A PERSONAL COMPUTER FOR CHILDREN OF ALL CULTURES

Ramsey Nasser, digital artist

At the time of this writing, the front page of laptop.org is a carousel of beautiful images: children the world over happily toting the colorful OLPC XO laptop. Its moving photos tell the story of the rugged, affordable green-and-white computer produced by the *One Laptop Per Child* project, and the impact it is having on the developing world. Featured are a classroom in Afghanistan, a stairway outside of a home in Nepal, a street in Palestine, and many more such scenes, full of smiling children excitedly using their machines.

Despite the name, OLPC's stated mission is not explicitly to distribute laptops to every child on earth, but rather to "empower the world's poorest children through education." The laptops are a means to this higher, humanitarian end rather than an end in and of themselves. This is by no means new or unique. The vision of the computer as an educational dynamo dates back at least to Seymour Papert's work on Logo at MIT in the late 60s and Alan Kay's work that followed closely after that. By 1972, Kay would produce his seminal paper *A Personal Computer for Children of All Ages*, which described the then- and still-hypothetical *Dynabook* computer. While mostly famous for inspiring the form factors of modern laptops and tablets, the paper is also significant in that it explicitly frames the Dynabook and computer programming as tools for "learning by making". The first speculative story Kay includes involves youngsters Beth and Jimmy learning about gravitational physics by playing and reprogramming a video game. Unsurprisingly, Kay and Papert were both involved in the launch of the OLPC project in 2005, bookending their careers as pioneers of educational computing. As the world becomes more digital, and computer literacy becomes important for participation in society and the workforce, the image of the computer as a universal engine of education and empowerment—almost as old as computing itself—has only become more pronounced.

More than most computer projects, the OLPC XO takes “localization” seriously. In pursuit of the project’s global goals the interface of their operating system and its documentation are available in a variety of world languages. Digging past the surface, however, one finds that the provided programming experience is built on the popular *Python* programming language, notably made up of English language words and for which no localization is provided. Furthermore, the XO provides a terminal program to access the underlying UNIX system and, in the words of the project, “[allow] kids to dig deeper into their systems, issue commands, and make modifications to their laptops.” But a UNIX terminal interacts with utilities from the POSIX standard, all of which have English language names, and for which no localization is provided. The dependency of parts of the XO on a particular written language seems to fly in the face of the goals of the project, so why is it there at all?

The fictional Dynabook is presented as an abstract, neutral platform for its users to engage with and repurpose. But the production of OLPC XO reveals the crucial difference between an idealized computer and a real one: real computers are technical artifacts produced by thousands of engineers over hundreds of person-years of labor. They are the accumulation of countless software and hardware components made by people who never directly coordinated with one another in most cases, and whose work is likely being used in manners increasingly divergent from any of their original intentions. Every one of these components necessarily makes assumptions about itself, how it will be used, and the world in which it will exist.

Adopting UNIX and Python were certainly pragmatic and effective design decisions for the XO. But those technologies and others turn out to have deeply rooted assumptions around the English language that the OLPC project cannot meaningfully alter, and the result is that if any of the smiling children from Afghanistan, Nepal, or Palestine were curious enough to pull back the layers of their laptops, they would invariably encounter a language foreign to them. Could OLPC have made different decisions? If the humanitarian project poised to empower the world’s poorest children produces machines that carry a bias for a particular written culture, what other fields of computing do the same? Does it matter? How empowering can a computing experience in another language be? And is an entirely non-English computing experience even possible?

VISIBLE BIAS

Another corner of computing exhibiting a surprising and highly visible linguistic bias is digital typography. Many production-grade typesetting systems use a simple text-layout algorithm that works something like this: Given a font and text to display, for each character of the text:

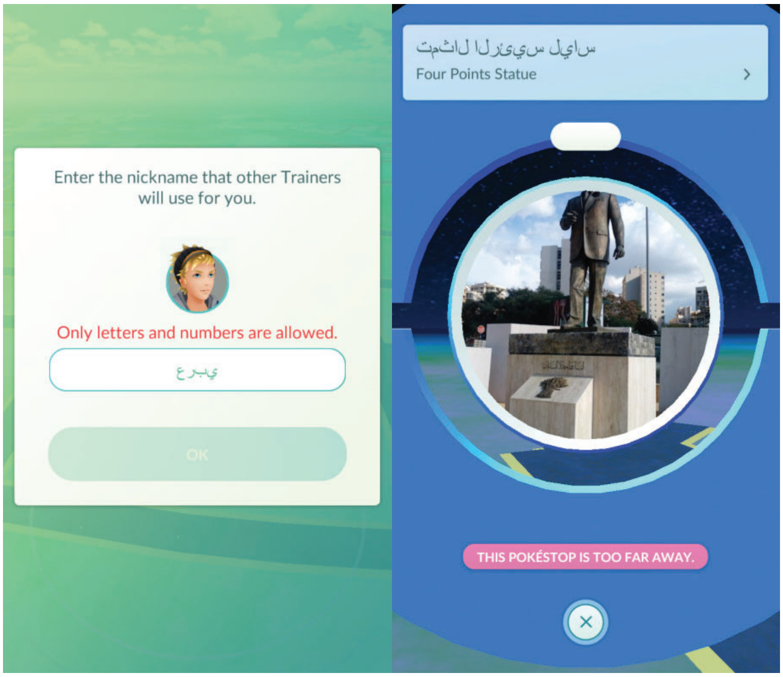
1. Look up the character's glyph in the font
2. Display the glyph on the screen
3. Move the cursor to the right by the width of the glyph

This approach is used in high profile projects including ImGui[1] and Three.js[2], and it effectively bakes in the assumptions that every character has exactly one corresponding glyph, and that text flows in a single direction. These assumptions are simple to implement and suit the Latin script that these systems were designed for, but fail for other languages. The Arabic script in particular presents something of a worst-case scenario: unlike Latin, Arabic flows from right-to-left and in certain circumstances glyphs are positioned vertically, and unlike Latin, Arabic is always cursive, meaning the same letter will have a different glyph depending on its surrounding letters. Other scripts like Devanagari and Thai are similarly poorly served and cannot be rendered correctly without considerable additional work.

This contributes to the consistent public butchering of non-Latin text by digital typesetting software. I maintain a blog at nopenotarabic.tumblr.com to keep track of examples of these issues as they affect the Arabic script. Examples of highly visible rendering failures include the Athens Airport, Pokemon Go, a Lil Uzi Vert video, Coke ads, Pepsi ads, Google ads, Captain America: Civil War, and anti-Trump art, among others (see Figures 1 and 2).

The Arabic in each example exhibits the exact same error: the text is correctly spelled but rendered backwards (i.e. from left-to-right), and none of the letters are joined. The resulting text does not approach legibility and may as well be chicken scratch to an Arabic reader. What's most likely happening in every one of these cases is that well-meaning but non-Arabic-speaking graphic designers are pasting Arabic language text into their graphics packages which is not equipped for it, and, seeing something vaguely Arabic-looking with no error message, considering their job done and moving on.

The experience of seeing text like this in public is a deeply hurtful reminder to every Arabic reader that the digital world was not built for them, and that their culture is an afterthought at best. Seeing text like this in an otherwise high budget movie or video game is its own kind of cultural violence, making a mockery of a script that is native and even holy to many. It is a communication by non-Arabs for non-Arabs, and it reveals a willingness to use Arabic as a cultural prop, but none to do the work to get it right.



Enter the nickname that other Trainers will use for you.

Only letters and numbers are allowed.

يبرع

سايل سيئرلا لاثمت

Four Points Statue

THIS POKÉSTOP IS TOO FAR AWAY.

Figure 1

These failures happen in two places. First off, they're evidence of a lack of diversity in production and graphic design firms. Had there been Arabs or Persians or Pakistanis present in the design or decision-making process or even consulted as subject matter experts these mistakes would have been caught immediately and most likely addressed. I can speak from personal experience and say I know that my presence as an Arab on or adjacent to software projects allowed me to point out incorrect rendering and prevent butchered script from going public on more than one occasion.

Second, the tools that are commonly available and claim to handle "text" have failed because instead of actually handling all text, they were really only designed and tested for the Latin alphabet. Instead of issuing a warning when faced with non-Latin text they did not support, they displayed a butchered version of the script that was similar enough to the real thing to fool the designers in charge. Adobe products before the Creative Cloud era and major game engines like Unity 3D and Unreal are all guilty of this. Technical solutions exist, but engineers have been slow to update large legacy codebases, and the problem persists.

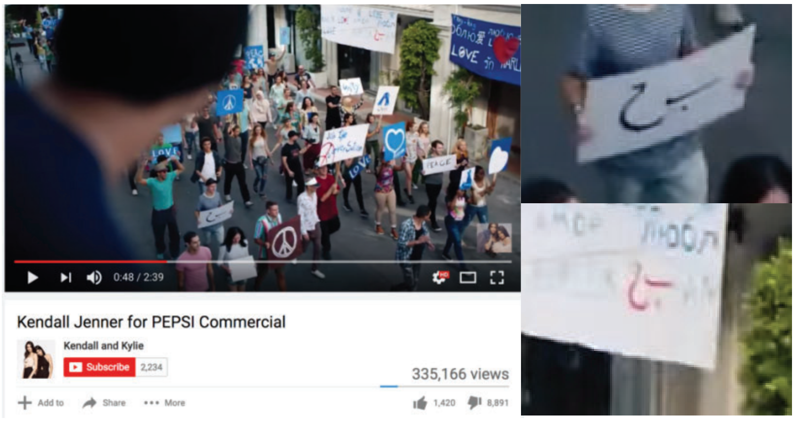


Figure 2

INVISIBLE BIAS

Although errors in text rendering get splashed across the billboards and screens of the world, bias towards the English language is deepest in the less visible world of computer programming languages. And while text rendering is slowly getting better, the situation is much more bleak in the world of programming, where it is much harder to find a toehold for hope.

Every programming language in serious use today is based on words and punctuation taken from the English language and writing conventions. In order to use these tools, some knowledge of English is a requirement. In order to use these tools most effectively, actual proficiency in English is unavoidable. This favors programmers natively familiar with English over others and makes a truly inclusive and culturally neutral programming experience impossible.

As concrete examples, consider these three recursive implementations of the Fibonacci sequence in Python, Swift and Ruby taken from the Rosettacode project:

Python

```
def fibRec(n):
  if n < 2:
      return n
  else:
      return fibRec(n-1) + fibRec(n-2)
```

Swift

```
func fibonacci(n: Int) -> Int {
  if n < 2 {
      return n
  } else {
      return fibonacci(n-1) + fibonacci(n-2)
  }
}
```

Ruby

```
def fib(n, sequence=[1])
 n.times do
   current_number, last_number = sequence.last(2)
   sequence << current_number + (last_number or 0)
 end
 sequence.last
end
```

These languages are useful as specimens of the dominant family of programming languages as of the time of this writing, but the following critique will apply to every language in contemporary use. Full and abbreviated English language words abound, and they can broadly be separated into two categories: keywords, and identifiers.

Most languages provide some set of basic built-in functionality that is not subject to creation or modification by the user. Syntax for this functionality is provided via keywords, parts of the language that are “baked in” and receive special treatment by the interpreter or compiler. Examples of these in the above examples are **def** in Ruby and Python (short for “define”) and **func** in Swift (short for “function”) introduce new functions. **if** and **else** in Python and Swift denote conditional branches to control the flow of program execution. **return** in Python and Swift cause a function to terminate and produce a result. Finally, while Python uses indentation to denote the start and end of its blocks of code and Swift uses curly braces, Ruby uses the English word **end** to mark the end of a block started by a **def** or a **do** keyword.

Identifiers are names that the programmer assigns to functions, values, and data structures. It is up to the programmer to define them, though the language will impose restrictions on what constitutes a legal identifier. In these examples, each function is given a name **fibRec** (short for “fibonacci recursive”), **fibonacci**, and **fib**, by the programmer. In theory they could be anything, though the chosen names convey the intent of the functions well. A bit trickier are the identifiers **last**, **times**, and **println**. These are names of functions that are provided by the standard library of the language, a set of useful utilities included with the language itself that allows programmers to be productive right away without having to reinvent standard operations. They are technically not treated differently from a programmer’s own functions, and in theory could be replaced, though in practice this can be a challenge and is generally not done.

This distinction is important because it separates the English language content of modern code into two categories: one which is an intrinsic part of the programming language itself and one which is at least in theory subject to change from the outside. At first glance, the former category seems to be the more entrenched one, requiring entirely new languages to be designed to introduce new keywords. That was my first intuition, but with time I came to find that the latter category, the one of user-defined names, is the insurmountable challenge that makes large scale non-English programming impossible.

قلب: لغة برمجة

Despite growing up in Lebanon speaking Arabic and studying Computer Science in Beirut, none of this was immediately obvious to me. I had the privilege of being proficient in English for most of my life, so I never gave much thought to the fact that every programming language I had ever seen shared a common linguistic heritage. When I moved to New York for graduate school my research took me towards designing better languages for new programmers. Initially my focus was on moving away from the old syntaxes and semantics of the 70s and towards something more modern. But with time I realized that every design for a new language I conceived of used English language words pervasively. The

English-centric assumptions ran deep enough to color not only my present, but also my imagination for different futures. I started to wonder who were these "new programmers" for whom I was designing, and what was the effect of the natural language their programming experience based on.

To probe this question and confront my own biases I became interested in designing a non-English programming language for non-English speaking new programmers. I picked Arabic as it is my native language and what I know best, and I knew that I had to make a new language in order to get away from the English keywords baked into all existing languages.

The language I built was called قلب, pronounced qalb or ‘alb in the Lebanese dialect that I speak. The word means "heart" and is a recursive acronym for قلب: لغة برمجة, pronounced ‘alb: lughat barmajeh, meaning "heart: a programming language." This is what the Fibonacci function from above looks like.

```
(حدد فيبوناتشي (لامدا (ن)
  (إذا   (أصغر؟ ن ٢)
    ن
  (جمع (فيبوناتشي (طرح ن ١) )
    (فيبوناتشي (طرح ن ٢) ) ) ) ) ) )
```

With support from the Eyebeam Art and Technology Center, I developed the قلب interpreter towards the end of 2012. It was crucially important that قلب not be a speculative project, but a real functioning programming language. I wanted to go as far as I could following all the rules of classic language design and implementation to understand why deviating from English never happens in practice.

قلب is, by design, a boring programming language. It is a text-book Scheme interpreter based on Peter Norvig's *Lispy*—the kind a first year Computer Science student might turn in for an assignment. Its sole deviation from Norvig's interpreter is in using a non-English language with a non-Latin script as its basis, treating the rest of the language as the control in the experiment.

In some ways قلب succeeds in providing a non-English programming experience. First off, it provides Arabic keywords in place of the more ubiquitous English ones. if becomes إذا, def becomes حدد, and even mathematical operators like + and - are written out as Arabic words جمع and طرح. Furthermore, the parser will reject user supplied identifiers that contain anything but Arabic letters, effectively requiring every visible word in the language to be in Arabic. And with the Arabic numerals used in Latin languages replaced with the Indic digits more common in Arabic, قلب's rejection of English is complete.

This small change is enough to cause serious problems, however. The project's source code is hosted at github.com/nasser/--- rather than the more correct github.com/nasser/قلب because GitHub requires ASCII-only project names, and collapses non-ASCII characters into hyphens. Text editors are easily confused by Arabic script, often failing to correctly remap the arrow keys and displaying source code in a manner reminiscent of the nopenotarabic blog. Similarly, terminal emulators used to interact with command line tools can be stumped by the presence of Arabic text. These failings and more make the act of programming in قلب tedious and error-prone compared to programming in the better supported English based languages. By choosing a language other than English as its basis, قلب reveals hidden linguistic biases in programming tools that otherwise "just work." It is not a language a programmer could ever be comfortable or productive in as a result.

These frustrations alone are not enough to doom the whole project, however. With time and effort, better text editors and terminal emulators can be written, and the assumptions that web hosting platforms make can be revised. What makes قلب or any project like it impossible to succeed at scale rather than merely difficult is something much deeper, and ultimately, much less technical.

THE OBJECTIVE BECOMES SUBJECTIVE

Computers are fundamentally number processing machines. As a piece of physical hardware a CPU is really only capable of basic operations on numeric data, like arithmetic, loading numbers from memory, and writing numbers to memory. When people point out that computers are "just ones and zeroes" this is likely what they are referring to: the binary representation of numbers that a computer manipulates. This is the closest computers come to being truly neutral devices.

The problem is that people generally want their computers to be more than giant calculators. Early computers were just that: engines to compute bomb trajectories and crack enemy encryption during war. But as the decades passed and they became more widely available and integrated into people's lives their tasks grew while their fundamental architecture remained largely unchanged. Modern computers seem to handle a lot more than numbers, from text to images to audio and video, but all of that data is still represented numerically at the level of the machine. And while numbers do have a measure of objectivity to them, the manner in which non-numerical data is represented numerically is completely subjective.

Representing Writing

Take text, for example. Text can be represented digitally as a sequence of numbers by assigning a number to each character in a natural language. The crucial question is: which numbers should represent which characters? Pure mathematics has nothing to say here because the mapping is arbitrary. It does not matter as long as it's consistent. If you decide that 0 should be 'A' and 1 should be 'B' and so on, you've successfully represented text as numbers. In an isolated context, there's little more to it than that. But as software is confronted with other software it is forced to communicate, and if everyone invents their own digital representation of text, their programs will not be able to share textual data. If a program designed using the above mentioned representation received text from a program that decided that 0 should in fact represent 'Z' and 1 should represent 'Y' and so on, it would misinterpret the data. In this case, neither program would have done anything technically wrong, but the failure rather would have come from the human programmers' lack of agreement.

This level of coordination is crucially important for the interoperation of computer systems. Historically, national standards bodies would define text encodings for a nation's computer systems to use. With the advent of the internet, even this approach became insufficient, as software had to deal with textual data from other nations. The Unicode Consortium was formed in 1991 to define a single encoding for all human writing systems: the *Unicode Encoding*. At its heart, the Unicode Encoding is an enormous table mapping numbers to characters in different scripts that almost every computing system in use has agreed upon. This allows text to be processed by computers as numbers, but the subjective meaning of those numbers comes this international agreement negotiated far and away from inside a CPU.

It is through assigning meaning to numbers that human bias and history to creeps into software. And Unicode itself is far from perfect. For example, for historical reasons Latin characters are assigned the lowest numbers and as a result can take up less space in memory than characters in other languages. To preserve space in the encoding, Han characters "common" to Chinese, Japanese, and Korean are assigned the same number, making distinguishing national variants difficult in some cases, and hurting Unicode's popularity in East Asia. There is consistent debate around new writing systems, and the decision around what language gets added to the standard is a deeply human, deeply political one.

The Second Hard Problem In Computer Science

Computer programming is an exercise in managing enormous amounts of complexity. The stream of millions if not billions of binary numbers needed to execute a non-trivial program at the level of a CPU immediately dwarfs the human mind. In order to make sense of anything within a finite lifespan, programmers use programming languages to act as a layer between themselves and the machine that will run their code. Their programming languages present a suite of tools more palatable to human thinking to express programs that will eventually be turned into streams of numbers a computer expects. One of the most prevalent and powerful of these tools is the ability to name things.

Assigning a name to a procedure or data structure makes it easier to think about and reuse. Compare reasoning about “the function at address 9036724” to “the print function”, or “the value at offset 24” to “the ‘name’ field of the ‘Person’ record.” Programming languages perform this transformation when compiling a programmer’s code into machine code, and in general the particulars of this process can safely be regarded as an implementation detail and paid no mind. Programmers as a result are given this convenient abstraction to work within, where code and data can have meaningful names, and the computer can continue to process numbers as it always has.

Names also facilitate a crucial kind of collaboration in software. Programmers often share useful code they have written as a *libraries*, also known as *frameworks* or *software development kits* in some contexts. By learning the names of the procedures and data structures in a library other programmers can build on the work of the original authors and avoid redundant effort. Libraries also provide mechanisms for a programmer’s code to talk to an operating system or hardware, like Apple’s *iOS Software Development Kit*, or IEEE’s POSIX standard.

It is important to stress that modern programming is only possible because of this collaboration. A programmer today is only able to write an application “from scratch” in less time than ever before because they are building on decades of existing code written by programmers they’ve never met, published as libraries. Without sharing code, every programming endeavor would have to start from the level of the hardware every time, and progress that transcends an individual project would be impossible.

Another important detail is that procedures and data structures in libraries must be accessed by way of the names they were assigned by their original author. These names fall in the category of identifiers discussed earlier. They are subject to definition by the original programmer of a library, and in theory could be anything the language considers a legal identifier. But a user of a library must include the exact names chosen by the original author in their own code and cannot exchange them for anything else. Put another way, the names used in libraries are not merely decorative or explanatory, they are an essential part of the library itself. Even more, the wider the use a library finds the more incentive there is to never change any of its names, as that would require rewriting any code that used the old names, and the amount of labor involved could be intractable. For example, consider the function **malloc** from the POSIX standard. **malloc** is a standard way for a program to request memory from an operating system. The name is cryptic to new programmers, but it is short for "memory allocate" and contractions like that were popular in the programming of decades past. But changing **malloc** to, say, the arguably more readable **memoryAllocate** is impossible as it would require the billions of lines of code deployed around the world already referring to **malloc** to be updated.

This inertia extends beyond libraries and into protocols, another realm of software coordination. Protocols are agreements between programmers on the formatting of data so that software systems can communicate. Many of them have no linguistic properties, only specifying the order of bytes numerically. But some so-called "readable" protocols do encode data as language, and are subject to the same problems as libraries. An example is the RFC 2616 specifying the *Hypertext Transfer Protocol*, HTTP or more commonly "the Web." HTTP uses named "headers" as part of the communication between a server and a browser. Each header has a name and a value. Examples of header names include **Location**, **Retry-After**, **Last-Modified**, and **If-Modified-Since**. A system participating in HTTP as a server or browser must use these exact headers with their punctuation, spelling and capitalization or it will be ignored for generating invalid data. Like libraries, these names are a hard requirement of the protocol, and no substitution is possible without creating a new protocol. As one of the most widely distributed protocols on earth, these names are practically eternal, to the point that even spelling mistakes in the original specification cannot be changed anymore. One of the HTTP headers is **Referer**, missing an r, but fixing that typo in every server and browser on the planet would likely be prohibitively expensive at this point.

Indeed names are so important that Kay's description for the programming language of the Dynabook revolves almost exclusively around them

> *The use of this language is essentially divided into two activities: 1. giving names to objects and classes (memory association), and 2. retrieving objects and classes by supplying the name under which they had been previously stored. A process consists of these (activities) and is terminated when there are no longer any names under scrutiny.*

Although all of such a language can be easily derived from just these two notions, a few names would have an a priori meaning in order to allow interesting things to be done right away.

And Computer Scientist Phil Karlton's famous joke uses them as a punchline

There are only two hard things in Computer Science: cache invalidation and naming things.

Naming things *is* terribly important and difficult, but not for the reasons most Computer Science texts get into. Absent from most conversations about names in programming is how deeply cultural the act of naming something is. Historically, naming a territory was part of the spoils of war, attested to by dozens of cities named "Alexandria" from Egypt to Afghanistan left behind by Alexander the Great's global conquest. Names attributed to a thing also encode the perspectives of the namer. What Westerners refer to China is known to its indigenous population as *Zhōngguó*, meaning "Central Kingdom." The name China likely comes from the Sanskrit or Persian names for the long-passed *Qin* dynasty. My own personal name, Ramsey, was deliberately chosen by my parents to be pronounceable in the West and in my native Lebanon (رمزي, *Remzi*, in Arabic) as they imagined my future before I was even born. Naming is a deeply human act that records history and language and can be poetic, beautiful, violent, and just about anything but neutral.

And therein lies the problem. Names are what allow human minds to comprehend and manipulate the vast complexity of computing, and modern programming is only made possible by building on existing systems. This necessitates using protocols and libraries built by others and invoking names of their choosing in any new code. As there is no such thing as a culturally neutral name, programmers today are forced into familiarity with the written culture of programmers past. The examples earlier from POSIX and HTTP had their roots in the English language that their authors were fluent in, and this is true of every library and protocol in contemporary use. Programming is always a social and collaborative act. Even when one is working alone, a programmer is always indirectly collaborating with the thousands of programmers that came before them and adapting the systems they left behind to new uses. The progress made at American organizations like Bell Labs and Xerox PARC from the 60s onward gave us the foundations of modern software, but also enshrined the culture of those engineers into every programming system that followed.

The fact that using English language names is unavoidable when interacting with libraries and protocols is what makes قلب, and any project like it, ultimately doomed to failure. Non-English programming projects are confronted with an impossible choice: cling to your conceptual and political purity and be cut off from the world of software, or abandon purity, allow English identifiers, and defeat your own purpose for existing. قلب itself is only able to implement basic games and browser interactions by maintaining a bridge between itself and JavaScript code internally, though this is effectively a cheat, and there is no way to expose this mechanism to the programmer. Purely non-English languages could never talk to the web, or email, or any other protocol based on English language. They could not build on the sixty years of software libraries written using English names, and would have to reinvent it all from scratch themselves, siloing them off from the rest of the world and from history, which is both unrealistic and undesirable.

The reality is that programming will most likely remain dominated by English indefinitely, and familiarity with the language is a prerequisite for entrance into the software engineering industry. A true Computer For Children of All Cultures, a computing experience where a learner could pull back layer after layer of software and never encounter anything but their own written culture is not meaningfully possible unless their written culture happens to be American English. When I think about قلب and the possibility of a young Arab learning how to program in their own language, only to inevitably outgrow the limitations of the system and eventually realize that to become a "real" programmer they'd have to learn English after all, it breaks my heart. That isn't a moment I want to craft for anyone, and why I consider قلب an impossible project. The door to a non-English programming experience is closed now, if it was ever open, and as time passes and more software gets written, its closes tighter still.

THE DIFFICULT PROBLEMS ARE NEVER THE TECHNICAL ONES

Accepting that the problem is not technical or computer-specific, but cultural and linguistic is a step towards imagining different futures. The central question is not "how do we build non-English programming experiences?" but rather the much trickier "how do we facilitate communication across linguistic boundaries?" A definitive answer is difficult, but there are a few non-solutions that can be discarded right away.

First off, projects like قلب that attempt to do what English-based programming does but in another language are not the way forward. As demonstrated above, the deep reliance on named things in programming reveals a flaw in the premise of such projects. But even if they could be overcome, investing an enormous amount of effort to add a single language to the pantheon of programming is beside the point. At best, it gets us towards A Personal Computer For Children of Some Cultures—whichever cultures can afford to invest in reinventing the history of software engineering—which is a much less compelling goal. Recreating existing power structures with a different group on top is not an act of liberation.

Picking a “common” auxiliary language on which to build programming languages on is also likely a dead end, primarily because such a language does not exist. *Esperanto* gets framed as such, but its script and grammatical structure are decidedly European and hardly global. A true auxiliary language for the people of the world that would be appropriate to consider as the foundation for a future of programming would borrow much more from Chinese, Hindi, and Arabic. The absence of such a language and the difficulty in designing and popularizing one makes this a poor way forward as well.

Finally, the more pragmatic-minded might suggest a system that involves automatically translating identifiers from the English-based ecosystem to the languages of the world. It’s important to reject this on both technical and political grounds. Technically, machine translation is poor to the point of being unusable in most cases, and programming is full of made up words. For example, what is the Pashto translation of **AbstractSingletonProxyFactoryBean**[3]? The technical shortcomings reveal the political problems: a translation based approach makes non-English languages second class citizens of the programming world. “Real” programming would continue to be done in English, while translations were generated for everyone else, modulo quality of translation. Again, this isn’t true equity, and not a terribly exciting goal to work towards.

None of these approaches meaningfully begin to build the bridge across linguistic gap between human beings that would be required for a truly equitable programming experience. Ideally, the languages of the world could pool together and build on each other. Code written in Arabic could use code written in French, which could build on code written in Japanese. With no specific natural language receiving special treatment, all languages could be treated equally, and a new common programming experience could emerge from it all.

This is a fantasy, but there have been moments in the human history where common languages emerged out of necessity. From the 11th to the 19th century sailors and traders around the Mediterranean spoke a language called *sabir* or *lingua franca*, an organic blend of Italian, Spanish, Portuguese, Berber, Arabic, Turkish, and Greek. This language was not designed but rather emerged naturally from the interactions of merchants from different cultures trying to do their jobs. Though far from equitable or utopian, it was a situation where one side could not easily assert complete linguistic dominance over any other, resulting in an emergent new means of communicating.

What would a *lingua franca* for programming look like? How does one design a programming language to emerge naturally from its users as opposed to being passed down unchanged from the past? It is hard to say. Such a language would face all the challenges mentioned here, and be incompatible with most current internet protocols and software ecosystems. But if it could promise a truly equitable programming experience upon which to build a real Personal Computer For Children Of All Cultures, it *just might* be worth hitting the reset button.

Ramsey Nasser is a computer scientist, game designer, and educator based in Brooklyn. He researches programming languages by building tools to make computing more expressive and makes work that questions the basic assumptions we make about code itself. His games playfully push people out of their comfort zones, and are often built using experimental tools of his design. Ramsey is a former Eyebeam fellow and a professor at schools around New York.

A PERSONAL COMPUTER NOTES

1. Available at: https://github.com/ocornut/imgui/blob/fe5347ef94d7dc648c237323cc9e257aff6ab917/imgui_draw.cpp#L2666

2. Available at: https://github.com/mrdoob/three.js/blob/f81506e172571ab106d0164530bbc1a4802fc2d4/src/extras/core/Font.js#L63

3. Available at: https://docs.spring.io/spring/docs/2.5.x/javadoc-api/org/springframework/aop/framework/AbstractSingletonProxyFactoryBean.html

A PERSONAL COMPUTER FIGURES

Figure 1: Pokemon Go screenshot (Nassar, 2016), accessed from: https://nopenotarabic.tumblr.com/post/149021392583/a-wild-pok%C3%A9mon-go-appeared-it-used-common-arabic

Figure 2: Pepsi Ad screenshots (Nassar, 2017), accessed from: https://nopenotarabic.tumblr.com/post/159231568203/nothing-says-love-like-butchering-a-language