintuitive appeal of esoteric languages and seeks to analyse reasons for this popularity. We will explore why people are attracted to esoteric languages in terms of (a) program comprehension and construction, as well as (b) language design and implementation. Our assertion is that esoteric languages can improve general PL awareness, at the same time as enabling the esoteric programmer to impress their peers with obscure knowledge. We will also consider pedagogic principles and the use of AI, in relation to esoteric languages. Emerging from the specific discussion, we identify a general set of 'good' reasons for designing new programming languages. It may not be possible for anyone to be exhaustive on this topic, and it is certain we have not achieved that goal here. However we believe our most important contribution is to draw more attention to the varied and often implicit motivations involved in programming language design.

CCS Concepts: • **Software and its engineering → Language types**; • **Social and professional topics → Cultural characteristics**.

Keywords: programming language design, esolangs, programming humor

ACM Reference Format:
Jeremy Singer and Steve Draper. 2025. Let's Take Esoteric Programming Languages Seriously. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759429.3762632

1 Introduction

Despite their name, *esoteric* programming languages are surprisingly popular in the general Computer Science community. This essay is based on a presentation we have given several times to audiences of academic Computer Scientists interested in programming language design; on each occasion we have encountered keen advocates of esoteric languages like Whitespace and Brainf***. However we have also detected a sense of shame—esoteric language usage is an unexpectedly common 'guilty secret'. In this work, we seek to analyse potential reasons for these attitudes to esoteric languages. We will argue that Computer Scientists need to take esoteric languages seriously.

Let's start with some definitions:

An *esoteric programming language* (hereafter contracted to *esolang*, following general convention) is 'a programming language designed to experiment with weird ideas, to be hard to program in, or as a joke, rather than for practical use'.[1] Temkin [61] characterizes an esoteric programming language as one that is 'intentionally ununsable, uncomputable or conceptual'. Another definition[2] is that 'esolangs are a class of languages made for reasons other than practical use'.

More generally, the adjective *esoteric* relates to philosophical doctrines and modes of speech. It refers to concepts or materials that are appreciated only by an 'inner circle' of advanced persons, known as initiates.

And, if a definition were needed, a *programming language* is a systematized notation for precise expression of intended computational behaviour. The ACM encyclopedia definition of 'programming languages' [53] describes them as: 'both tools for directing the operation of a computer and tools for organizing and expressing solutions to problems'.

The claim of this essay is that esoteric languages might be inherently useful in some significant ways, and so they deserve to be studied in a serious manner. The first question we seek to address is: *What is the appeal of esolangs, in general?* Our discussion will cover the following four high-level reasons for their popularity:

1. A sense of fun or playfulness
2. Recalling a lost 'golden age' of coding
3. Sense of cultic initiation
4. Artistic value

[1] https://esolangs.org/wiki/Esoteric_programming_language
[2] https://esoteric.codes/blog/esolangs-as-an-experiential-practice

The second question we will address is: *What is the appeal of designing esolangs, in particular?* We will explore the reasons of parody and comedy, before moving on to considering the skills of working within very tight self-imposed constraints, and the attraction of creating a programming language 'of one's own', as Virginia Woolf might have said.

Taking inspiration from Wing's conceptual framework of *Computational Thinking* [69], we will ask: *what do programmers actually do?* We will explore how this characteristic behaviour is manifested in the medium of esolangs. Whereas Wing's claim is that practitioners in other disciplines can benefit from a grasp of computing principles, our study of esoteric languages leads us to believe that computer programmers can benefit from a wider appreciation of other disciplines. This is Gordon's argument in his Onward! essay from last year [18] which takes the complementary stance to Wing, encouraging computer programmers to appreciate natural language and linguistics.

In considering pedagogic principles, we will ask *Why is it beneficial to expose learners to esoteric languages?* We will explore this question through the lens of variation theory, *inter alia.*

Although esolangs may seem trivial, they deserve our attention since they can change the way we think about programming and language design. As Perlis [47] puts it:

> Epigram 19: A language that doesn't affect the way you think about programming, is not worth knowing.

2 Background

In this section, we briefly survey the field of esolangs. We commence with the prototypical esolang, INTERCAL, in Section 2.1. After this, for the sake of convenience we cluster the languages into broadly related sets, considering esolangs that have a natural language syntax (Section 2.2), those with non-textual syntax (Section 2.3) and those with an exotic computational model (Section 2.4). We acknowledge that these characteristic properties are not mutually exclusive. Further, we acknowledge that other esolangs may not fall into any of these categories, but we hope this is sufficient for an imprecise 'first cut' at surveying the field.

One key observation is that almost all esolangs have low-level semantics, in terms of primitive arithmetic operations on integer values, limited input/output facilities and primitive control flow. Higher-level facilities such as abstract data types, modularity and concurrency features are almost entirely absent.

Another general observation is that the surface-level, concrete syntax of all esolang programs is highly unconventional. Typical esolang code does not look like mainstream code, indeed, it may not look like code at all [12].

2.1 INTERCAL

We first discuss INTERCAL, since this is the archetype of esolangs. To the best of our knowledge, its language reference manual [70] makes the first mention of 'esoteric' in relation to programming languages of this nature.

> The examples of INTERCAL programming which have appeared in the preceding sections of this manual have probably seemed highly esoteric to the reader unfamiliar with the language. With the aim of making them more so, we present here a description of INTERCAL.

Like many subsequent esolangs, INTERCAL was devised by university students as a playful, humorous artifact. It has become 'the longest-running and most convoluted joke in the history of programming language design' [51]. Although the language initially had a minimal compiler and negligible source code corpus, it caught the attention of a small group of dedicated enthusiasts including high profile Computer Scientists like Raymond [50] and Knuth [29] who helped to popularize it.

INTERCAL is not a well-formed acronym or abbreviation: confusingly[3] it stands for 'Compiler Language With No Pronounceable Acronym'. The language was invented in 1972 by two Princeton students as a parody of the verbose imperative languages of the day.

INTERCAL has unusual lexical elements, such as the use of the characters ¢ and $ as operators. There are also unconventional binary operations, such as bitwise interleavings of values. For these reasons among others, fairly standard arithmetic operations become complex and convoluted in INTERCAL.

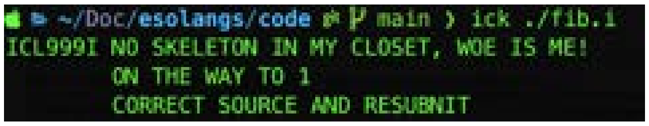
One of the (many) highly original features of INTERCAL is the `PLEASE` command modifier. The compiler requires occasional politeness, however excessive `PLEASE`s cause a compilation error. In a remarkable back-to-the-future moment, recent ChatGPT users have demonstrated similar politeness with AI prompts [71].

Bratishenko [6] identifies the philosophy of INTERCAL as 'technomasochism', since programmers working with this esolang seem to derive pleasure from their painful software development experience.

Error messages are highly unreadable, as Figure 1 illustrates, when the `ick` INTERCAL compiler is unable to parse a program correctly.

The C language version of the INTERCAL compiler improved its reach [50]. Now, INTERCAL is available on many open-source package management systems. For instance, you can `brew install intercal` on macOS or alternatively, `apt install intercal` on Ubuntu.

[3]The contraction is 'for obvious reasons', according to the language manual [70].



```
~/Doc/esolangs/code  main ❯ ick ./fib.i
ICL999I NO SKELETON IN MY CLOSET, WOE IS ME!
        ON THE WAY TO 1
        CORRECT SOURCE AND RESUBNIT
```

Figure 1. Typical INTERCAL compiler error message

We consider INTERCAL to be an archetypal esolang since it demonstrates foundational principles that define the genre, i.e. syntactic absurdity and a satirical stance toward conventional programming language design. Later esolangs share these playful characteristics. INTERCAL sparked amusement and curiosity among the broader software developer community, which continues to engage with esolangs more generally.

2.2 Natural Language-like Programming Languages

Programming in natural language was an early ambition for language developers. Initial high-level languages like COBOL reflect this drive, to some extent. Perhaps the most 'natural' programming language today is Inform7, used for interactive text adventure games [43].

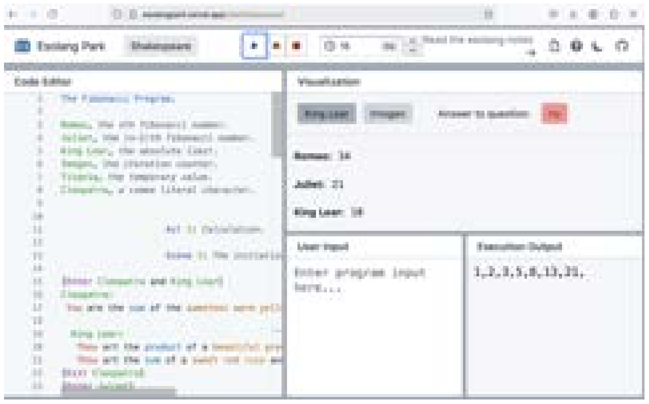
However, a large number of esolangs have program syntax that appears to be a structured form of natural language artifacts. Table 1 presents a few instances of this category of esolang.

For such languages, although the syntax is natural, the fixed structure of the artifact means that programs can be parsed in a straightforward manner. The underlying computational model is generally reasonable, involving integers, simple arithmetic operations, assignments to local variables, conditional tests and basic imperative control flow.

For example, Figure 2 shows the step-by-step execution of a Shakespeare fibonacci calculation. Characters are individual variables storing integer values. In Shakespeare, only two actors can be on the stage at once. Their dialogue involves assigning values to each other. The most complex part of Shakespeare is the representation of integer literal constants, involving complimentary and derogatory nouns (denoting +1 and -1 respectively) with every prefixed adjective doubling a value. So 'a sweet red rose' would be 2×2×1, whereas a 'poisonous toad' would be 2×-1.

Similarly in Chef, the ingredients are used to store literal values (with quantities) and these values are stored in variables, which are integer stacks, denoted as mixing bowls. The various actions of cooking (pour/blend/fold) involve computation on integer values in the bowls.

In general, these textual languages have primitive imperative features, often lacking facilities like dynamic memory allocation and compound data structures.



```
The Fibonacci Program.

Romeo, the nth fibonacci number.
Juliet, the (n-1)th fibonacci number.
King Lear, the absolute limit.
Banquo, the iteration counter.
Titania, the temporary value.
Cleopatra, a comma literal character.


                    Act I: Calculation.

                    Scene I: The initializ

[Enter Cleopatra and King Lear]
Cleopatra:
 You are the sum of the [illegible]

  King Lear:
   Thou art the product of a beautiful [illegible]
   Thou art the sum of a sweet red rose [illegible]
[Exit Cleopatra]
[Enter Juliet]
```

Romeo: 34

Juliet: 21

King Lear: 18

Enter program input here...

```
1,2,3,5,8,13,21,
```

Figure 2. Online Interpreter for Shakespeare programs

Figure 3. Set of mixing bowls, appropriate for the CHEF esolang

2.3 Non-Textual Syntax

While all esolangs feature unconventional syntax, some of them have entirely non-textual syntax. Two such languages are Whitespace and Piet.

Both these languages feature relatively straightforward abstract machines. They are stack based with a sensible instruction set that has appropriate low-level operations. In

Table 1. Esolangs based on structured natural language patterns

Esolang	Structure	Example assignment (x=2)	Example output (print x)
Shakespeare	Elizabethan theatrical script	Thou art a red rose	Open thy heart
Rockstar	1980s song lyrics	Let your love be 2	Shout your love
Chef	Cooking recipe	2g Chocolate	Serves 1
LOLCODE	Internet chat	I HAS A x R 2	VISIBLE x

each case, the only truly unusual aspect of the language is the syntax.

Whitespace has only three lexemes, all of which are whitespace characters, i.e. space, tab and newline. Various combinations of these lexemes make up instructions and literal data.

On the other hand, a *Piet* program is a bitmap image. There is a fixed palette of colours, each of which is available in three different shades. A two-dimensional program counter, known as the 'direction pointer' traverses the image from the top-left pixel and instruction execution takes place when the direction pointer crosses a colour boundary. The difference between colours and shades is used to perform a lookup in a two-dimensional opcode table, which has traditional stack-based operations. The executable properties of Piet mean that a program could be carefully recoloured or resized, without changing its semantics.

In these languages, programs deliberately don't look like programs. For Whitespace, code is invisible to the human eye. For Piet, code is a colourful image. This is a kind of steganography, hiding code in visual media.

2.4 Unconventional Computational Model

While all esolangs feature unconventional syntax, one group of esolangs combines this with unconventional abstract machines.

In theory, all of computation is built on the foundation of a strange machine, i.e. the Turing machine. As long as a computational framework can emulate the execution of a Turing machine, then it is Turing complete.

Mateas and Montford [39] identify *minimalist* esolangs, which have a small set of operations and closely resemble Turing's model of computation. The most well known language of this kind is Brainf*** (BF). The language has eight primitive operations (denoted by single characters) which mimic a Turing machine. There is a fixed length byte array, with operations to read and write to the current element of the array, to move left or right to an adjacent element, to jump to a different part of the program based on the value stored in the current element, and to do character I/O.

Because of its minimal nature, BF programs can be executed with a tiny interpreter. However most BF programs that perform 'interesting' tasks are very long (like Turing machines).

Turing's assumption of an infinite-length tape leads many programmers to assume storage space is infinite and that execution time may be unbounded. In contrast, BF imposes strict static limits on storage (generally the array is 30KB in size).

The Befunge esolang is somewhat similar to BF. Again, operations have a single character. Programs are laid out in a two-dimensional grid, rather than BF's one-dimensional line. Direction operators are used to change the direction of the instruction pointer. There is an auxiliary stack for intermediate computation.

Malbodge was intentionally designed to be the most difficult language in which to write code. It is named after Dante's eighth circle of Hell. The abstract machines uses ternary bits. An instruction's precise effect depends on its memory location. Code is automatically self-modifying. Effectively, a Malbodge program re-encrypts its source code after every instruction execution.

As may be expected, there are relatively few extant Malbodge programs and most of these are synthesized from search-based code.

Conway's Fractan language [10] is a purely mathematical esolang. A program starts with an initial integer value N and a fixed list of fractions $f_1, f_2, \ldots, f_k$. Conway explains the abstract machine semantics as follows:

> You repeatedly multiply the integer you have at any stage (initially N) by the earliest f_i in the list for which the answer is integral. Whenever there is no such f_i, the game stops.

He demonstrates a program that generates the sequence of prime integers, then another that computes successive digits of π. Conway criticizes other programming languages as having problems that 'stem from a bad choice of the underlying computational model'.

2.5 Discussion

It is commonly accepted that computer languages do not need to be Turing complete; for instance, some domain-specific languages are not. However, many esolangs are Turing complete. Generally, they fall into the category of *Turing tar-pits*—a phrase coined by Perlis [47].

> Epigram 54: Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Essentially, most programs are verbose and tricky to write. This was an explicit design goal of INTERCAL [70] and remains true for other esolangs.

Cox [12] considers various aspects of esolangs, presenting a philosophical and largely non-technical appreciation. We review further esolang literature in Section 8.

3 Appeal of Esoteric Languages

Esolangs have an enduring sense of appeal to many computer scientists, particularly programming language designers. In this section, we explore the reasons for this popularity. The characteristics we analyse have more to do with the nature of the programmer, than of the programming language. Although it's natural for one to reflect the other …

3.1 Playfulness

The stereotypical hacker (in the classical sense of a computing enthusiast) possesses a genuine sense of playfulness. Raymond [49] makes the following observation of this community of practice:

> We hackers are a playful bunch; we'll hack anything, including language, if it looks like fun …Deep down, we like confusing people who are stuffier and less mentally agile than we are, especially when they're bosses. There's a little bit of the mad scientist in all hackers, ready to discombobulate the world and flip authority the finger – especially if we can do it with snazzy special effects.

This playful anti-authoritarian attitude is perfectly captured by esolangs. On the surface, programs in such languages appear to be confusing and counterintuitive—highly 'discombobulating', as Raymond puts it.

One way to define 'play' is an activity which is defined by the activity alone, and is performed to discover the outcome, rather than to achieve a pre-defined goal. In this sense, the playfulness of esolangs is significant.

Really, esolangs are only appropriate for expressing 'toy' programs and simple algorithm implementations. For the most part, esolangs feature primitive IO, perhaps only supporting `putchar` and `getchar` functions with no higher-level libraries. As such, code written in esolangs is simple in terms of I/O, probably the kind of program required in coding competitions like the Code Olympics and the Advent of Code. This reinforces the sense of playfulness; many people attempt Advent of Code problems in esolangs. For instance, a google search for 'advent of code in brainf***' highlights a number of attempts, demonstrating a non-trivial intersection between people who like esolangs and people who enter coding competitions.

Intcode is an esolang designed explicitly for Advent of Code in 2019; really it is a low-level instruction set for a simple virtual machine.

We acknowledge that it is not a new educational idea to connect play and learning. For instance, the Logo turtle graphics language is a playful way for school-age learners to encounter geometry, maths and programming. Papert [46] introduces the notion of 'hard fun', which captures 'hard challenging things to do, and complex artifacts to play with'. Further, play is experiential learning by discovery [30]. To players, the *activity* is key and the *outcome* is not necessarily important. Again, these factors are evident in esolangs.

Esolangs preserve a 'sense of fun' in programming. Perlis, as quoted by Abelson and Sussman [1], states:

> I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun …I hope the field of computer science never loses its sense of fun.

3.2 Nostalgia

It is often the case that experienced developers enjoy wearing 'the hair shirt' [48] which involves putting themselves through difficult experiences while developing supposedly more elegant code.

This trend originated with Turing, who invented the notion of *optimal coding*, whereby he would situate instructions at precisely the best locations in memory to be executed with minimum delay in a rotary memory machine—the Pilot ACE had circulating mercury delay lines to store instructions. Whereas other early stored program electronic computers employed complex control circuitry to avoid needing to make these instruction placement calculations, Turing wanted to defer the complexity to the programmer. Campbell-Kelly records Turing's thinking about using hardware interlocks to avoid the need for optimal coding [8] as being:

> …much more in the American tradition of solving one's problems by means of much equipment rather than by thought.

Again, this attitude harmonizes neatly with the legendary story of Mel the Hacker from Royal McBee, recorded in Eric Raymond's jargon file [42], describing how Mel wrote optimal code that worked perfectly on a specific rotating memory machine due to the increment of a self-modifying instruction resulting in a jump at precisely the right moment in the program.

This idea that programming should be challenging for humans, requiring significant mental effort, is a common one. The proportion of time spent thinking, relative to compiling or executing code, used to be much higher. Of course, in

Figure 4. Visual Studio Code with the LOLCODE plugin in operation

the early days of computers, compilation was much more expensive, perhaps an overnight job. The use of esolangs raises the level mental activity required for programming. For instance, no program was written in the Malbodge language for several years after it was designed, and the first correct program was machine-generated.

The key takeaway here is the hacker instinct that programming ought to be difficult, and esolangs enforce this. By contrast, coding in mainstream languages is becoming progressively simpler. Tooling support is another reason why coding in mainstream languages is relatively straightforward. Many new developers use integrated development environments like Visual Studio Code, featuring syntax highlighting, autocorrection, large language model (LLM) hints, static checking etc. Such modern IDEs have minimal support for esolangs: LOLCODE on VS Code is a notable exception, featuring a syntax highlighting plugin—see Figure 4 for instance. Mostly developers working with esolangs just use a text editor and terminal compilation.

Further, many mainstream managed language runtimes feature dynamic safety checks, e.g. for out-of-bounds memory accesses. Even traditionally unsafe languages, like C, can be executed with address sanitizers or binary instrumentation to detect bugs. Esolang runtimes do not have these safety checks, by default.

Another issue relates to AI-generated code. LLMs are notoriously helpful at generating reasonable code for mainstream languages. However AI synthesized code is less successful for esolangs, see Section 7.

The upshot of all of this is, for programmers hankering for the lost 'golden age' of programming, when there were no fancy development environments or AI assistants, esoteric languages represent a viable way to return to this time and relive this experience. In some sense, this is a *pre-Raphaelite* attitude to programming, with value placed on complexity and attention to detail.

3.3 Sense of Belonging

Language serves as a powerful marker of community and group identity. Fluency in a language often signifies inclusion within a particular discourse community. This is true not only for natural languages but also for constructed ones—consider, for instance, Creole languages or fictional tongues like Klingon. In all these scenarios, language functions as a social bond, connecting individuals through shared forms of expression and cultural identity.

In recent programming practice, there has been a growing emphasis on localization—developing notations that reflect a range of ethnically diverse languages as a means of promoting inclusivity [58]. Some esoteric programming languages also engage with this idea of cultural identity. For example, Cree# incorporates vocabulary from Nehiyawewin, the Plains Cree language, offering a unique intersection between programming and Indigenous linguistic heritage [11].

To the novice programmer, unfamiliar syntax and abstract concepts can feel confusing and discouraging. In contrast, experienced programmers often embrace this complexity, finding enjoyment and stimulation in the challenge. What may seem obscure or unconventional to a beginner can appear refreshingly creative and original to a seasoned practitioner [12]. Esoteric programming languages, in particular, foster a sense of community—not necessarily through fluency in a single language, but through a shared appreciation for playful, cryptic syntax and unconventional algorithmic expression. This shared enthusiasm hints at a kind of cultic initiation, where an 'inner circle' of programmers engages with code not merely as a practical tool, but as a source of amusement, aesthetic pleasure, and intellectual play.

Another key concept in esoteric practice is the idea of *transmission*, which is considered a critical feature of such traditions [15]. Unlike mainstream languages, which are typically acquired via textbooks and formal instruction, esolangs are learned through more elusive means: scattered resources, word-of-mouth exchanges, and personal initiation. This difficulty in access not only shapes the learning process but also contributes to a sense of mystique, reinforcing the allure of the esoteric.

Nevertheless, the advent of the internet means it is now more straightforward to build a community around a constructed language [17].

3.4 Artistic Expression

In his *Mythical Man Month*, Brooks [7] refers to the 'sheer joy of making things' as the primary motivation for the creative acts of programmers.

Every program is an artifact, which may be viewed from different perspectives. Lonati et al [37] enumerate six facets of programs as artifacts, dividing these into three pairs of contrasting features, namely:

1. notational artefact *versus* executable entity

2. human-made *versus* tool
3. abstract entity *versus* physical object

Programs in esolangs emphasize:

1. the *notational* artefact, given the weird syntax
2. the *human-made*, given the ingenuity to construct the program
3. the *abstract entity*, given some higher-level semantics for the abstract machine of the particular esolang

Knuth's literate programming [28] conveys a deep truth which almost all programming language creators have ignored—code is written for two utterly different readers: the compiler which acts physically on the code, and the human who reads the code to understand it.

Another perspective, acknowledged by Knuth, is that code has an *aesthetic* value. Source code may be appreciated as art, cf. 'The Art of Computer Programming'. In this case, the code itself is an artifact: generally (although not always) a visual object that can be enjoyed directly, by the viewer.

There are some exhibits of *software as art*, for example at the Programming Language History museum.[4]. However it's not clear how much of these artistic displays are source code. Esoteric languages may be considered as an art form in some circles [63].

We might also mention 'live coding', where the programming itself is generating art (e.g. live music [4]). This is usually performative art, rather than artifact creation. More generally, usable live programming involves fluid code editing with immediate feedback [40].

If artifacts can be interpreted as code, e.g. bitmap images as Piet programs, or 16th Century English plays as Shakespeare programs, can we interpret arbitrary artifacts as code? How does the real text of 'Romeo and Juliet' compute? Or is this a nonsensical question to ask? Certainly valid Shakespeare programs respect implicit syntactic restrictions, such as allowing only two actors on-stage at any given time.

The idea of a program having multiple interpretations was demonstrated by Worth, who wrote a Chef program that can be interpreted as a 'Hello world' program and also interpreted as a viable cake recipe.[5]

4 Pedagogic Value

In earlier sections, we hinted at the value of esoteric languages with respect to learning programming. As already discussed, Papert's notion of 'hard fun' is relevant here—esoteric languages are intriguing and may engage learners in different ways to grapple with the challenges of programming. We learn when we are having fun, and there is no need to be embarrassed about this.

Early exposure to multiple programming languages is widely considered to be beneficial to novice programmers [13]. Seeing the same underlying concept (e.g. a counted loop) expressed in differing contexts is conducive to learning; this is the basis of variation theory [64].

Concept transfer from one programming language to another is extremely important, helping students to enrich their notional machine models. Identifying true- and false-carry over concepts is highly relevant for mainstream languages, often when they have broadly similar syntax [66]. For esoteric languages with unconventional syntax, instead the deep underlying conceptual transfers are enforced, which only have semantic (not syntactic) correspondence.

There are concrete instances of esolang usage in education. Chef was used in the 'LCC 2700 Introduction to Computational Media' course at Georgia Technology from 2005 onwards.[6]

There are many other beneficial learning outcomes when novice programming students are exposed to esolangs.

One obvious learning outcome is the reinforcement of Dijkstra's dictum that `GOTO` is 'considered harmful' [14]. For instance, some esolangs only have a `GOTO` style construct, such as Shakespeare's *Let us proceed to Scene N.* Shakespeare code is sadly often spaghetti code.

Another positive learning outcome is the realisation that syntax is merely the surface form of a program, and can easily be altered. The bizarre or unexpected nature of much esolang syntax foregrounds the fact that the surface 'look-and-feel' of code is flexible. Learners begin to understand that what really matters is the underlying structure and semantic content behind the syntax. This liberates novice programmers from being overly attached to any one language (language tribalism), instead promoting a more conceptual, language-agnostic approach to software development.

Many esolangs pare away the conveniences of high-level languages and expose directly the underlying computational model. The abstract machine may be some variant of a Turing machine, or some alternative. The programmer is forced to consider how control flow, program state, and logic work at a low level. This understanding is highly valuable, as it concretizes what's happening when programs execute.

There are additional learning outcomes for more advanced (i.e. non-novice) students.

For instance, most esoteric languages are of the size and scale that would be suitable for a toolchain implementation as a student project. Students would have a more rounded education if they designed an esolang and then implemented it, using appropriate language engineering tools. This 'toe-in-the-water' experience would be relatively straightforward to support, and would introduce learners to the issues around what makes a good language. The key critical thinking exercise here would introduce students to at least a few basic ideas on how assess what is desirable in a programing language.

[4]https://spectrum.ieee.org/art-of-code
[5]https://web.archive.org/web/20230501044254/https://www.mike-worth.com/2013/03/31/baking-a-hello-world-cake/
[6]https://www.dangermouse.net/esoteric/chef.html

Finally, for programming educators, esolangs offer the opportunity to experience similar levels of frustration as the average novice programmer does with a mainstream language. You might be adept at expressing quicksort in a handful of lines of Haskell, but what happens when you try to write a sorting algorithm in an esolang? We might be able to enhance our empathy as educators, as we relive the agony of debugging cryptic fragments of programs we hardly understand.

So, at first glance esolangs might seem silly or impractical, but they offer serious educational value. They demonstrate that programming is not just about learning syntax or toolchains, but about reflecting critically about what we do when we write programs.

5 Designing New Esoteric Languages

In this section, we discuss the design and development of esolangs. First we examine the kinds of people who create new esolangs (Section 5.1) and then we move on to consider their reasons for creating such languages (Section 5.2).

5.1 Who Develops Esolangs?

5.1.1 Mostly Students? Many esolangs were designed by Computer Scientists during their university studies. For instance, the INTERCAL language was created by two Princeton University students: Woods and Lyon. The Whitespace language was created by two undergraduates at Durham University: Brady and Morris. Further, the Shakespeare language was created by students Åslund and Wiberg as a coursework project submission in their Compilers course at Royal Institute of Technology in Stockholm.[7]

These instances demonstrate that the simplicity of esolangs means their implementation is tractable as a one- or two-person programming exercise, for advanced undergraduate students.

Following Brooks' 'plan to throw one away' principle of software engineering [7], some student esolang developers have proceeded to develop more serious, mainstream languages in later life. For example, Brady (of Whitespace fame) later invented *Idris*, a dependently-typed functional language [5].

Similarly, Morgan-Mar is a serial inventor of esolangs, including Chef and Piet.[8]

5.1.2 Mostly Males? Similarly to common perceptions regarding mainstream languages, it seems that the majority of esolang designers are males. This is based on a cursory analysis of designer names, with some implicit cultural assumptions. A more formal analysis would be required to verify this claim. However it generally concurs with the work of Hermans on feminism in programming languages [21], as she notes that 'exclusion of women from technology is both a cause and a result of a gendered interpretation of what programming is and what contributions matter'. Hermans notes that valuing 'hard' and employing 'difficult tools that not everyone can use' are both male-dominant characteristics, which again, might align with the prevailing trends of esolangs.

Beckwith and Burnett [2] identify how male and female developers approach end-user programming environments in different ways. For instance, low self-efficacy (less common in males) attributes failure in difficult tasks to personal lack of ability. This effect might be exacerbated in the complex syntactic surfaces of esolangs.

On the other hand, alternative programming paradigms, such as *programming as story telling* has been shown to appeal more to female students [26]. It is certainly the case that some esolangs (particularly the textual ones where programs don't look like programs) follow the story telling pattern, in some sense.

5.2 Why Develop an Esolang?

5.2.1 Comedy. The most commonly asserted reasons for developing an esolang involve humor.

INTERCAL is essentialy a parody of mainstream languages from the 1970s. The courteous `PLEASE DO` command is a not-so-subtle criticism of the imperative nature of FORTRAN and the verbose syntax of COBOL. The INTERCAL manual [70] is written in a breezy, comedic style. For example, the first paragraph states: 'Any resemblance of the programming language portrayed here to other programming languages, living or dead, is purely coincidental'.

The Whitespace language was publicly announced as an April Fool's joke on the Slashdot developer news site on 1 April 2003. The language itself was an implementation of an original idea suggested by Stroustrup as an April Fool some five years earlier [57].

From these two cases, we acknowledge that esolangs are not particularly serious in general. The online web comic *xkcd* also features esolangs, see Figure 5.

Tiwari et al. [65] explore how humor is great motivation for software developers. Further, Kuutila et al. [32] analyse reddit posts to characterize programming humor. They show that 'things that violate accepted patterns in a non-threatening way are perceived to be humorous'—attributes that clearly apply to esolang programs.

However the sheer number of esolangs and the efforts involved in supporting them lead us to believe that there are other, deeper motivations for esolang development.

5.2.2 Challenge. Next we consider the skills of working within very tight self-imposed constraints, which are stock-in-trade for the experienced developer.

[7] https://web.archive.org/web/20220721085340/http://shakespearelang.sourceforge.net/report/shakespeare/shakespeare.html
[8] https://www.dangermouse.net/esoteric/

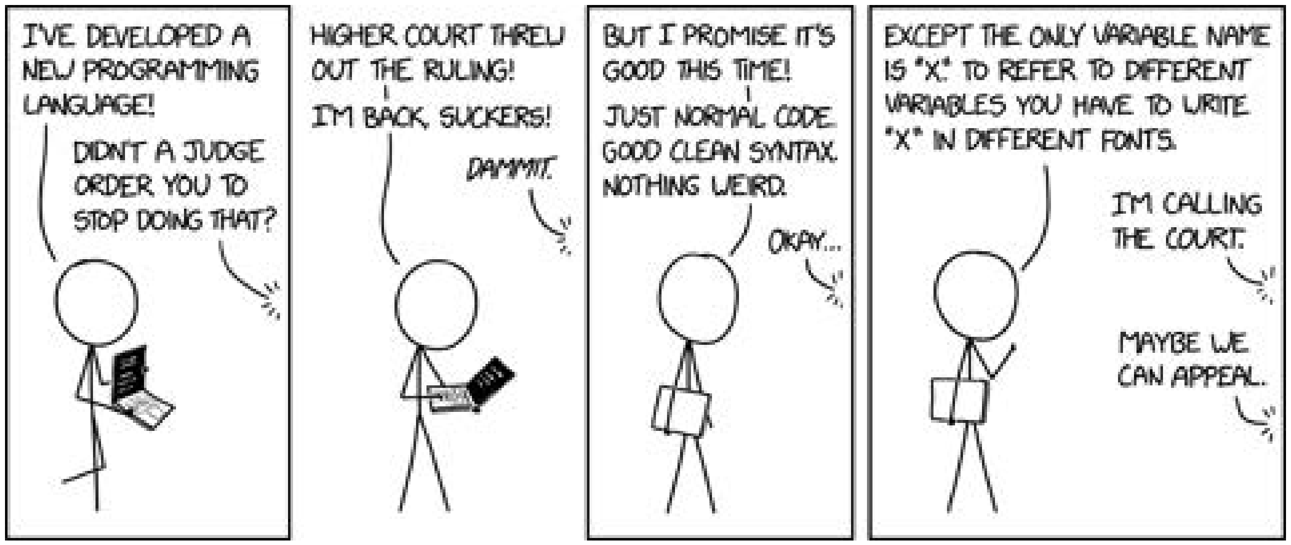

I'VE DEVELOPED A NEW PROGRAMMING LANGUAGE!
DIDN'T A JUDGE ORDER YOU TO STOP DOING THAT?
HIGHER COURT THREW OUT THE RULING! I'M BACK, SUCKERS!
DAMMIT.
BUT I PROMISE IT'S GOOD THIS TIME! JUST NORMAL CODE. GOOD CLEAN SYNTAX. NOTHING WEIRD.
OKAY...
EXCEPT THE ONLY VARIABLE NAME IS "X." TO REFER TO DIFFERENT VARIABLES YOU HAVE TO WRITE "X" IN DIFFERENT FONTS.
I'M CALLING THE COURT.
MAYBE WE CAN APPEAL.

Figure 5. Hypothetical esolang suggestion from xkcd.com webcomic, number 2309 (CC BY-NC 2.5 Randall Munroe)

For esoteric languages, programs need to be written differently, in an unconventional syntax and structure. Constraints are ever-present, and in some cases programs are practically impossible to write, e.g. for the Malbodge language. We discussed these issues earlier, in Section 3 when we were considering the attraction of writing code in an esolang. Similar challenges will apply to develop an esolang, whether or not that language is self-hosting.

Developers report that challenging problems are a key intrinsic motivation for their productivity [19]. However again, this aspect on its own does not seem to account for the wide variety of esolangs and the intense development effort.

5.2.3 Creativity. Finally, there is the attraction of creating a programming language 'of one's own'. Constructing an esolang permits the programmer to enjoy the luxury of a 'break from the bland, bloated multiparadigm languages with imperative roots' [27].

This facility of expression, of 'speaking code' [12], is a powerful motivation for people to generate their own, custom esolangs. As we note in Section 4, creating an esolang gives excellent opportunity for students to reflect on what make a good programming language, and to identify trade-offs in language design.

After all, it feels as if esolangs are probably designed in exactly the same way as many widely used languages, i.e. for the convenience of the language designer rather than of the end-user programmer.

Having considered the reasons why people develop esolangs, which are mostly to do with creative exploration and experimentation, in the next Section we will move into a more general discussion about motivations for designing any programming language.

6 Reasons for Developing a New Programming Language

As we study the field of esoteric programming languages, we are confronted with a question that relates to more general programming language philosophy: *Why does a person choose to design and develop a new programming language?* Can we identify a set of overarching motivations for the construction of new programming languages? Here we are using our analysis of esolangs to reflect on wider aspects of programming language design.

In this section, we suggest five generic themes pertaining to programming language design motivation:

1. Expressivity
2. Efficiency
3. Education
4. Economy
5. Exploration

We consider each of these themes in detail below, providing example mainstream programming languages in each case. We recognize that some languages may have multiple design motivations.

6.1 Expressivity

Steele [56] beautifully illustrates the notion of 'growing a language' with increasing expressive power, which enables succinct description of complex concepts.

One key motivation for developing a new language is the need for expressive power. Of course, computability theory tells us we could all write in Church's λ-calculus and express any computation we like, but programs would be very long and unwieldy, and programming would be complex and unenjoyable, if not intractable. More expressive notational systems allow higher-level programs to be written. This was the original motivation for early high-level languages like Fortran, Cobol and Lisp. This is still the motivation for more modern high-level languages like Clojure or Prolog.

Jones and Bonsignour note that a programmer can write a fairly constant number of lines of code per hour, regardless of the language used [23]. So a more expressive language should enable programmers to be more productive. They conclude that 'the overall effort associated with coding and testing are much less significant for high-level ... languages than for ... low-level languages'.

6.2 Efficiency

Low-level systems implementation languages like C were developed, at least originally, to map closely to the underlying hardware and enable efficient code generation. As Ritchie puts it, C is 'a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines' [52]. C has an enduring popularity [25].

More modern systems languages like Rust and Zig also explicitly aim for efficient execution of target code as a primary goal.

6.3 Education

Another reason for developing a new language is for the purposes of programming education. Languages designed explicitly for novice learners include Grace [3] although this has not been adopted widely. More venerable examples of educational languages include Logo and BASIC. Scratch and Alice are instances of graphical languages with non-traditional syntax for school-age learners. Hedy [20] is a recent programming language that is aimed at novice learners, which has a gradually increasing syntax and semantics, to accommodate an incremental concept inventory and learning style.

However, it is noteworthy that no mainstream industrial strength languages[9] were explicitly designed for educational use, or with respect to pedagogical principles [31]. Further, initially simple industrial languages accumulate new features throughout their lifetime, generally measured in decades [16].

6.4 Economy

Some languages are commercial imperatives—needed to satisfy economic ends. The language that springs to mind most readily is C#, originally conceived as a 'direct rival' to the Java ecosystem, which seemed to be a 'deep challenge to Microsoft' [59].

[9]One reviewer kindly pointed out that Pascal might be an exception here.

6.5 Exploration

Some programming languages are designed primarily as an intellectual exercise, seeking to explore a new area or evaluate a new concept. The Haskell language [22] may well fall into this category. In his HOPL presentation of this paper, Peyton Jones explains how Haskell was intended to 'avoid success at all costs' which means the relatively small user-base are happy for the language to be 'nimble' in terms of feature updates.

6.6 Esolang Motivations

Earlier, in Section 5.2, we discussed why people are motivated to design and develop new esolangs. In this section, we want to locate those identified motivations within the five point hierarchy of programming language design motivation outlined above (the five 'E's).

For many of the motivations, it feels like esolangs have diametrically opposed objectives.

In terms of *expressivity*, esolangs deliberately use obscure syntax and limited constructs, making it difficult to express many algorithms in a meaningful and elegant manner. An esolang pares down the available operations to a minimal set, unlike most mainstream languages.

In terms of *efficiency*, while esolang semantics frequently map directly onto a low-level computational model, such a model is generally quite distinct from mainstream computer architecture, making execution less efficient.

In terms of *education*, esolangs are explicitly *not* designed for novice programmers. For instance, the INTERCAL language manual [70] states: 'Since it is an exceedingly easy language to learn, one might expect it would be a good language for initiating novice programmers. Perhaps surprising, than, is the fact that it would be more likely to initiate a novice into a search for another line of work'.

In terms of *economy*, esolangs are not commercially viable in any sense. No meaningful software is developed in esolangs and no esolang toolchain is available on a commercial basis. Perhaps the only people who might benefit financially from esolangs are book authors and publishers. For instance, MIT Press has a current esolang title available [12] and another one forthcoming [63].

Therefore, it seems that the majority of esolangs fall into the curiosity-driven, *exploratory* category. They seek to experiment with syntax or semantics in unconventional ways, pushing the boundaries of programming in some sense.

7 AI and Esolangs

No research work is complete nowadays without a consideration of the implications of AI on the topic. In this section we briefly discuss how AI might be useful for esolangs.

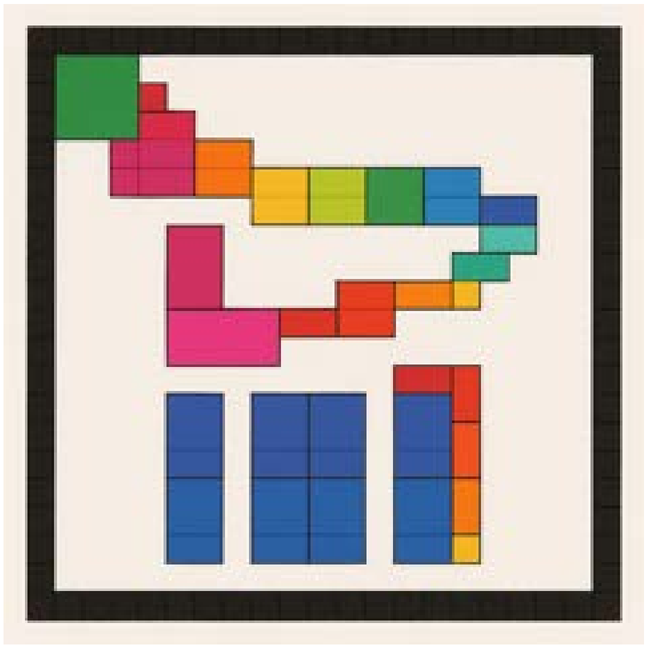


Figure 6. AI-synthesized Piet Fibonacci program, which contains syntax errors

7.1 Esolang Code Synthesis

Could an AI-based large language model (LLM) synthesize correct code to solve a specified problem in a given esolang? On the one hand, LLM-generated code seems to work well for mainstream languages [34, 35]. On the other hand, our initial experiments suggest that ChatGPT and Llama.cpp do not create syntactically correct code for esolangs including INTERCAL, Shakespeare, Whitespace and Piet. While the textual conversation surrounding the generated code sounds plausible, the code itself generally does not compile and often appears to contain fundamental syntactic flaws. For example, Figure 6 shows a Piet program generated by ChatGPT, responding to our prompt to 'produce a Piet program to calculate the first 10 Fibonacci numbers'. The resulting image cannot be parsed by the `npiet` interpreter.

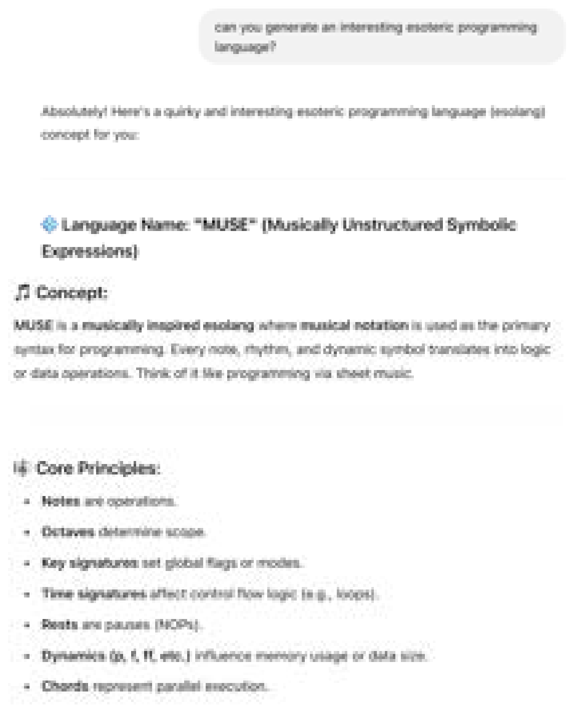
One reason why AI might be poor at synthesizing esoteric code is the limited training set of online code examples, in contrast to mainstream languages like Python and JavaScript [67, 68].

AI will not 'render software engineering irrelevant' [24]. This is notably true for software development in esolangs. The obscurity of esolangs means they are significantly under-represented in most LLM training corpora.

7.2 Esolang Creation

Could an AI model generate an interesting esolang, either a definition or a language implementation? We conducted some brief experiments with ChatGPT, but the results were not particularly impressive. Generated languages were not Turing complete, i.e. they did not define control flow properly. Further, example programs were not well-formed. Figure 7 shows an example of a constructed language with a syntax based on musical notation but the language definition is both incomplete and ambiguous. Further, this idea is highly unoriginal since a music-based esolang is already available, named *Velato* [60].

While there are some online tools for synthesizing constructed languages (e.g. https://vulgarlang.com for fantasy novels), these work based on user-defined parameters and employ a pseudo-random algorithm (rather than an AI model) which incorporates a definite notion of linguistic grammar.



can you generate an interesting esoteric programming language?

Absolutely! Here's a quirky and interesting esoteric programming language (esolang) concept for you:

Language Name: "MUSE" (Musically Unstructured Symbolic Expressions)

Concept:

MUSE is a **musically inspired esolang** where **musical notation** is used as the primary syntax for programming. Every note, rhythm, and dynamic symbol translates into logic or data operations. Think of it like programming via sheet music.

Core Principles:

- **Notes** are operations.
- **Octaves** determine scope.
- **Key signatures** set global flags or modes.
- **Time signatures** affect control flow logic (e.g., loops).
- **Rests** are pauses (NOPs).
- **Dynamics (p, f, ff, etc.)** influence memory usage or data size.
- **Chords** represent parallel execution.

Figure 7. Excerpt of user interaction to develop an example esolang with ChatGPT

8 Related Work

Landin discusses the 'next 700 programming languages' [33], optimistically stating that 'we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction'. Every esolang is a 'laboriously devised construction', in contrast to Landin's best intentions.

Chatley et al. [9] envision the 'next 7000 programming languages' and discuss the reasons for language longevity. Esolangs exhibit none of the features required for longevity, albeit many of them are long-lived.

Mashey explores why programing languages 'succeed' [38]. However again, the reasons do not seem to be applicable to esolangs in general.

In the Onward! venue, there has been much discussion over the years regarding the nature of programming languages [44, 45] with similar discussion at HOPL [54].

Steele and Gabriel [55] present 50 languages in 50 minutes: this live performance has taken place at a number of conference venues over the past few years. Their selection of languages include a number of esolangs such as Shakespeare (Computational Drama), Befunge (Stack Machines), and Piet (Visual Languages). It is striking that Steele, a seasoned and successful mainstream language designer, finds esolangs to be worthy of attention.

McIver [41] lists seven design failures that would impair the efficacy of novice programming languages. Esolangs commit all seven 'deadly sins', from syntactic complexity to expectation violation.

Mateas and Montford [39] discuss the sociological and phenomenological issues around esolangs. They link this community with the International Obfuscated C Code Contest (IOCCC), which is a celebration of unconventional code expressed in a mainstream programming language (i.e. C) but where the syntax and semantics of the language are highly abused in order to generate code that looks unusual (some kind of ASCII art, generally) and does something unexpected when compiled (some kind of textual output, normally). Cox [12] explores various non-technical aspects of esolangs in further depth. Temkin [61] gives the most complete review of the esolang culture and philosophy. He seeks to explore the nature of esolangs, and the motivation for their design and use. His findings broadly concur with ours. His more recent study [62] posits that esolangs enable 'personal expression and . . . elegance within chaos' and that they 'challenge base assumptions of who languages are designed for and how they should be used'. Esolangs favour individual personality over practicality, and expression over comprehension.

More generally, inspired by Gordon's work relating programming languages to linguistics [18], we see strong linkage between esolangs and the constructed languages evident in modern language study—from Elvish to Esperanto [36].

9 Conclusion

Esoteric languages are, by their very nature, less accessible than mainstream programming languages. This goes against the prevailing trend, in terms of improving the accessibility of computing—whether from a usability or diversity perspective. So esolangs are implicitly contrarian, set against the direction of mainstream philosophy and advocacy.

As we have looked at esolangs, we have endeavoured to learn something about the essence of programming, but in fact, we appear to have uncovered something about the essence of programmers. . .

Precisely what? That they are a strange bunch? It appears that esolang users and developers derive perverse pleasure from the constraints and complexities of esolangs, engaging in the 'technomasochism' identified by Bratishenko [6]. Perhaps this observation does not apply to all esolang programmers, but certainly it covers a significant fraction.

Ultimately, the use and development of esolangs is a form of *freedom of expression* [12]. Such freedom must be preserved and valued by the community. Even if that freedom of expression allows people to express themselves in highly limited ways, the freedom to be so constrained must still be preserved.

The key point of esolangs is that they enable exploration and experimentation in a manner that permits a 'break from the bland, bloated multiparadigm languages with imperative roots' [27]. Esolangs challenge critical thinking ability regarding both programming language design and the nature of programming. Finally, a restatement of Perlis' epigram is appropriate [47]:

> Epigram 19: A language that doesn't affect the way you think about programming, is not worth knowing.

For this reason above all others, esolangs are clearly 'worth knowing'.

Acknowledgments

We would like to thank the following people who gave us useful additional comments which improved this paper: Alan Blackwell, Norman Gray, and John Levine. This work was supported in part by UKRI/EPSRC under grant EPSRC M4Secure.

References

[1] Harold Abelson and Gerald Jay Sussman. 1996. *Structure and interpretation of computer programs.* MIT Press.

[2] L. Beckwith and M. Burnett. 2004. Gender: An Important Factor in End-User Programming Environments?. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. 107–114. doi:10.1109/VLHCC.2004.28

[3] Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking grace: a new object-oriented language for novices. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. 129–134. doi:10.1145/2445196.2445240

[4] Alan F Blackwell and Nick Collins. 2005. The Programming Language as a Musical Instrument.. In *PPIG*. 11. https://www.ppig.org/files/2005-PPIG-17th-blackwell.pdf.

[5] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. doi:10.1017/S095679681300018X

[6] Lev Bratishenko. 2009. Technomasochism: Getting spanked by INTERCAL. *Cabinet Magazine* 36 (2009). https://www.cabinetmagazine.org/issues/36/bratishenko.php.

[7] Frederick P Brooks Jr. 1995. *The mythical man-month (anniversary ed.).* Addison-Wesley.

[8] Martin Campbell-Kelly. 2012. Alan Turing's other universal machine. *Commun. ACM* 55, 7 (July 2012), 31–33. doi:10.1145/2209249.2209277

[9] Robert Chatley, Alastair Donaldson, and Alan Mycroft. 2019. The next 7000 programming languages. *Computing and software science: State of the art and perspectives* (2019), 250–282. doi:10.1007/978-3-319-91908-9_1

[10] John H Conway. 1987. Fractran: A simple universal programming language for arithmetic. In *Open problems in Communication and Computation*. Springer, 4–26.

[11] Jon Corbett. 2023. Cree Coding. Available from https://pinnguaq.com/stories/cree-coding/.

[12] Geoff Cox and Alex McLean. 2012. *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press. doi:10.7551/mitpress/8193.001.0001

[13] Paul Denny, Brett A. Becker, Nigel Bosch, James Prather, Brent Reeves, and Jacqueline Whalley. 2022. Novice Reflections During the Transition to a New Programming Language. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*. 948–954. doi:10.1145/3478431.3499314

[14] Edsger W Dijkstra. 1968. Go to statement considered harmful. *Commun. ACM* 11, 3 (1968), 147–148.

[15] Antoine Faivre. 1994. *Access to Western esotericism*. SUNY Press.

[16] J.-M. Favre. 2005. Languages evolve too! Changing the software time scale. In *Eighth International Workshop on Principles of Software Evolution*. 33–42. doi:10.1109/IWPSE.2005.22

[17] Federico Gobbo et al. 2005. The digital way to spread conlangs. *Language at Work: Language Learning, Discourse, and Translation Studies in Internet* (2005), 45–53.

[18] Colin S. Gordon. 2024. The Linguistics of Programming. 162–182. doi:10.1145/3689492.3689806

[19] Tracy Hall, Helen Sharp, Sarah Beecham, Nathan Baddoo, and Hugh Robinson. 2008. What Do We Know about Developer Motivation? *IEEE Software* 25, 4 (2008), 92–94. doi:10.1109/MS.2008.105

[20] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 259–270. doi:10.1145/3372782.3406262

[21] Felienne Hermans and Ari Schlesinger. 2024. A Case for Feminism in Programming Language Design. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 205–222. doi:10.1145/3689492.3689809

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. 12–1–12–55. doi:10.1145/1238844.1238856

[23] Capers Jones and Olivier Bonsignour. 2011. *The Economics of Software Quality*. Addison-Wesley Professional.

[24] Eunsuk Kang and Mary Shaw. 2024. tl;dr: Chill, y'all: AI Will Not Devour SE. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 303–315. doi:10.1145/3689492.3689816

[25] Stephen Kell. 2017. Some were meant for C: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 229–245. doi:10.1145/3133850.3133867

[26] Caitlin Kelleher. 2008. Using Storytelling to Introduce Girls to Computer Programming. In *Beyond Barbie and Mortal Kombat: New Perspectives on Gender and Gaming*. MIT Press. doi:10.7551/mitpress/7477.003.0022

[27] Ronald Kneusel. 2022. *Strange Code: Esoteric Languages That Make Programming Fun Again*. No Starch Press.

[28] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[29] Donald E. Knuth. 2011. Chapter 7: TPK in INTERCAL. In *Selected Papers on Fun and Games*. Center for the Study of Language and Information, Stanford, California.

[30] David A Kolb. 2014. *Experiential learning: Experience as the source of learning and development*. FT press.

[31] Michael Kölling. 2024. Principles of Educational Programming Language Design. *Informatics in Education-An International Journal* 23, 4 (2024), 823–836. doi:10.15388/infedu.2024.29

[32] Miikka Kuutila, Leevi Rantala, Junhao Li, Simo Hosio, and Mika Mäntylä. 2024. What Makes Programmers Laugh? Exploring the Submissions of the Subreddit r/ProgrammerHumor.. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 371–381. doi:10.1145/3674805.3686696

[33] Peter J Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.

[34] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158

[35] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[36] Joseph Lo Bianco. 2004. Invented languages and new worlds. *English Today* 20, 2 (2004), 8–18. doi:10.1017/S0266078404002032

[37] Violetta Lonati, Andrej Brodnik, Tim Bell, Andrew Paul Csizmadia, Liesbeth De Mol, Henry Hickman, Therese Keane, Claudio Mirolo, and Mattia Monga. 2022. What We Talk About When We Talk About Programs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*. 117–164. doi:10.1145/3571785.3574125

[38] John R. Mashey. 2004. Languages, Levels, Libraries, and Longevity: New programming languages are born every day. Why do some succeed and some fail? *Queue* 2, 9 (Dec. 2004), 32–38. doi:10.1145/1039511.1039532

[39] Michael Mateas and Nick Montfort. 2005. A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. In *Proceedings of the 6th Digital Arts and Culture Conference* (IT University of Copenhagen). 144–153. https://nickm.com/cis/a_box_darkly.pdf

[40] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 53–62. doi:10.1145/2509578.2509585

[41] L. McIver and D. Conway. 1996. Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. 309–316. doi:10.1109/SEEP.1996.534015

[42] Ed Nather. 1983. The Story of Mel. Available from http://www.catb.org/jargon/html/story-of-mel.html.

[43] Graham Nelson. 2006. *Natural Language, Semantic Analysis And Interactive Fiction*. 141–188. https://www.ifarchive.org/if-archive/books/IFTheoryBook.pdf.

[44] James Noble and Robert Biddle. 2023. programmingLanguage as Language. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 191–204. doi:10.1145/3622758.3622885

[45] Dominic Orchard. 2011. The four Rs of programming language design. In *Proceedings of the 10th SIGPLAN symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 157–162. doi:10.1145/2089131.2089138

[46] Seymour Papert. 2002. Hard Fun. Available from http://www.papert.org/articles/HardFun.html.

[47] Alan J. Perlis. 1982. Special Feature: Epigrams on programming. *SIGPLAN Notices* 17, 9 (Sept. 1982), 7–13. doi:10.1145/947955.1083808

[48] Simon Peyton Jones. 2003. Wearing the hair shirt: a retrospective on Haskell. (January 2003). https://www.microsoft.com/en-us/research/

publication/wearing-hair-shirt-retrospective-haskell-2003/ invited talk at POPL 2003.

[49] Eric Raymond. 2000. Guest Editorial: World Domination. *Linux Journal* (Jan. 2000). https://www.linuxjournal.com/article/3676

[50] Eric S Raymond. 1990. INTERCAL. https://gitlab.com/esr/intercal.

[51] Eric S Raymond. 2010. Risk, Verification, and the INTERCAL Reconstruction Massacree. http://esr.ibiblio.org/?p=2491.

[52] Dennis M. Ritchie. 1996. The development of the C programming language. In *History of Programming Languages—II*. 671–698. doi:10.1145/234286.1057834

[53] Jean E. Sammet and David Hemmendinger. 2003. *Programming languages*. John Wiley and Sons Ltd., 1470–1475.

[54] Mary Shaw. 2022. Myths and mythconceptions: what does it mean to be a programming language, anyhow? *Proc. ACM Program. Lang.* 4, HOPL, Article 234 (April 2022). doi:10.1145/3480947

[55] Guy Steele and Richard Gabriel. 2008. 50 in 50. http://lambda-the-ultimate.org/node/3101.

[56] Guy L Steele. 1999. Growing a language. *Higher-order and symbolic computation* 12, 3 (1999), 221–236. doi:10.1023/A:1010085415024

[57] Bjarne Stroustrup. 1998. Generalizing Overloading for C++2000. https://www.stroustrup.com/whitespace98.pdf.

[58] Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. 13–25. doi:10.1145/3622780.3623645

[59] Don Syme. 2020. The early history of F#. *Proc. ACM Program. Lang.* 4, HOPL, Article 75 (June 2020), 58 pages. doi:10.1145/3386325

[60] Daniel Temkin. 2009. Velato. http://velato.net.

[61] Daniel Temkin. 2017. Language without code: intentionally unusable, uncomputable, or conceptual programming languages. *Journal of Science and Technology of the Arts* 9, 3 (Sep. 2017), 83–91. doi:10.7559/citarj.v9i3.432

[62] Daniel Temkin. 2023. The Less Humble Programmer. *DHQ: Digital Humanities Quarterly* 17, 2 (2023).

[63] Daniel Temkin. 2025. *Forty-Four Esolangs: The Art of Esoteric Code*. MIT Press. (to appear).

[64] Michael Thuné and Anna Eckerdal and. 2009. Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education* 34, 4 (2009), 339–347. doi:10.1080/03043790902989374

[65] Deepika Tiwari, Tim Toady, Martin Monperrus, and Benoit Baudry. 2024. With Great Humor Comes Great Developer Engagement. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*. 1–11. doi:10.1145/3639475.3640099

[66] Ethel Tshukudu and Quintin Cutts. 2020. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM conference on international computing education research*. 227–237. doi:10.1145/3372782.3406270

[67] Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. 2025. LLMs Love Python: A Study of LLMs' Bias for Programming Languages and Libraries. arXiv:2503.17181 [cs.SE] https://arxiv.org/abs/2503.17181

[68] Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R. Lyu. 2024. Exploring Multi-Lingual Bias of Large Code Models in Code Generation. arXiv:2404.19368 [cs.SE] https://arxiv.org/abs/2404.19368

[69] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35. doi:10.1145/1118178.1118215

[70] Donald R. Woods and James M. Lyon. 1973. The INTERCAL Programming Language Reference Manual. https://3e8.org/pub/intercal.pdf.

[71] Yicong Yuan, Mingyang Su, and Xiu Li. 2024. What Makes People Say Thanks to AI. In *Artificial Intelligence in HCI*, Helmut Degen and Stavroula Ntoa (Eds.). Springer Nature Switzerland, Cham, 131–149. doi:10.1007/978-3-031-60606-9_9

Received 2025-04-24; accepted 2025-08-11