Why does English-based, 'typewritten' code remain dominant, and what problems does it present?

Aidan Hall

12th May 2022

I declare that the work presented in this essay is my own and that it has not been submitted for assessment on any other module.

Abstract

All popular, modern programming languages are based on plain, ASCII text with English identifiers. The style is archaic, making code harder to read, especially for non-native English speakers. It developed within the technical limitations of typewriters, but has persisted due to language designers only making improvements incrementally, and dependence on existing tools and libraries. In this article, I propose ways to improve programming languages, including Unicode operators and identifiers based on non-English natural languages. I also discuss the virtues of programming paradigms other than imperative; I consider functional and symbolic (for their reduced use of English identifiers), as well as literate programming, which shifts the focus of programming away from computer execution towards communication between humans. I then discuss the potential practice of using multiple notations in unison, in a form of 'heterogeneous programming'. None of the ideas I discuss can see quick adoption, but can be gradually implemented over time as new languages and developments arise, with the ultimate goals of making programming more accessible and expressive.

1 Introduction

Today, the dominating standard form for computer programming languages is ASCII text (Stack Overflow 2021). I will refer to it as typewritten code (Arawjo 2020, p. 6). This ubiquitous notation restricts innovation in the field of programming language design (Iverson 1980, § 5.4). It also presents additional barriers to non-native English speakers due to the use of English-based identifiers (e.g. for variables and functions) (Guo 2018, p. 2).

In his retrospective analysis of early 'visions' of programming notation, Arawjo describes how FORTRAN, one of the earliest foundational programming languages, developed. He says that it wasn't 'inevitable'; instead, it was the product of "incremental improvements to [the programmers'] interactions with computers", within the constraints of the typewriter keyboards the designers used (Arawjo 2020, p. 8).

The virtually universal adoption of typewritten code ensures that it will remain dominant indefinitely. Legacy code maintenance keeps the same languages in use years after they would have otherwise lost relevance (Fleishman 2018). Additionally, even when developing new programs, programmers are almost certain to rely on software libraries created in a typewritten language; although it is possible to make procedure calls between languages, doing so requires some commonality of form to construct argument lists and make said calls.

Beyond the code itself, many of the common software tools used in aid of software development work almost exclusively with typewritten text; code editors and version control systems (Git) are among the most notable of these.

On top of all this, there is the community of programmers who have worked with typewritten code for years or even decades, gaining irreplacable familiarity with the associated coding environment.

Iverson warned of an 'unfortunate circularity of design' (Iverson 1980, § 5.4) of programming languages and hardware, where early preoccupations with efficient performance informed the design of early programming languages, which then went on to inform the design of later computers, and so later programming languages.

In 2022, Python and JavaScript are the most popular programming languages (Stack Overflow 2021). These indicate that the persistence of typewritten code is no longer based on a continued obsession with performance, since they prioritise ease of use over efficiency. Instead, it is due to the legacy of older languages and programming environments. As a result, I contend that we are ready for a shift towards even more easily-useable notations.

In this essay, I will consider ways for programming methodologies and notations to improve, both to make programming more accessible to non-native English speakers, and to improve their effectiveness as tools of thought and communication.

2 Unicode Operators

Despite the fact that Unicode now supports all of the major mathematical symbols that programming notation previously needed to approximate, ($\leq$, $\wedge$, $\rightarrow$), all major programming languages remain predominantly stuck with ASCII identifiers and operators (`<=`, `&&`, `->`) (Stack Overflow 2021). For older languages, this is unavoidable, due to the need for backwards-compatibility, but even newer languages like Rust retain these conventions.

Code ligatures visually replace some of the more obscure ASCII notations with their Unicode counterparts. One of their primary purposes is to make ASCII-based operators more readable (Fira Code), so there are evidently programmers who desire Unicode operators. Unfortunately, hiding the ASCII operators only increases confusion for the uninitiated.

Modern programming languages such as Python and Haskell allow the creation of Unicode identifiers. The Julia language (Ekre et al. 2022) supports both styles, and the APL language (Iverson 1980) is predominantly composed of non-ASCII symbols. There is evidently no technical or theoretical barrier to implementing them into languages. Instead, the problem is the practical matter of inputting the characters.

Since old typewriter keyboards only had keys for a tiny number of maths symbols, the designers of early languages had to create the ASCII imitations for the excluded ones (Arawjo 2020, p. 6-7). Newer keyboards then only needed to be capable of inputting the same ASCII characters to support the existing languages, perpetuating the prior limitations. This is a case of the 'unfortunate circularity of design' referred to by Iverson (see introduction).

The first and most obvious solution is a keyboard which has more symbols on, such as the APL keyboard [image]. This is not a viable approach with respect to maximising adoption, since not all programmers would be willing or able to replace their hardware.

A more plausible alternative would be support within the text editor to input these symbols. For example, the TEX input methods in Emacs (Emacs28, § 22.3) and the Julia REPL (Julia, § Unicode Input) allows the user to input TEX macros, which are then replaced with the corresponding Unicode symbol (e.g. `\le` $\rightarrow \leq$). Unfortunately, any such 'macro' system

would inevitably add complexity, meaning the additional symbols would still be harder (and crucially slower) to input than ASCII.

3 Multi-Lingual Identifiers

Guo (2018, p. 2) discusses how English-based identifiers for logic, functions and variables can present serious obstacles to non-native English speakers. This applies not just to reading other people's code, but also to creating semantically appropriate names themselves (ibid., p. 6) (Chistyakov 2017).

One potential solution is simply creating programming languages with identifiers based on natural languages other than English. Whilst programming with one's native language is naturally easier for learners (Guo 2018, p. 2), I don't see this as a good solution to the problem of communication between programmers. The issue is that code is inextricably linked to a single language in the first place; English is the standard predominantly for historical reasons.

A more promising approach would be 'multi-lingual' programming languages, which could support identifiers in multiple natural languages. Guo (ibid.) refers to these as 'bilingual labels'.

The first method was allowing others to create variants of the same language (Wijngaarden et al. 1973, § 0.1) with non-English identifiers. In this case, there will effectively just be multiple incompatible languages, with the same result as before. This technique, therefore, would be of limited value in aiding communication between programmers with different native tongues.

A more advanced method Guo mentions is block-label localisation in Scratch (Guo 2018, p. 2). The principle is to have an underlying representation for what a given object is, then display the appropriate localised label within the editor. While this does liberate the code somewhat from any given natural language, the method has other problems:

- Every new construct would need localised identifiers in all supported languages, making the process of adding new ones slower.
- As Guo discusses later on (ibid., p. 6), there may not be direct 1:1 translations for certain labels.
- In the case of Scratch, user-created identifiers (for lists, variables and custom blocks) are still stuck in one language, ignoring the localisation system entirely.

- This runs counter to the concept of 'plain' text, wherein a displayed character or word directly corresponds to the underlying representation. Consequently, it would be challenging to implemented into conventional typewritten languages.

The flaws of these techniques show that multi-lingual programming is not a problem we have a good solution for yet, especially within the domain of typewritten text.

4 Alternative Typewritten Paradigms

Acknowledging the dominance of the typewritten, we should consider ways to make this family of programming languages more accessible to non-native English speakers. Following the principle of 'universal design' referred to by Guo (2018, p. 10), doing so could be beneficial for the programming community at large.

4.1 Functional

Using the functional programming paradigm directly reduces the number of identifiers the programmer needs to create, in comparison to imperative code (Chistyakov 2017). In functional languages, programmers build up complex behaviour through function composition, with the need for local 'state' variables being lower than in imperative languages. The programmer then often only needs to create identifiers for the function and its parameters, with point-free programming techniques reducing the need even for parameter names.

Point-free programming involves making use of features in certain programming languages to define a function (usually a composition: `f(g(x))`) without explicitly referring to its parameters. For example, this Haskell function converts a String to an array of upper-case words:

```
import Data.Char (toUpper)
uppercaseWords :: String -> [String]
uppercaseWords = words . map toUpper
```

4.2 Symbolic

Taking the approach of function composition to the extreme brings us to APL (Iverson 1980), which is so terse that the implementation for complex function compositions such as matrix product is just a few characters (`+.×`).

At that point, a meaningful identifier would be longer than the implementation itself, so APL programmers form collections of 'idioms': short code snippets for use in one's own programs (LearnAPL: TL;DR).

Of course, this only makes sense up to a certain point of complexity, and Iverson himself refers to the power of 'Subordination of Detail' (brevity, including use of named abstractions) in writing code (Iverson: § 1.3).

Furthermore, APL only levels the playing field of accessibility insofar as it makes code just as hard to read for (uninitiated) native English speakers as it is for others, by stripping away the associations with natural language, since it is symbolic. Iverson describes APL as a (universal, executable) mathematical notation (Iverson), and these tend to be more language-agnostic.

Given only 0.65% of respondents to the Stack Overflow developer survey (Stack Overflow 2021) reported using APL, it evidently hasn't had a significant long-term impact on the programming landscape.

4.3 Literate

An approach which could make code easier to understand for everyone, not just non-native English speakers, is adoption of a literate programming style (Knuth 1984). With literate programming, the main focus is clear communication (to another person) of the purpose and design of a program, as opposed to creating a working product by any means. The concrete implementation in Knuth's paper involves defining short blocks of code with natural-language identifiers, interspersed with text that documents not just what the code does, but also how and why. The programmer can then reference these code blocks using the natural-language identifiers similarly to procedure calls. The technique is similar to the combined use of mathematical notation and natural language in academic maths[1].

The natural-language code block identifiers are of interest in relation to English accessibility. Guo's survey revealed a desire for 'in-line dictionaries', or the option to translate identifiers within the code editor (Guo 2018, p. 8). While attempting to do this with 'normal' code identifiers could prove challenging, computers can translate natural language moderately well already. Having (sufficiently accurate) translations provided automatically mitigates most of the issues with multi-lingual identifiers (see above). Natural-language identifiers are also inherently easier to understand, in comparison to cryptic abbreviations (ibid., p. 2).

[1] I consider Iverson's paper on APL (Iverson 1980) to be a close approximation of literate programming, and Knuth lists APL as an alternative language for his WEB system.

A compelling advantage of this style of documentation over simple comments is that the natural-language description is the identifier, and is consequently at less risk of losing its relevance in later versions of the code. Of course, variables and functions within the program code itself would still have intrinsic textual names (almost certainly in English), but a literate style could also encourage the programmer to explain their use of a less-than-obvious name (or to correct unjustifiable ones).

One could point out that potentially verbose in-line documentation of a program could make it harder for non-natives to read. However, since most implementations of literate programming use some form of markup for the written explanation, this could easily include diagrams and images that illustrate features of the program more clearly than text. This is powerful, since 23% of the learners in Guo's survey specifically expressed a desire for 'More Visuals and Multimedia' in educational materials (Guo 2018, p. 7), p. 7), owing in part to their language-independence.

One notable inclusion would be block or flow diagrams, which early programmers considered an essential part of the programming process (Arawjo 2020, p. 5). This ties into the idea that literate programming is an effort (or at least an opportunity) to document the development process of a program, not just its final behaviour.

Given the focus on communication that literate programming entails, I consider it well-suited (if not best-suited) to creating code examples. Since the style allows the programmer to break code up, beyond even sensible procedural abstraction, they can explain and illustrate each part to whatever extent is necessary right alongside the code itself. Each block of code and its accompanying explanation can be short, making them easier to comprehend (especially for non-native English speakers!). The focus on the development process is valuable here as well, since part of the purpose of examples is to teach others how to create similar code for themselves.

Furthermore, since such examples would need to be 'tangled' into a full working program, the author would have to make sure the example was correct, complete and up-to-date. For example, the classic practice in educational code examples of omitting error handling and/or cleanup for brevity (Kelly 2019) wouldn't be necessary, since these could be in separate literate blocks (Knuth 1984, p. 12).

Prominent contemporary implementations of literate programming environments include Jupyter and Emacs Org-Babel.

5 Heterogeneous Programming

The next logical step after literate programming is heterogeneous programming (Arawjo 2020, p. 9). The principle is to use multiple different notations for programming, choosing whichever is most applicable for a given situation. Arawjo gives the example of embedded quantum circuit diagrams within a program which 'are the code', coexisting with surrounding typewritten procedure calls.

Scientific research papers could take the form of executable programs (Lasser 2020), with the mathematical equations themselves forming part of the code. Lasser uses Python for its readability, but an executable mathematical notation such as the one in Geogebra would be even more so, even for more complex equations:

```
def quadratic(a, b, c):
    det = b*b - 4*a*c
        detRootDiv = math.sqrt(det)/(2*a)
        bDiv = -b/(2*a)
        return [bDiv + detRootDiv, bDiv - detRootDiv]
```

$$quadratic(a, b, c) = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

For inter-operation with equations that include non-ASCII symbols, such as λ in radioactive decay, using Unicode identifiers instead of something like `lambda` would make the meaning clearer.

6 Conclusion

Typewritten text is deeply entrenched into the practices and culture of programming. In industrial applications, where large teams develop code iteratively over the course of years or decades, it is likely to persist indefinitely. Given all the products this medium has produced, one cannot deny its effectiveness. Instead, a focus on inclusion and accessibility in the field of Computer Science (of which programming is but a small part) should inform our judgements about how we design and implement programs and programming languages.

By shifting the focus of programming from instruction and execution to communication, we can bring an international community of computational thinkers closer together, and create better, more understandable software. In treating programming as a medium of thought, we can consider techniques

to make it more expressive, which can grow to gain wider adoption and significance over time.

References

Arawjo, Ian (2020). 'To Write Code: The Cultural Fabrication of Programming Notation and Practice'. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, pp. 1–15. ISBN: 9781450367080. URL: https://doi.org/10.1145/3313831.3376731.

Chistyakov, Artem (28th June 2017). *The language of programming*. URL: https://temochka.com/blog/posts/2017/06/28/the-language-of-programming.html (visited on 04/05/2022).

Ekre, Fredrik, Kristoffer Carlsson, Milan Bouchet-Valat, Michael Hatherly, Alex Arslan, Valentin Churavy, Tim Holy, Sacha Verweij, Morten Piibeleht, Mohit Nain, Kiaran B Dave, Jeff Bezanson, Rafael Fourquet, Chris Foster, Amit Murthy and Chris Rackauckas (27th Apr. 2022). *Julia 1.7 Documentation*. Version 1.7.2. URL: https://docs.julialang.org/en/v1/ (visited on 04/05/2022).

Fleishman, G. (Apr. 2018). *It's COBOL all the way down*. Stripe. URL: https://increment.com/programming-languages/cobol-all-the-way-down/ (visited on 03/05/2022).

Guo, Philip J. (2018). 'Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities'. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, pp. 1–14. ISBN: 9781450356206. DOI: 10.1145/3173574.3173970. URL: https://doi.org/10.1145/3173574.3173970.

Iverson, Kenneth E. (Aug. 1980). 'Notation as a Tool of Thought'. In: *Commun. ACM* 23.8, pp. 444–465. ISSN: 0001-0782. DOI: 10.1145/358896.358899. URL: https://doi.org/10.1145/358896.358899.

Kelly, A. (9th Jan. 2019). *Error handling omitted for brevity*. URL: https://www.allankelly.net/archives/2890/error-handling-omitted-for-brevity/ (visited on 29/04/2022).

Knuth, D. E. (Jan. 1984). 'Literate Programming'. In: *The Computer Journal* 27.2, pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. eprint: https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf. URL: https://doi.org/10.1093/comjnl/27.2.97.

Lasser, J. (19th Aug. 2020). 'Creating an executable paper is a journey through Open Science'. In: *Communications Physics* 3.1, p. 143. ISSN: 2399-3650. DOI: 10.1038/s42005-020-00403-4. URL: https://doi.org/10.1038/s42005-020-00403-4.

Stack Overflow (15th June 2021). *2021 Developer Survey*. URL: https://insights.stackoverflow.com/survey/2021 (visited on 03/05/2022).

Wijngaarden, A. van, B.J. Mailloux, C.H.A. Koster J.E.L. Peck, L.G.T. Meertens M. Sintzoff C.H. Lindsey and R.G. Fisker (Sept. 1973). *Algol 68. Revised Report on the Algorithmic Language*. report. ALGOL 68 Revision Committee. URL: https://web.archive.org/web/20150906170502/http://jmvdveer.home.xs4all.nl/algol68/report.html (visited on 02/05/2022).