JavaScript Affogato: Programming a Culture of Improvised Expertise

Brian Lennon

Configurations, Volume 26, Number 1, Winter 2018, pp. 47-72 (Article)

Published by Johns Hopkins University Press
DOI: *https://doi.org/10.1353/con.2018.0002*

JavaScript Affogato: Programming a Culture of Improvised Expertise

Brian Lennon
Pennsylvania State University

ABSTRACT: This essay attempts a philological—meaning a both technically and socially attentive—historical study of an individual computer programming language, JavaScript. From its introduction, JavaScript's reception by software developers, and its importance in web development as we now understand it, was structured by a continuous negotiation of expertise. I use the term "improvised expertise" to describe both conditions for and effects of the unanticipated development of JavaScript, originally designed for casual and inexpert coders, into a complex of technical artifacts and practices whose range and complexity of use has today propelled it into domains previously dominated by other, often older and more prestigious languages. "Improvised expertise" also marks the conditions for and effects of three specific developmental dynamics in JavaScript's recent history: first, the rapidly accelerated development of the language itself, in the versions of its standard specification; second, the recent, abruptly emerging, yet rapid growth of JavaScript in server-side networking, data processing, and other so-called back-end development tasks previously off limits to it; third, the equally recent and abrupt, yet decisive emergence of JavaScript as the dominant language of a new generation of dynamic web application frameworks and the developer tool chains or tooling suites that support them.

Introduction

2016 was an inconspicuously transitional year for the information space once commonly referred to as the World Wide Web. Those

Configurations, 2018, 26:47–72 © 2018 by Johns Hopkins University Press and the Society for Literature, Science, and the Arts.

attentive to linguistic usage will recall that the 2016 edition of *The Associated Press Stylebook and Briefing on Media Law* released in June recommended that the words "internet" and "web" no longer be written with initial capital letters,[1] in a sign that the propriety marked by their referents' novelty had finally settled, or worn off. For those more attuned to matters of technical infrastructure, what may come to mind instead is the announcement by Oracle Corporation that its Java web browser plugin would be deprecated in the forthcoming ninth version of its Java Development Kit (JDK), a platform for writing and packaging software applications in the Java programming language.[2] Taking these two real, if lesser milestones together, it seems safe to say that for anyone who remembers the original promise made for Java applets as a common WWW technology, at their moment of emergence in the mid-1990s, this was a chapter of recent technological and cultural history quietly coming to an end.[3]

To be sure, Oracle's hand had been forced by Microsoft, Google, and Apple, who had either reduced Java plugin support in their browser products or removed it entirely. And yet embedded Java applets had long since become a legacy technology, still useful for some computationally intensive graphical visualization tasks (disproportionately in scientific applications), but no longer in wide use outside that domain. Whether or not they are old enough to remember the role originally imagined for Java in the browser, in particular, most of those who design websites and program web applications for a living today would be unlikely to regret their eclipse. Even before the emergence of personal data security as a substantive public issue in 2012, Java applets presented grave, often intractable security risks that web developers had had to learn how to manage, or ignore. A more general reason for the irrelevance of Java browser applets by 2016 was a historical one, linked to changes in the profile of the Java programming language and Java programmers in the software development industry as a whole. When Fredrick P. Brooks Jr. chose for the seventh chapter of *The Mythical Man-Month: Essays on Software Engineering* (1975) the title "Why Did the Tower of Babel Fail?," he

1. Davey Alba, "The AP Finally Realizes It's 2016, Will Let Us Stop Capitalizing 'Internet,'" *Wired*, April 2, 2016, http://www.wired.com/2016/04/ap-finally-realizes-2016-will-let-us-stop-capitalizing-internet/.

2. Tom Warren, "Oracle's Finally Killing Its Terrible Java Browser Plugin," *Verge*, January 28, 2016, http://www.theverge.com/2016/1/28/10858250/oracle-java-plugin-deprecation-jdk-9.

3. Some readers may also have thought of the acquisition of Yahoo Inc. by Verizon Communications, announced on July 25, 2016, several months after an initial draft of this essay was completed.

was reflecting on the biblical story of Babel as a fable of engineering (the hubristic or merely presumptuous construction of a tower tall enough to reach heaven), rather than a fable of language (divine punishment imposed in the form of linguistic difference and permanently impaired communication).[4] Nevertheless, *The Mythical Man-Month*, the first widely read and still the most celebrated reflection on managing large software projects, was also an informal study of communication, not excluding the metaphorized communication that a software programmer struggles to achieve with a machine.

The present essay is about the negotiation of technical expertise, specifically the technical expertise involved in software programming, and in particular that involved in programming websites and applications—that is, what is today called "web development." I take my bearings from the present historical moment, understood as an interval of continued economic recession (or, if one insists, "uneven recovery") shaped by both economic and political investment in "coding" instruction as job retraining for unemployed and underemployed US blue- and white-collar workers alike. In recent years, sociologically oriented cultural studies scholars like Adrian Mackenzie have produced valuable work on cultures of software development, work that is laudable for being simultaneously technically informed and socially focused.[5] While it has been receptive to such research in so-called software studies, scholarship in humanities disciplines has not displayed a proportionate interest in the specifically social and cultural dimensions of the specifically linguistic history of computing, and this is the case especially where individual programming languages and their development and usage cultures are concerned.[6] The broad exception is, of course, the historiography of

4. See Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary ed. (Reading, MA: Addison-Wesley, 1995).

5. See Adrian Mackenzie, *Cutting Code: Software and Sociality* (New York: Peter Lang, 2006).

6. A partial exception is Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter, *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* (Cambridge, MA: MIT Press, 2013), seven of the eleven main chapters of which focus on the BASIC programming language. Only one of these seven chapters could reasonably be called a study of the BASIC language, however, with the remaining six devoted to explaining very simple command sequences and very brief programs to readers who are assumed to have no knowledge either of BASIC or any other programming language. The book's eighth main chapter, entitled simply "BASIC," does discuss language design and syntax variation in some detail, but is otherwise given over to reviewing BASIC's implementation and usage history, again for a reader assumed to lack elementary knowledge of the subject. As of this writing, *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* is the only book-

computing, and science and technology studies more broadly. But even here, Mark Priestley is surely right to suggest that early, purely technical histories of programming languages have been followed by socially attentive histories of software as a general object and domain, leaving individual languages behind as objects of potentially equally both technically and socially focused study.[7]

Granting that no duplication of the early, narrow technical histories is necessary—they were meticulous, if unsurprisingly disproportionately anecdotal in character[8]—how can we describe the humanities research space separating an early historiography of programming languages that is as old as the Fortran, Lisp, Algol, and Cobol languages themselves (which originates, that is to say, in the late 1950s), and recent social histories of the software concept as Martin Campbell-Kelly's *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*?[9] A clue is to be found, I would suggest, in an essay by William Paulson entitled "For a Cosmopolitical Philology: Lessons from Science Studies,"[10] insofar as in that essay, Paulson suggested the value of bringing science and technology studies (STS) scholarship into contact with an older literary humanist tradition of philology: a tradition whose methodologies were globally comparative and multilingual, whose mode was the study of texts in multiple languages (which required intensive study of the languages themselves), and which was rooted in a specific Western intellectual-historical tradition, the tradition of secular or historical humanism. If we set aside this latter tradition (one that STS scholars would surely understand themselves as sharing with "philologists," that is, with

length publication to have emerged from "critical code studies," an undertaking that Mark C. Marino attempted to distinguish from "software studies" more than a decade ago. See Mark C. Marino, "Critical Code Studies," *Electronic Book Review*, December 4, 2006, http://www.electronicbookreview.com/thread/electropoetics/codology.

7. Mark Priestley, *A Science of Operations: Machines, Logic and the Invention of Programming* (New York: Springer, 2010), p. 2.

8. See Jean E. Sammet, *Programming Languages: History and Fundamentals* (Englewood Cliffs, NJ: Prentice-Hall, 1969); Donald E. Knuth and Luis Trabb Pardo, "The Early Development of Programming Languages," in *Selected Papers on Computer Languages* (Stanford, CA: CSLI Publications, 2003), pp. 1–94; Richard L. Wexelblat, ed., *History of Programming Languages* (New York: Academic Press, 1981); Thomas J. Bergin and Richard G. Gibson, eds., *History of Programming Languages II* (New York: ACM Press / Addison-Wesley, 1996); Thomas J. Bergin, "A History of the History of Programming Languages," *Communications of the ACM* 50: 5 (May 2007): 69–74.

9. See Martin Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry* (Cambridge, MA: MIT Press, 2004).

10. William Paulson, "For a Cosmopolitical Philology: Lessons from Science Studies," *SubStance* 30:3 (January 2001): 101–119.

language and literature scholars), philology's characteristic mode of focus, grounded as it is in the mandates of linguistic specificity, even incommensurability, cannot be described as a great strength or even necessarily a normal characteristic of scholarship in STS.

From a position close to Paulson's own, then, one might invite software studies and so-called critical code studies, as well as STS itself, to establish an as-yet imagined contact with philology. What, we might ask, would a philological study—that is, a minimally both technically and socially oriented historiography—of a specific computer programming language look like? For an example of how this question might be posed within the disciplinary context of the information sciences, rather than within that of the humanities (as I shall do here), we can consult recent work like Andrew J. Ko's "What Is a Programming Language, Really?" "In computing," Ko remarks, "we usually take a technical view of programming languages (PL), defining them as formal means of specifying a computer behavior. This view shapes much of the research that we do on PL, determining the questions we ask about them, the improvements we make to them, and how we teach people to use them. But to many people, PL are not purely technical things, but *socio*-technical things."[11] Still, essays like Ko's are quite remarkably few and far between, in the domain of the technical sciences as much as in the social sciences and the humanities—and often, as in this particular case, perhaps unavoidably perfunctory. Regardless of how we choose to explain it, Ko's conclusion in 2016 that "[o]ther agendas, particular those that probe the human, social, societal, and ethical dimensions of PL, are hardly explored at all"[12] is certainly warranted.[13]

Java and JavaScript

In December 1995, when Sun Microsystems and Netscape Communications issued a joint press release announcing "JavaScript,

11. Andrew J. Ko, "What Is a Programming Language, Really?," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (New York: ACM Press, 2016), 32–33, at p. 32.

12. Ibid., p. 33.

13. The "politico-social history of Algol" promised by R. W. Bemer, for example, turns out to be a bibliography with abridged extracts from various primary sources (letters, meeting minutes, committee resolutions, and so on), many relating to the famously fractious negotiations of the specification of Algol 60 in particular. It is to that history of conflict to which the term "politico-social" presumably refers; still, this document is entirely descriptive and offers no analysis whatsoever. See R. W. Bemer, "A Politico-Social History of Algol (with a Chronology in the Form of a Log Book)," *Annual Review of Automatic Programming, Annual Review of Automatic Programming* 5 (1969): 151–237.

the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet,"[14] Sun's Java programming language was already well on its way to achieving the virtually uncontested market dominance, comparative prestige, and privilege as an instructional language that it would enjoy for a decade and more. Though Java 1.0, the first public release, had appeared only the same year, Sun's promise of true platform-neutrality and portability for the Java Runtime Environment was immediately attractive to enterprise software developers tiring of the demands placed on them by the C and C++ languages then widely in use. Java promised to moderate some of the complexity entailed by the access both C and C++ provided to low-level memory management, as well as the specific complexities introduced by C++ imagined as "C with classes,"[15] without reducing the power and expressivity those languages offered to enterprise systems programmers specifically. Though it was initially designed for the lightweight hardware application of embedding in programmable consumer appliances, and only later adapted for serving and embedding in HTML pages, Java was very much a professional's language, restrictive in its requirements for data types (being both statically and strongly typed) and in its promotion of a single programming style, the object-oriented programming (OOP) paradigm it would help popularize, as well as in the verbosity that both these forms of restriction produced. Presented as a professional alternative to both C and C++ rather than a radical departure from either, Java's relative ease of use included no special claims of approachability for inexperienced coders or nonprofessionals.

Java*Script* was different. Sun and Netscape's press release used the word "complementary" three times to describe JavaScript's relation to Java: "JavaScript as a complement to Java" (in the document's subtitle). "The JavaScript language complements Java." "[JavaScript is] complementary to and integrated with Java." Java, the press release emphasized, "is used by programmers to create new objects and applets," while JavaScript "is designed for use by HTML page authors and enterprise application developers to dynamically script the be-

14. "Netscape and Sun Announce Javascript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet," December 1995, https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html.

15. Designed by Bjarne Stroustrup while working at AT&T in the late 1970s, C++ was originally called "C with Classes," marking Stroustrup's intention to "superset" the C language (that is, to remain completely compatible with it) while also improving it. See Bjarne Stroustrup, "Bjarne Stroustrup's FAQ," Bjarne Stroustrup's Homepage, October 1, 2017, http://www.stroustrup.com/bs_faq.html.

havior of objects running on either the client or the server." If the mention of "enterprise application developers" and server-side applications suggested a place for JavaScript in the established industry of Java development, the sentence that followed better illuminates how HTML page authors were imagined, and how they imagined JavaScript's complementing of Java in a different sense. "JavaScript is analogous to Visual Basic," it read, "in that it can be used by people with little or no programming experience to quickly construct complex applications."[16]

From its introduction, JavaScript's reception by software developers, and its importance in "web development" as we now understand it (as an area of either software development or graphic design, depending on whom one asks), was structured by a continuous negotiation of expertise. Especially today, it is rare to encounter an introductory tutorial or textbook for beginners that fails to pause to disambiguate JavaScript from Java before undertaking to cover even the basics.[17] Most often, and especially today, the motive for such disambiguation is less to clarify the historical relationship of these two languages than to clear a space for JavaScript by separating it from association with Java—specifically, with Java's verbosity and its object-oriented programming paradigm, and perhaps from Java's association with enterprise application programming, the drudge work of software engineering—and its diminished presence in the more

16 "Netscape and Sun Announce Javascript" (above, n. 14).

17 See, for example, Jeremy McPeak and Paul Wilton, *Beginning Javascript*, 5th ed. (Indianapolis, IN: John Wiley & Sons, 2015), p. 2: "Perhaps this is a good place to dispel a widespread myth: JavaScript is not the script version of the Java language. In fact, although they share the same name, that's virtually all they do share. Particularly good news is that JavaScript is much, much easier to learn and use than Java." Even some classic books on JavaScript written for developers already expert in other languages, or newer books that address the same kind of reader, frame the issue similarly. In *JavaScript: The Definitive Guide*, widely considered an authoritative comprehensive study of JavaScript, David Flanagan begins thus: "The name 'JavaScript' is actually somewhat misleading. Except for a superficial syntactic resemblance, JavaScript is completely different from the Java programming language." See David Flanagan, *Javascript: The Definitive Guide*, 6th ed. (Sebastopol, CA: O'Reilly Media, 2011), p. 1. In the introductory volume of a rigorous and well-received multivolume study of contemporary JavaScript, Kyle Simpson writes, "[T]he name [JavaScript] is merely an accident of politics and marketing. The two languages are vastly different in many important ways. 'JavaScript' is as related to 'Java' as 'Carnival' is to 'Car.'" See Kyle Simpson, *Up & Going*, You Don't Know JS (Sebastopol, CA: O'Reilly Media, 2015), p. vii. Douglas Crockford reminds us of Java and JavaScript's historical concurrency and does not exaggerate their unrelatedness, but has little to say about the issue beyond one sentence: "When Java™ applets failed, JavaScript became the 'Language of the Web' by default." See Douglas Crockford, *Javascript: The Good Parts* (Sebastopol, CA: O'Reilly Media, 2008), p. 1.

flexible and experimental startup culture of the 2000s and 2010s, as well. While such gestures are understandable at a moment when Java's reputation is more or less clearly in decline,[18] they can obscure the historical entwinement of these two languages, with consequences that are regrettable from any but the most purely practical or instrumental perspective.

I use the term "improvised expertise" to describe both conditions for and effects of the unanticipated development of JavaScript from a mere complement to Java, designed for casual and inexpert programmers, into a language whose range and complexity of use has now propelled it ahead of Java in some ways, even (by some measures, in some domains) where Java once dominated. My argument is that such "improvised expertise" separates JavaScript at least partly from other, otherwise similar experiments in making programming accessible to non-experts, from the original BASIC language, developed

18. In response to such claims, Java programmers often point to Java's leading position in the TIOBE Programming Community Index compiled by the software services provider TIOBE Software BV, or similar rankings aggregators like the PYPL Popularity of Programming Language Index—leaving unmentioned such rankings' historical "trend" indexes for Java's position, which are frequently negative. See Robert McMillan, "Is Java Losing Its Mojo?," *Wired*, January 8, 2013, http://www.wired.com/2013/01/java-no-longer-a-favorite; and David Cassel, "Evolve or Die: Java, C++ Confront Newcomers on the TIOBE Index," *The New Stack*, March 14, 2016, https://thenewstack.io/evolve-die-popular-programming-languages-confront-newcomers-tiobe-index, both of whom argue that TIOBE data itself shows Java's position "slipping" and "trending down" (McMillan also quotes Paul Jansen, managing director of TIOBE Software, as stating that "Java is falling down"). In any case, at any point in the history of a programming language past the point of its initial adoption, a language's reputation—as expressive or otherwise pleasant to use, as adaptable to ongoing hardware evolution, as usable in solving newer computational problems—may diverge from its market share or other measures of usage quite radically, if only because once they are in place, large industrial software infrastructures are kept operating for as long as possible. It is the incongruence of Java's reputation with its market share, today, that animates nonmeaningless if possibly glib comparisons of Java to Cobol, such as that made by Bill Snyder in "Java Is Becoming the New Cobol," *InfoWorld*, December 28, 2007, http://www.infoworld.com/article/2650254/application-development/java-is-becoming-the-new-cobol.html. The inclusion in Java version 9 of a REPL (Read-Evaluate-Print-Loop) feature for exploratory programming is a concession to the encroachment on Java's position of both scripting languages and newer functional programming languages, languages in both categories of which have offered REPL-type features—whose purpose and usage are fundamentally incompatible with Java-style object oriented programming—for many decades. Arguably, the rise of Scala, Clojure, and other languages designed to run on the Java Virtual Machine (JVM) and provide access to Java standard libraries, but otherwise breaking either partly or completely with Java's imperative syntax and its enforcement of an object-oriented paradigm, marks the endurance of the JVM as a platform but the eclipse of Java as a (paradigmatic) language.

as an instructional language at Dartmouth College in the 1960s, onward.

The concept of improvised expertise also encapsulates the conditions for and effects of three specific developmental dynamics in JavaScript's recent history. First of these is a rapid acceleration in development of the language itself, now occurring at such a pace that ECMAScript, the specification on which JavaScript is based, shifted in 2015 from using traditional ordinal version numbers for editions to a year-based designation (so that the official name of ECMAScript version 6 is now ECMAScript 2015, with new editions to be released yearly going forward). Second is the recent abrupt emergence and extremely rapid growth of JavaScript in server-side networking, data processing, and other so-called back-end development tasks, a domain traditionally handled separately from the user-facing, design-oriented front-end site development that Sun and Netscape's 1995 press release suggested would be JavaScript's main use case. Third is the equally recent and abrupt, yet decisive emergence of JavaScript as the dominant language of a new generation of dynamic web application frameworks (principally Ember.js, AngularJS, and Facebook's React, but also Meteor, Express, and others) and the developer tooling suites that support them, in a partial displacement of the Ruby language-based Rails framework popularized during the late 2000s.

This rapid, largely unanticipated growth in JavaScript's range of application and its general importance in the software industry has even seen it enter elementary computer science instruction as language of preference, in some cases displacing Python (which itself has selectively displaced Java) in the classroom. Here, the phrase "improvised expertise" marks a paradox: while core JavaScript remains a small, approachable language when abstracted from its main domains of application, website and application development, using JavaScript professionally in those domains today is virtually impossible without very substantial, ongoing study of the language's advanced features and support for multiple programming paradigms, as well as of the new JavaScript-based development frameworks and tooling suites, the frenetic development pace of which virtually ensures that they will be replaced by other, newer frameworks and tools before they emerge from beta status and a commensurate level of documentation. This ensures that the learning curve for new professional JavaScript developers—not to mention the nonprogrammers JavaScript was originally designed to serve—will be very, very steep indeed, and it suggests that sooner or later, JavaScript's improvised expertise will have some part to play in the disappointments

of the latest push for "computer science for all" and other economic management schemes that conflate coding skills with basic literacy (that is, reading and writing in human languages) and with basic so-called computer literacy, as well (that is, using both general and domain-specific prepackaged software applications effectively). Where JavaScript's history as a programming language is in many ways a routine, if interesting case of simplification producing complexity, the logic of "coding for all" and its variants are arguably repetitions of magical thinking about the management of complexity in software production itself, with these two dynamics converging in the historical present.[19] In that sense, what we call "JavaScript" is not just a programming language, and not just a collection of environments and tooling supporting a programming language, including specifications and other documentation, implementations, and primary and secondary program artifacts (from development tools and frameworks to specific interpreters or "engines," compilers and transpilers, and other software components embedded in a browser or server applications). JavaScript can, at least at the moment and for the near term, be understood also as an assembly of broader technical and technical-historical dynamics, labor and management practices and arrangements, and discourses about education, job training, and production that privilege technical expertise, but also seek to generalize it in and for a historical interval.

System and Scripting Languages

The first edition of ECMA-262 (ISO/IEC 16262), Ecma International's specification for ECMAScript, a standard for JavaScript, was published in June 1997. Edited by Guy L. Steele Jr., it described ECMAScript as a scripting language, defining the latter as "a programming language that is used to manipulate, customize, and automate the facilities of an existing system,"[20] rather than being used to cre-

19. The difficulties of larger-scale software production are documented by a management-oriented literature stretching back to the 1970s. The canonical text, mentioned previously, is Brooks Jr., *The Mythical Man-Month* (above, n. 4). See also Robert N. Britcher, *The Limits of Software: People, Projects, and Perspectives* (Reading, MA: Addison-Wesley, 1999); Scott Rosenberg, *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software* (New York: Three Rivers Press, 2008); Gerald M. Weinberg, *The Psychology of Computer Programming*, silver anniversary ed. (New York: Dorset House, 1998); and, for useful counterpoint, Ellen Ullman, *Close to the Machine: Technophilia and Its Discontents* (New York: Picador / Farrar, Straus & Giroux, 2012).

20. *ECMAScript: A General Purpose, Cross-Platform Programming Language. Standard Ecma-262, June 1997* (ECMA, June 1997), p. 1, http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf.

ate a new system. It acknowledged that the "existing system" of ECMAScript's original design was a World Wide Web page browser and a Web-based client-server architecture more generally, but also insisted that the ECMAScript specification had been written with a variety of possible host environments in mind.[21] The first edition of ECMA-262 was equally pointed, and in some ways more specific, in emphasizing that "[a] scripting language is intended for use by both professional and non-professional programmers, and therefore there may be a number of informalities and built into the language."[22] The history of what we now call higher-level programming languages is of course a history of efforts to make programming less arduous for *professional* programmers, as operation codes provided mnemonics for instructions that could otherwise only be expressed in binary, octal, or other numeric forms, followed by what we now call programming languages providing another platform-independent layer atop the hardware-specific operation codes, a layer still more remote from numeric encoding and apparently closer to natural language (then and still today, the English language specifically).

Efforts to make programming accessible to nonprofessionals did not, as one might expect, lag behind the effort to make programming more convenient for professionals; rather, they were coterminous and developed in parallel, not without significant overlap. At Dartmouth College, John Kemeny had devised DARSIMCO (DARtmouth SIMplified COde), "Dartmouth's first crack at a simple computer language,"[23] a year before the appearance in 1957 of FORTRAN, the first widely adopted and lasting example of a higher-level or "third generation" language.[24] "Dartmouth students," Thomas E. Kurtz recalled in 1978, "are interested mainly in subjects

21. Ibid., p. 2.

22. Ibid., p. 1.

23. Thomas E. Kurtz, "BASIC Session," in *History of Programming Languages*, ed. Richard L. Wexelblat (New York: Academic Press, 1981), p. 516.

24. The most widely used term in both professional software development and the discipline of computer science is "higher-level language," a spatial metaphor used to describe abstraction from the "lower" level of hardware operation codes. Academic researchers use the historical metaphor of the generation in a similar, if perhaps also more sensible way: a first generation of purely numerically represented instructions ("machine code") is followed by a second generation of mnemonic abbreviations ("assembly language"), followed in turn by compiled, hardware-independent algebraic syntaxes and keywords and phrases in the English language ("programming language" as we use the term today). The classification includes a fourth and a fifth generation, which is beyond my purview here. See James Martin, *Application Development without Programmers* (Englewood Cliffs, NJ: Prentice-Hall, 1982).

outside the sciences," and most of the future "decision makers of business and government" among them were not science students.[25] The rationale for Dartmouth BASIC, or "Beginner's All-Purpose Symbolic Instruction Code," was to provide such students with experience in writing programs (rather than merely learning about computer use) without having to understand operation codes "or even FORTRAN or ALGOL" (the latter another higher-level language developed in the 1950s), which Kurtz and his colleagues considered "clearly out of the question. The majority would balk at the seemingly pointless detail."[26] But the growth of Dartmouth BASIC into an entire family or class of languages represented its dissemination not only as an instructional language, but also in some lines of development (such as that which produced Microsoft's Visual Basic) as "a production programming language for professionals" as well.[27]

Today, terminological usage more or less clearly distinguishes "scripting" languages from "system programming" languages. System programming languages like C and C++ were designed to abstract away much of the detail of assembly-language programming (that is, programming in operation codes) while still leaving the programmer facilities for manually allocating and de-allocating memory and thus staying "close to the metal," as programmers like to say, while enjoying the benefits of higher-level abstraction where that was preferred (for example, in syntax for iteration, branching and other control structures, function calls, and creating and managing collections of items of data). Managing memory efficiently involves distinguishing clearly among different data types (primarily, between mathematical and textual data types) as one makes use of them, so that no more memory than is needed is allocated for storing an item of data, and compilers for system programming languages typically enforce such discipline in the programmer—for example, by refusing to compile a working executable otherwise. Scripting languages, by contrast, abstract away and automate both data typing and memory allocation and deallocation, for the convenience of the programmer. This is partly because they can take for granted the presence of an underlying system programming language and its libraries, for whose components they serve as a kind of adhesive or connective tissue, and in which they themselves are implemented (that is, the interpreter that provides a scripting language with its execution environment is itself a system-level language program).

25. Kurtz, "BASIC Session" (above, n. 23), p. 518.

26. Ibid.

27. Ibid., p. 547.

Since the 1990s, however, various factors, including the accelerating sophistication of hardware and innovations in programming language design, have eroded some difference in the performance of scripting languages relative to system programming languages, at least in specific environments and for specific applications, and significant gains have arguably been made in some measures of programmer productivity. The economy of expression made possible once memory allocation and data typing are abstracted away can be fairly dramatic. If code in a system programming language like C is three to six times shorter, in countable individual instructions, than its equivalent in assembly language code,[28] the same instructions in a scripting language like Python might be half as long as their equivalent in C, C++, or Java syntax, and depending on the task possibly much shorter than that.

Other Contexts

Although it took nearly two decades, it was JavaScript rather than Java itself that made good on the promise of programmable Web pages and the browser application as a distributed multiplatform environment. As HTML-based Web publication promised to disrupt local monopolies of print publishers, JavaScript promised inexpert programmers access to a scriptable environment, while Java was to do the heavier lifting. In one of many interesting early formulations, the Web was imagined as a "shell" for interactive application development, by analogy with the AI-oriented "expert system" shells developed for use with Lisp and Prolog and marketed for rapid application prototyping in Java and other languages.[29] But the popularity of the Web was also used to justify the teaching of JavaScript to novices and as a "precursor to Java."[30]

28. J. K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *Computer* 31:3 (1998): 23–30, at p. 24.

29. See Alison Lee and Andreas Girgensohn, "Developing Collaborative Applications Using the World Wide Web 'Shell,'" in *CHI EA '97 CHI '97 Extended Abstracts on Human Factors in Computing Systems* (New York: ACM Press, 1997), pp. 144–145, at p. 144.

30. See Robert Ward and Martin Smith, "Javascript as a First Programming Language for Multimedia Students," *ACM SIGCSE Bulletin* 30:3 (September 1998): 249–253, at p. 249: "The World-Wide Web is increasingly influencing the teaching of Computing Science and associated subjects, and Web-related programming topics are now appearing in many syllabuses. Whilst in this respect there has been much development and discussion of Java as a first programming language with many text books now available, JavaScript has been comparatively ignored. . . . We propose here that JavaScript is sufficiently rich in concepts to support the teaching of introductory programming, and that it is especially suitable for Multimedia students." See also Rebecca Mercuri, Nira Herrmann, and Jeffrey Popyack, "Using HTML and JavaScript in Introductory Program-

The fading of Java's promise as a browser language did not immediately elevate JavaScript. One writer of the late 1990s correctly anticipated the development of browser-independent implementations of JavaScript (fully realized in 2009 with Node.js, discussed below), but incorrectly expected JavaScript to be displaced by Perl as a browser scripting language.[31] Today, after twenty years of emphasis on JavaScript's role in client-side web development (that is, on the software browser's presentation of data to the user), it is seldom remembered that Netscape Communications had explored server-side applications for JavaScript from the start. This is clear from the language of the 1995 joint press release with Sun, which specified that "JavaScript is an easy-to-use object scripting language designed for creating live online applications that link together objects and resources *on both clients and servers*," and that it was "designed for use by HTML page authors and enterprise application developers to dynamically script the behavior of objects running on *either the client or the server*."[32]

Still, it is not difficult to identify in retrospect some conditions that arguably later served JavaScript's explosive growth, including developments virtually coterminous with its first appearance. On April 30, 1995, the US National Science Foundation's NSFNET, a publicly funded network of supercomputer centers and telecommunications

ming Courses," *ACM SIGCSE Bulletin* 30:1 (March 1998): 176–180, at p. 176: "Here we report on a course designed to exploit students' burgeoning interest in the World Wide Web (WWW), where we used HTML and JavaScript to teach programming concepts. These languages allow students at different skill levels to work side by side, learning common abstract ideas while implementing them at different levels of complexity, motivated by the rewarding and exciting interactive environment of the WWW."

31. See Aaron Weiss, "JavaScripting into the Next Millenniun," *netWorker* 3:4 (December 1999): 34–35, at p. 35: "As a programming language alone, JavaScript's main appeal has been its simple learning curve, but to more experienced programmers it lacks serious muscle-power. There are sharks in these waters—established, mature programming languages such as Perl can now be embedded into some Web browsers. . . . For a seasoned developer, the prospect of combining client-side Perl—with its agile handling of advanced programming models—with access to the DOM would be lethal to JavaScript. We will likely see the migration of other scripting language into the Web client as well, including Python, TCL, SmallTalk, and perhaps more."

32. "Netscape and Sun Announce Javascript" (above, n. 14; emphasis added). On the early history of Netscape, see Robert Reid, *Architects of the Web: 1,000 Days That Built the Future of Business* (New York: John Wiley & Sons, 1997); Michael A. Cusumano and David B. Yoffie, *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft* (New York: Free Press, 1998); Joshua Quittner and Michelle Slatalla, *Speeding the Net: The Inside Story of Netscape and How It Challenged Microsoft* (New York: Atlantic Monthly Press, 1998); and Jim Clark, *Netscape Time: The Making of the Billion-Dollar Start-Up That Took On Microsoft* (New York: St. Martin's Press, 1999).

backbones serving academic research, was decommissioned, and the Internet as we know it today, unthinkable without private telecommunications carriers and Web-facilitated "e-commerce" and "B2B" or business-to-business transaction activity (to use two terms common in the mid- to late 1990s), began to take shape. America Online and Prodigy, up to that point private "online service" providers, also began offering access to the open Web. When the "Guide to the World Wide Web" created by Stanford graduate students Jerry Yang and David Filo was rebaptized "Yahoo!" and acquired the yahoo.com Web domain, large-scale Web indexing as a service was born.

Financial speculation linked to all these developments drove the Dow Jones Industrial Average past the 4,000-point threshold in February 1995 and the 5,000-point threshold in November, making two historic transitions in a single year. In this context, we are justified in remarking the larger context of the moment when JavaScript emerged as an instance of what I am calling "improvised expertise." Facilitated by new consumer-friendly electronic financial networks and services, so-called day trading by individual small investors would grow by the late 1990s into a widely publicized pastime. Day traders responded rapidly to intraday price movements and sought out (as well exacerbated) price volatility, buying and holding stocks for as little as a few minutes at a time and making a point of closing their positions at the end of each day. Commercial service centers opened to provide such traders with the network and PC hardware, software, and data and financial services then unavailable to home PC users. As a mode of improvised expertise permitting individual, often inexperienced and inexpert speculators to bypass both the authority and the fees of stockbroker and other expert (or at least certified) financial service providers, day trading was associated with the volatility of so-called Internet stocks and the improvised company creation and management practices of the dot-com bubble, and it was famous for the financial disasters such securities inflicted on day traders themselves, long before they triggered a US economic recession.[33]

The opening of securities markets to a new class of investor whose expertise was improvised, at best, was not the only significant economic event of 1995 and the years following it. It was in February 1995 that the 233-year-old Barings Bank, one of the world's oldest surviving financial institutions, collapsed due to losses incurred by a single Singapore-based derivatives trader who relied on the global

33. See Walter Hamilton, "Hooked on Speed: How Day Trading Works," *Los Angeles Times*, February 21, 1999, http://articles.latimes.com/1999/feb/21/business/fi-10174.

distribution of Barings's operations help him evade scrutiny of his activities. Billionaire business publishing executive Steve Forbes launched his campaign for the 1996 Republican presidential nomination, refusing matching funds from the US Federal Election Commission to avoid any obstruction in expending his personal wealth, a decision that would change US national electoral campaign financing for good by removing the relative financial restraint imposed by FEC funds matching. Also in 1995, a new, fully formalized international institution, the World Trade Organization (WTO), replaced the treaty structure known as the General Agreement on Tariffs and Trade (GATT) that dated to the end of the Second World War—an event that might be understood as economically stabilizing, were it not for the prompt eruption of disputes between developed and developing-economy members (the as-yet unresolved "Singapore issues") and the attention of antiglobalization activists, which would culminate in violent street protests at the 1999 Seattle conference.

1995 was not an uneventful year politically, either. US national political volatility increased as Speaker of the US House of Representatives Newt Gingrich, capitalizing on Republican success in the 1994 midterm elections, finished crafting the insurgent conservative legislation known as the Contract with America and forced the first of a series of US federal government closures in a dispute with President Bill Clinton. The nearly two-decade-long bombing campaign of Theodore John "Ted" Kaczynski, a former University of California, Berkeley mathematician who had simultaneously renounced modern technology and taught himself to construct primitive explosives (and who had targeted academic scientists and computer stores in particular) culminated with a series of explanatory letters and the publication of the so-called "Unabomber Manifesto" by the *New York Times* and the *Washington Post*. And Timothy McVeigh and Terry Nichols destroyed the Alfred P. Murrah Federal Building in Oklahoma City with a truck bomb in the United States' most significant act of domestic terrorism then and since.

While such details merely share a broad historical context with my topic of focus in this essay, the history of the JavaScript programming language, each of these details, from the emergence of newly privatized and newly publicly accessible Internet services, new economic governance institutions, and a new class of inexpert financial speculators, to what are still remembered today as very significant acts of domestic terrorism, involved conflicts and negotiations of technical expertise, in a broad sense, and some of them were marked

by such conflicts and negotiations in the narrower sense relating specifically to computers, as well. In that sense, that broader context cannot be separated entirely from my topic here.

JavaScript as Multiparadigm Programming Language

The Java-like language that Brendan Eich was commissioned to design for the Netscape Navigator web browser in 1995 (a task that he reportedly completed in ten days) was initially named Mocha and then LiveScript. It acquired the name JavaScript with the joint press release issued by Sun and Netscape in December of that year, which I have already mentioned. In the two decades since, Java applets have almost completely vanished from the web, and it is JavaScript that provides the main interactive element in browser pages. Sun and Netscape's joint press release reminds us just how far our current situation today is from the expectations they articulated in 1995. Though some of the rhetorical choices made in this text are perhaps more directly reflective of competition and licensing conflicts than anything else, it is worth dwelling on just how closely the respective domains of Java and JavaScript were positioned at the time:

> The JavaScript language complements Java, Sun's industry-leading object-oriented, cross-platform programming language.
>
> JavaScript is an easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers. While Java is used by programmers to create new objects and applets, JavaScript is designed for use by HTML page authors and enterprise application developers to dynamically script the behavior of objects running on either the client or the server.
>
> "Programmers have been overwhelmingly enthusiastic about Java because it was designed from the ground up for the Internet. JavaScript is a natural fit, since it's also designed for the Internet and Unicode-based worldwide use," said Bill Joy, co-founder and vice president of research at Sun. "JavaScript will be the most effective method to connect HTML-based content to Java applets."[34]

Java would be used to create code objects including applets (that is, small applications), and JavaScript programs would connect such objects and script (that is, configure and control) their behavior, providing them with a HTML-based user interface. If this particular separation of roles (Java as application programming language vs. Ja-

34. Quotations are all from the same source: "Netscape and Sun Announce Javascript" (above, n. 14).

vaScript as scripting language) is clear, the attention the press release also devotes to "server-side JavaScript" may cloud it somewhat:

> With JavaScript, an HTML page might contain an intelligent form that performs loan payment or currency exchange calculations right on the client in response to user input. A multimedia weather forecast applet written in Java can be scripted by JavaScript to display appropriate images and sounds based on the current weather readings in a region. A server-side JavaScript script might pull data out of a relational database and format it in HTML on the fly. A page might contain JavaScript scripts that run on both the client and the server. On the server, the scripts might dynamically compose and format HTML content based on user preferences stored in a relational database, and on the client, the scripts would glue together an assortment of Java applets and HTML form elements into a live interactive user interface for specifying a net-wide search for information.
>
> Java programs and JavaScript scripts are designed to run on both clients and servers, with JavaScript scripts used to modify the properties and behavior of Java objects, so the range of live online applications that dynamically present information to and interact with users over enterprise networks or the Internet is virtually unlimited. Netscape will support Java and JavaScript in client and server products as well as programming tools and applications to make this vision a reality.[35]

While there is no reason that two server-side programs (or for that matter, entire code bases) cannot maintain such distinctly complementary roles as are imagined here, the question of whether JavaScript might someday be able to perform alone in both such roles seems already latent in these formulations. Indeed, there exist unambiguous records of the tension around this issue, which it does not require much imagination to find in some of the joint press release's strained locutions, which read like a parent ordering two sibling children to get along. As Eich has put it, "If I had done classes in JavaScript back in May 1995, I would have been told that it was too much like Java or that JavaScript was competing with Java. . . . I was under marketing orders to make it look like Java but not make it too big for its britches. . . . [JavaScript] needed to be a silly little brother language."[36] Given that Java was already established, some at Netscape did not initially see any benefit in establishing and maintaining a separate language.[37]

35. Ibid.

36. Quoted in "Computing Conversations with Brendan Eich," YouTube video, 12:00, posted January 24, 2012, https://www.youtube.com/watch?v=IPxQ9kEaF8c.

37. Ibid.

Under such conditions, it is unsurprising that JavaScript was designed and implemented in haste, and with the hope that its shortcomings could be addressed in continued development. Eich brought a great deal of both organic and improvised expertise to bear on JavaScript's design. On the one hand, JavaScript's syntax is derived from the systems programming language C, by way of Java's own C-like syntax. On the other hand, Eich modeled JavaScript features like first-class functions and function closure on their equivalents in Scheme, a dialect of Lisp and so a member of a very different programming language family. Eich also adapted the prototype-based inheritance model of Self, a dialect of Smalltalk (a language and integrated programming environment designed in the 1970s for instructional and expressive computing), to provide JavaScript with object-oriented programming features. In combining these three quite different models—procedural or imperative in the case of C and Java, functional in the case of Scheme, and object-oriented in the case of Self—Eich made JavaScript a multiparadigm language from the very start.

Although JavaScript was not the first such multiparadigm language, such a synthesis is nontrivial in the design labor involved and in the prospects that JavaScript still presents for study even today. In synthesizing the three major programming paradigms, JavaScript certainly incorporated more complexity, even at the start, than most people would consider necessary in a language designed for novice and inexpert programmers. For novices, learning one programming model at a time (or only one model at all!) would certainly be considered more than enough. The tension between the design expertise that Eich brought to bear in creating JavaScript and its promotion as a language for inexpert users is especially interesting in Eich's own statements, which wholeheartedly endorse such promotion:

> [W]hat people wanted back then (and still want) is the ability to go one step beyond HTML and add a little bit of code that makes a web page dynamic—that makes things move, respond to user input, or change color; that makes new windows pop up; or that raises a dialog box to ask a question, with an answer necessary to proceed—things that HTML cannot express. That's really where you need a programming language, but something simpler than Java or C++. Content creation should not be recondite. It should not be this bizarre arcana that only experts and gold-plated computer science gurus can do.[38]

38. Marc Andreessen, "Innovators of the Net: Brendan Eich and Javascript," June 24, 1998, https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html.

JavaScript as Translational Programming Language

Despite borrowing most of its syntax from C and Java, JavaScript can certainly be written in a way that makes it resemble Scheme (see fig. 1). To its explicit synthesis of multiple programming models or paradigms (procedural or imperative, functional, and object-oriented), and to the divergent idiomaticity facilitated by the incorporation of features from very different languages, whose syntactic expressions may deform JavaScript's basically C-like syntax, we must add two other "multilingual" contexts for the development of JavaScript from 1995 to the present. The first is the development of the ECMAScript standard, implementations of (and deviations from) the standard in major web browsers, and the ongoing, both forward and backward "translation" by which the development of the standard, its implementation in browsers, and its use in web programming are mediated. The second is the compilation of JavaScript to other, often non-related programming languages. I will describe each in turn.

I have already mentioned the first edition of ECMA-262 (ISO/IEC 16262), Ecma International's specification for ECMAScript, which described ECMAScript as a scripting language "intended for use by both professional and non-professional programmers."[39] There exist seven published editions to date (ECMAScript 1, 2, 3, 5, 5.1, 6, and 7, excluding ECMAScript 4, which was abandoned), and the first edition's emphasis on design for nonprofessional users was retained all the way through the fifth edition published in 2009. The third edition published in 1999 included minor changes to the initial paragraphs of the section entitled "Overview" (section 4), changing "A scripting language is intended for use by both professional and non-professional programmers, and therefore there may be a number of informalities built into the language" to "A scripting language is intended for use by both professional and nonprofessional programmers. To accommodate non-professional programmers, some aspects of the language may be somewhat less strict."[40] The fifth edition published in 2009 deleted the latter sentence, leaving only "A scripting language is intended for use by both professional and nonprofessional programmers."[41]

39. *ECMAScript* (above, n. 20), p. 1.

40. *ECMAScript Language Specification: Standard ECMA-262*, 3rd ed. (ECMA, December 1999), p. 1, http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf.

41. *ECMAScript Language Specification: Standard ECMA-262*, 5th ed. (ECMA, December 2009), p. 2, http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf.

```
function Y(le) {
    return (function (f) {
        return f(f);
    }(function (f) {
        return le(function (x) {
            return f(f)(x);
        });
    }));
}
```

Figure 1. The applicative order Y combinator, from Daniel P. Friedman and Matthias Felleisen, *The Little Schemer*,[1] implemented by Douglas Crockford in JavaScript.[2]

1. Daniel P. Friedman and Matthias Felleisen, *The Little Schemer*, 4th ed. (Cambridge, MA: MIT Press, 1996).

2. "The Little JavaScripter," Douglas Crockford's World Wide Web, 2003, http://www.crockford.com/javascript/little.html.

The sixth edition published in 2015 made much more significant changes, which put meaningful distance between JavaScript's history and its present. To the paragraph defining a scripting language, a new sentence was prepended: "ECMAScript was originally designed to be used as a scripting language, but has become widely used as a general purpose programming language."[42] An entirely new paragraph was added elaborating this point: "ECMAScript usage has moved beyond simple scripting and it is now used for the full spectrum of programming tasks in many different environments and scales. As the usage of ECMAScript has expanded, so has the features and facilities it provides. ECMAScript is now a fully featured general purpose programming language."[43] Since the end of the ten-year interval separating the third and fifth editions, which saw the development and then abandonment of a fourth edition, feature addition has been rapid and extensive, with the fifth edition in 2009 and the sixth in 2015 both adding significant new features. After the long, partly fallow interval from 1999 to 2009, this rapid pace of development stimulated the development of a culture of experimental implementation in which features still only in proposed or only partially and nonbindingly approved form, in published drafts and other documents relating to the ECMAScript specification, were included in beta or developer versions of major web browsers, and

42. *ECMAScript Language Specification: Standard ECMA-262*, 6th ed. (ECMA, June 2015), http://www.ecma-international.org/ecma-262/6.0/.

43. Ibid.

eventually even in some user versions. Even before reaching a developer version of a web browser like Google Chrome, such features found their way into use through the mediation of source-to-source compilers (also called transcompilers or transpilers) that rewrote JavaScript code using experimental features in a form compliant with previously published editions of the ECMAScript specification (and thus guaranteed to work in user versions of browsers). At the same time, as other browser vendors (Microsoft with its Internet Explorer browser, and to a lesser extent Apple with its Safari browser) failed to implement all the features in already published past editions of the ECMAScript specification, transpilation was used to make JavaScript code uniformly executable across browser platforms.

Source-to-source compilation is as old as the history of higher-level programming languages themselves, with examples dating all the way back to the 1950s. The especially vigorous, even frenetic pace of such activity in web development and other JavaScript programming today merely hyperanimates the long history of *translation metaphors* through which the history of digital computing itself can be traced.[44] Yet JavaScript programming may well be unique and unprecedented in the range and scale of such activity, if not in its mere fact. It is appropriate indeed that the most widely used source-to-source JavaScript compiler used to rewrite JavaScript code to conform to different ECMAScript specifications has the name Babel.[45]

We have not mentioned the many other programming languages that provide the option to transpile to JavaScript in addition to their original targets. An authoritative list includes not only many variants best described as JavaScript subsets, supersets, or extensions (many of them with names relating to coffee, such as the large CoffeeScript family), but compilers that will take in code in C/C++, Java, Perl, Python, Ruby, C#, Scala, Clojure, OCaml, Haskell, and other both major and minor, older and newer languages and rewrite it in JavaScript.[46] Here too, it is unlikely that anything else of this range and scale has ever been seen in the history of software programming. In a domain that is and has always been defined by constant translation, JavaScript programming culture can be distinguished as exceptionally *translational*.

44. See David Nofre, Mark Priestley, and Gerard Alberts, "When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960," *Technology and Culture* 55:1 (2014): 40–75.

45. "Babel: A Compiler for Writing Next Generation Javascript," Babel, 2016, https://babeljs.io/.

46. Jeremy Ashkenas, "List of Languages That Compile to JS," GitHub, 2016, https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-JS.

Node.js

A major, perhaps *the* major factor in the rapid expansion of JavaScript's domain is the Node.js project, which dates to 2009. Node.js is a JavaScript runtime environment disembedded from the browser software application and written much as scripting languages like Python and Ruby are written: a developer may use command-line utilities, including a REPL (Read-Eval-Print Loop) and debugger, along with a software application text editor or Integrated Development Environment (IDE) to write and test programs locally on her or his own computer, but outside the browser application environment. The Node.js interpreter can run on a server, as well as on a desktop PC and in other software and networking contexts, and it can be embedded in a wide range of computing devices. While the earlier popularization of so-called Ajax (Asynchronous JavaScript and XML) programming techniques, closely associated with Google's Maps products, certainly got the ball rolling in this respect, it may be Node.js more than any other single factor that has transformed JavaScript from a scripting language initially used disproportionately to create "annoyances" like browser pop-up windows[47] into something approaching a full-fledged systems programming language. Arguably, the Node.js project has revived, reactualized, and then realized the promise of the forgotten chapter of JavaScript's history marked by Netscape's early imagination and exploration of server-side JavaScript applications. For the first time since the appearance of the World Wide Web and the software browser application, Node.js unifies so-called front-end and back-end web development, so that so-called full-stack developers, who write code for both user-facing and data processing components of an application, can use a single programming language for both tasks.

The applications of Node.js go beyond the server (though in that context, we should also mention the displacement of XML, as used in the earliest Ajax techniques, by the JavaScript-associated JSON [JavaScript Object Notation] data exchange format). Web application frameworks written in Node (Express.js and Meteor are two of the best known) have eroded the popularity of Rails, the Ruby-based application framework that dominated web development from the late 2000s onward. The Node-based Electron framework is increasingly used to develop platform-agnostic desktop GUI applications (that is, conventional applications that a user downloads and runs on her or his own machine, rather than using in a browser window). Node's popularity also explains Apple's inclusion in 2015 of JavaScript as one of the operating system languages of macOS (formerly OS X), us-

47. "Computing Conversations with Brendan Eich" (above, n. 36).

able for interapplication communication, as well as the inclusion of JavaScript "bridges" in the Apple iOS and Google Android operating systems for mobile platforms, enabling compilation to a JavaScript bound to the native system programming languages of those platforms (C, C++, Objective C/Swift, and Java).

The Frameworks

For the reasons mentioned above, it is Node.js, more than anything else, that has driven the recent hyperprofessionalization of JavaScript programming, removing the language quite decisively from that portion of its design origins that emphasized accessibility to inexpert and nonprofessional programmers. A development separate from the Node.js project, but of nearly equal impact on both JavaScript's expansion and its hyperprofessionalization, is the proliferation of other JavaScript-based web application frameworks *not* directly designed for or implemented in Node. The Ember.js, AngularJS, and React frameworks provide web developers with very sophisticated, unified abstractions of the three core browser technologies (HTML, CSS, and client-side JavaScript itself) that have made possible great leaps in both the sophistication of web applications and the creativity and productivity of those who write them. But they have also quite decisively propelled web development beyond the domain of accessibility imagined for the Web when it first appeared, which may have persisted in real terms for as long as ten years after 1995, in the sense that wage-earning web programming techniques could still be learned through casual and part-time training or retraining.

The professional construction and maintenance of websites today require both initial and ongoing training, and require a level of skill maintenance and retraining, that put them well out of reach for virtually anyone who is unable to devote her- or himself to its full-time pursuit—even academic researchers, excepting those who study and teach web technologies as a well-defined and well-developed technical research specialty. It is not the mere passage of time that makes the humanities-based wave of emancipatory hypertext theory of the 1990s, for example, seem so quaint,[48] and makes its reinstantiation in the present "digital humanities" movement seem so disingenuous or so guileless, depending on whom one asks. It is not that JavaScript no longer stands for technical improvisation, but that in what I have been calling improvised expertise, expertise is now both unambigu-

48. See, for example, George P. Landow, *Hypertext: The Convergence of Contemporary Critical Theory and Technology*, Parallax (Baltimore: Johns Hopkins University Press, 1992).

ously and unambivalently the agent of improvisation, instead of its object.

JavaScript Fatigue—and Other Futures

This shift has not been universally (or even broadly) welcomed. Indeed, the frenetic pace of change, if not the levels of skill required, is clearly a burden even to well-trained and experienced full-time professional developers. In early 2016, the phrase "JavaScript fatigue" began appearing in social media posts, blog writing, podcast discussion, and discussion at professional JavaScript developer conferences and briefly dominated such discussions as a collective preoccupation.[49] The developer who began circulating the phrase had lamented the "confusing nest of build tools, boilerplate, linters, & [other] time-sinks" that the individual and combined profusion of new language features, transpilers, frameworks, and other developer "tooling" (that is, custom applications for various programming tasks) represented: an entire preliminary phase of assessment and labor that was required before a JavaScript web application could even begin to be designed.[50]

It is this profusion, which many JavaScript developers find suffocating, that my intentionally lighthearted title "JavaScript Affogato" is intended to name, referring to the Italian dessert consisting of ice cream or other sweets "drowned" in espresso coffee.[51] A soberer assessment would be bound to remind us that at some level, howsoever mediated, this hypertrophy of labor, and all the structural and personal strains that go with it, reflect or perhaps coconstitute the extensive economic violence of the interval beginning in 2008 and continuing to this day. Professional programmers have been among the few labor-market beneficiaries of an era of austerity and generalized economic pain and suffering, and such luck is nothing if not equally a curse. Certainly the significant priority placed on JavaScript, in particular, and its concomitant growth during this period, reflects the priorities declared by investment patterns focused on the user-facing "app" as a cultural token, associated with making and build-

49. See Eric Clemmons, "JavaScript Fatigue," *Medium*, December 26, 2015, https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4#.c0ve3n241; and Calvin French-Owen, "The Deep Roots of Javascript Fatigue," *Segment Blog*, March 15, 2016, https://segment.com/blog/the-deep-roots-of-js-fatigue/.

50. Clemmons, "JavaScript Fatigue" (above, n. 51).

51. "JavaScript Affogato" is a variation on (and homage to) the witty phrasing of Reginald Brathwaite in a series of advanced, theoretically sophisticated books on JavaScript. See Reginald Braithwaite, *JavaScript Allongé* (Leanpub, 2013); and *JavaScript Spessore* (Leanpub, 2015).

ing things as "free" labor (free as in freedom) and adaptation to austerity, and as a new object of financial engineering.

In 2014, Brendan Eich's resignation as CEO of the Mozilla Corporation, only nine days after taking the position, was described by a writer for the *New Yorker* as "the least surprising C.E.O. departure ever," given that Silicon Valley was "a region of the business world where social liberalism is close to a universal ideology."[52] (Eich's having donated to an anti-marriage equality campaign supporting California's ballot Proposition 8 in 2008 was a fact known beforehand, which became newly controversial upon Eich's appointment.) It might be more accurate to say that in a broader context, the episode reflects the fundamental confusion of the specifically cyberlibertarian politics of Silicon Valley investment and management culture, which borrows ideas freely, but mostly unreflectively and unsynthetically from both statist left-wing and anti-statist right-wing political platforms, in ways that seem to reflect the startup culture's ambivalence about its own expertise. Though a great deal of work remains to be done to articulate the meaning such contexts lend to topics such as my own, in this essay, such social dynamics cannot be delinked from the technical history of the artifacts designed and produced in contexts determined by them—even, or perhaps especially, such an artifact as a programming language.

52. James Surowiecki, "How Mozilla Lost Its C.E.O.," *New Yorker*, April 4, 2014, http://www.newyorker.com/business/currency/how-mozilla-lost-its-c-e-o.