# Cultural Biases in Programming Languages

## Cultural Bias

Culture meaning implicit ideals, narratives, practices

## Cultural Context

First of all, the drastic gender imbalance in software practictioners is well-known. For instance, the 2022 Stack Overflow Developer survey listed 91.8% of respondents as self-reporting as male (out of 70,000 answers). This was so awkward that they got rid of the gender question altogether for their 2023 survey.

As for the people who designed the EME, it’s overwhelmingly northern

## Cultural Practices

On a more qualitative level, we can look at the excerpt from the source code of the video game Dead Island, developed by the Polish studio Techland. The fact that a qualifier such as `FeministWhorePurna` made it all the way to the release says a lot about the culture of the development team.

## Encodings and interpretations

On a broader level, we can also point to discussions about the Imperialism of ASCII, in *Standards and their stories: how quantifying, classifying, and formalizing practices shape everyday life*. While some might argue that ASCII is just the lowest common denominator, it remains that it excludes, at different degrees, all the individuals who do not use the English script.

Some of the syntax of software systems have been deemed biased, and some of it has been updated. We moved from ASCII to UTF-8, and Unicode is pursuing its mission of inclusivity (along with releasing emojis) and, on GitHub updated their default naming of a `master` branch to `main` branch, following the Black Lives Matters movement. This move in particular was met with some controversy. Is Git using the term `master`, seen as implicitly connected to `slave` really racist? Or is it the interpretation by a particular group of people that is to blame? Some of the people in that debate argue that technology is orthogonal to culture.

## Technics orthogonal to culture

This is a reproduction of a bridge designed by Robert Moses, on the parkway road from New York City to Long Island. Robert Moses was the leading urban planner of the city of New York, and also famously racist. Some commentators, including his biographers, have argued that the bridges were designed so low in order to prevent buses to use these roads, in a historical context in which African-Americans would tend to use public transportation, while White Americans would use individual cars, effectively infusing a racist policy into an artefact.

When technical artefacts operate on cultural preferences, they also become political. This question whether artefacts have politics has been famously addressed by Langdon Winner, illustrated partly with these bridges built by Moses. Starting from the point that "Technology is a way of building order in our world", his argument has two main parts:

- a technology is political insofar as it becomes a way of settling political matters (meaning what gets to be done, what gets to be said). Here, the bridge example is a way of settling who should be allowed in particular places.

- a technology is political insofar as it depends on a certain political system or configuration in order to exist. *"appear to require or to be strongly compatible with particular kinds of political relationships."* The example here is that of the nuclear reactor. A nuclear reactor is technical artefact that is complex enough that it requires a stable, centralized political environment such as a nation-state. It seems unlikely for an anarchist community operating on short-term deliberative consensus to build such a thing.

## Epistemological Pluralism

So a political artefact is one to resolves the question of 'what is the right way to do things'? And in programming, there is a right way to do things.

One cultural bias would be what Papert and Turkle call the "hard" style of programming: top-down, organized, rational. Most programming languages support this kind of thinking.

## Questions

So what does that entail for programming languages? Does their design settle specific issues that might have been resolved differently through a differently-designed artefact? Does their existence rely on specific political configurations that enable them to thrive? Are there intrinsic configurations of the programming languages we currently have, which favor a particular way of doing things?

## Plan

So the way that I propose to address these questions is to look first at what are programming languages, how they might embody values, and what are the agreed-upon desired values of programming languages?

Then, I’ll take a comparative approach by presenting a few cases of *esoteric programming languages*, called *esolangs*. Offering radically different (dissenting, one might say) design approaches, I will show how they help cast a new light on some assumptions of mainstream programming languages. This way, we might be able to recast desirable properties of these languages as biases.

## Programming languages

So what is a programming languages?

So a programming language is a kind of a technical artefact (Turner, XXXX), defined by having two kinds of main properties: functional properties, and structural properties.

## Functional

Functional properties of a programming language concerns what the programming languages actually *does*, how it behaves. What a programming language does, on a high-level, is that it allows for the creation of computational artefacts, such as data structures, algorithms, or (correct) programs. They are represented through the semantics of the language, and their effectiveness.

## Structural

Structural properties, are the ones that make up the formal arrangement of a programming language, the elements that actually constitute it in a material way. Here, I mean material in the sense that they render perceptible the ideas and concepts that are involved in, and intended by, an artefact. This concerns the syntax of the programming language, what kinds of words can be written, according to which rules.

And there is an agreement about “the right way to design a programming language”: most programming language designers and programmers agree that a good language should be correct, clear, and expressive.

Indeed, there’s a question here: even though the Turing-completeness would make each programming language virtually equivalent to every other, there are thousands that exist. All of them are different at least in syntactic choices, while the overall structure remains the same. Are these differences superficial, or do they reveal deeper socio-cultural biases?

## Preferences

And yet, all Pls are not similar, they all make design decisions that allow certain things to be done easily

There are specific design decisions to facilitate specific things. Go facilitates asynchronicity. Processing facilitates drawing. Rust facilitates memory manipulation.

To the extent that syntax allows for the different representations, for the different considerations of things, it can also influence (without dictating) the importance given to some entities of the problem domain over other entities (in some cases, the problem domain is the real world).

Different paradigms (declarative, logic, imperative, functional, object-oriented, etc.) are one way of reasoning, describing, and acting upon the world.

## Data types and first-class citizens

Similarly, primitives derive from ontological precedence in PL components. Such precedence can then be interpreted as a hierarchical relationship: some things are more fundamental than others in programming languages. continuing the metaphor of political rights, we can point to fundamental laws preceding (or being the building blocks of) subsequent elements.

For instance, one data type could be Kotlin Money.

## Code Poetry

Code poetry is a counter style of programming.

Perl is an imperative scripting language, which matches a grammatical mood of natural languages. Such mood emphasizes actions to be take rather than, for instance, descriptions of situations, and sets a clear tone for the poem.

The fact that Perl is based on stating procedures to be executed and states to be changed creates this feeling of relentless urgency when reading through the poem, a constant need to be taking actions, for things to be changed, to be done.

Here, the native constraints of the programming language interacts directly with the poetic suggestion of the work in a first way: the nature of Perl is that of giving orders, resulting in a poem which addresses someone to execute something. It shows the language in a new, ominous light.

## Esolangs

One way to highlight biases is to provide alternatives (this is a role that is easily borne by art in general, and by fiction in specific). We subvert expectations and twist things to make see the background assumptions of our ways of working, being.

As such, they’re mostly primarily cultural artefacts, rather than first and foremost technical artefacts. They were reacting to and building on the aesthetics of commercial coding and the often unstated values of computer science. These disciplines, which are sometimes at odds with each other, are both driven by a pragmatism that esolangs actively eschew. In rejecting practicality, esolangs carve out their own aesthetic and make clear the contradictory factors at work in mainstream code aesthetics.

For instance, the first programming language was INTERCAL. The idea here was to create a technical artefact as a practical joke (taking practical in the very literal term), but also to highlight the culture of the programming environment, making it transpire through this creation (and the accompanying manual!). This is from section 2.2:

> INTERCAL’s main advantage over other programming languages is its strict simplicity. It has few capabilities, and thus there are few restrictions to be kept in mind. 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.

Here, this makes fun about the value of simplicity: indeed, programming languages are actually not simple, but programmers might like to pretend they are.

## TrumpScript

TrumpScript was released by Sam Shadwell in 2016, in the run up of the United States presidential election, with the tagline `Make Python Great Again`. It is a dialect of Python, but with some twists.

The reason why this is a funny satire, it is because it highlights what are deemed to be shortcomings in seeing the world through Donald Trump’s lens. Implicitly, it sets up the absurdity of holding such irrational views. It has interesting points about ambiguity and abstraction.

Fact and lie: Some of these views include the denial of truth and falsehoods for facts and lies: but you don’t want to lie if you’re writing a program, you don’t want to deceive the machine, because the implicit value, as we have seen, is clarity and consistency.

Globalization: The last one is about the assumption of interoperanbility, globalization and abstraction. Ever since C and the advent of modern operating system, the ability to disconnect from the hardware has become a backbone of programming language design, hardware which has been replaced by computer architectures. It doesn’t matter that it’s made by Lenovo, or in Taipei, as long as it is an x86 processor on which the code will run (Java’s “Write once, run everywhere”).

## Alb

Correlating this bias towards globalization of Alb, by Ramsey Nasser. His tools broke, but the extended ligatures are interesting

## C+=

C+= is a programming language that exists as a commentary on the futility of importing cultural biases within technical artefacts. It was originally designed as a satire by The Feminist Software Foundation, and the proposal of a language is clearly a (tasteless) stab at what the authors might consider as “social justice warriors”.

The underlying question is: why shouldn’t there be a feminist programming language? In fact, the original inspiration for C+= is a long and rich exchange on feminist programming. For instance, someone proposes that “a feminist programming language is one that respects the agency of objects, acting upon them through mutual consent”. doesn’t seem so far fetched. maybe languages do not need to be turing-complete, maybe just becase you can do something, doesn’t mean you should do it. consent might be stupid, but computers already have approaches to *consensus* and *niceness*.

One implicit bias we might see here, and one that is highlighted by *esolangs* in general, is that of the culture of performance, highlighted by Byung Chul-Han as one of the traits of our contemporary societies. This is something that Zach Blas develops on in his speculative work on a *QueerOS*:

> “QueerOS embraces uncertainty. It welcomes crashes.”, src

There could be other aspects, such as Aspect-Oriented Programming, or Relationship-Oriented Programming, as mentioned briefly in David Golumbia’s The Cultural Logic of Computation

## Cree# 1

The last example is CreeSharp, and it is perhaps the most serious contender for the idea of proposing an alternative to the paradigms of programming languages. CreeSharp is written by Jon Corbett, and aims to be a design of a programming language based on specific cultural references, metaphors and principles of indigenous peoples. When he was encoding the practice of beaded portraiture, Corbett found himself confronted with a key problem. He was implementing a cultural practice in a programming language that had been based on another culture's knowledge paradigms.

It does propose interesting things:

e.g. **smudging**

> For example, in my acimow/CREE# language, I introduced smudging, the act of burning herbs to purify and protect bodies and spirits, as a computational function. Programmatically, this digital smudging clears the physical and cached memories, initializes peripherals, and clears the screen. Essentially, it prepares the system to execute a program in an environment free of latent data that could negatively impact it src Further more, he proposes metaphors of rivers and arms for the conditional statement, which does tie back in to the flow of control flow.

Further more, he proposes metaphors of rivers and arms for the conditional statement, which does tie back in to the flow of control flow.

## Cree# 2

So, according to Corbett, considering the cultural biases of a programming language is also about reconsidering non-computational concepts, such as the value of time, efficiency, verbosity. Additionally, you also need, just like Nasser and Alb, to reconsider the peripherals to the programming language. For instance, he also proposes a keyboard to organize character writing differently.

Cultural biases towards keyboard writing is well-documented in Thomas Mullaney's *The Chinese Typewriter*, where the challenging assumption that a restricted keyboard could not be made up of all the characters necessary for the output has led to the invention of T9 input (that is, decoupling the direct relationship between input and output).

He also mentions this requirement of producing a useful artefact, something he describes as being "what the machine wants". During the design process, he reports a few moments of whether or not to compromise this attempt so as not to "go against the grain". So even though we're talking about cultural biases, there does seem to be some essence to computation that might be somewhat disjointed from cultural concerns.

But, functionally speaking, it's still non-existent!

## Conclusion esolangs

So the arbitrariness of cultural approaches when going against “the grain” of computation (trumpscript, c+= which deride cultural approaches by implying that they are non-functional), can reveal some interesting questions.

Should all computers assume interconnectedness at all times? Is that the consequence of an imagination coming out the cultural bias of globalization and extension being good? Why should things always be available everywhere at all times?

This also suggests a bias towards decontextualization (which can perhaps be called abstraction): programming languages tend towards abstracting away differences (in hardware, encodings), in order to propose a single plane at which it exists or matters (ignoring any other networks of dependency, such as input peripherals, display peripherals, documentation, etc.).

The imperativeness implies that giving orders is the only way of getting things done (which is not always the case of declarative languages), and it demands on the clear distinction and repeatability of entities. Such property is valued because it is useful, but it might also be worth considering non-usefulness as cultural value (from play to wu wei). In such a context, the fragility and uncertainty of technical artefacts would not be so looked down upon.

## Conclusion technical cultural bias

I have been talking about the cultural biases of programming languages, but I would like to end on a corollary.

André Leroi-Gourhan makes a distinction between technical tendencies, and technical facts: all cultures will invent some sort of way of inscribing characters as a tendency, but there will be design differences (papyrus, turtle shells, etc.) as technical facts. We have been talking about the technical facts of culturally marked programming language designs, but how about the overall tendency?

Jacques Ellul proposes that there is an essence of technique (his word for technology), with properties on its own, driven by a never ending yearning for efficiency and autotelism. Yuk Hui re-uses this concept by showing how a culture as radically different from Western culture as China hasn’t managed to make technology their own, but rather was affected by the technological imperatives. He says there is less of a Chinese cultural bias towards technology, but rather a technological bias to chinese culture.

So maybe we should look at how the core desire for efficiency in programming languages are creating a bias in cultural activities that originally did not need to be efficient?