Translating the Future: Transpilers and the New Temporalities of Programming in JavaScript

Andrew Pilsch

This essay is about transpilation and the future of translation work done by machines. “Transpilation” is a particularly ugly portmanteau word that refers, in web development, to a particularly confusing new concept used in building online JavaScript application. Mashing together “translation” with “compilation,” it refers to the process of translating one human-readable computer programming language into another. While compilation without translation refers to the conversion of human-readable programming languages into the digital codes understandable by computers, the end product of transpilation is another human-readable language.

Transpilation has a variety of uses that I discuss below, but it has become a prominent topic in web development because of its current use in JavaScript development. The only programming language that runs in all modern web browsers, JavaScript is the language in which the dynamic web is now written.[[1]](#endnote-1) As JavaScript has matured, the various corporate factions that participate in the language’s standardization have demanded legacy compatibility, meaning that all new versions of JavaScript should be executable by older versions of the language’s compiler.[[2]](#endnote-2)

However, the newest version of the language, originally prototyped as ECMAScript 6 (JavaScript is officially standardized by the European Computer Manufacturers Association as “ECMAScript” because “Java” as a programming language name is trademarked by Sun Microsystems[[3]](#endnote-3)) but now released as a series of yearly versions (ECMAScript 2015, ECMASCript 2016, etc), significantly departs from this vision of legacy compatibility and introduces new language syntax that will not execute in old browsers. Large parts of ECMAScript 6 (ES6), the abbreviation still used in development circles to refer to any of the new, yearly JavaScript releases, are not even implemented in any existing browsers. Yet, at the same time, they are being widely used in production applications across the Internet.

It is possible to write code in a language that has not been implemented and therefore does not exist yet because of transpilation. Babel, an extremely popular JavaScript transpiler, translates code written in ES6 into the previous version of the language, ECMAScript 5 (ES5). Babel’s slogan is “Use next generation JavaScript, today,” capturing the power of the program: it makes a future version of the language available in the present.

In this article, I consider transpilation as an example of machine translation in the present. Where “machine translation” is usually used to refer to the automatic, algorithmic translation of a *human* language into another *human* language, I propose that what Babel does when transpiling ES6 to ES5 is similar enough in practice to what Google Translate does in rendering English into Spanish that considering the two phenomenon together can reveal something interesting about the nature of translation in the present.

Compilation has historically, in ways that will perhaps give many translators fits, been discussed in terms of translation (textbook discussions of compiler design write about “translating” and “rendering grammars”). Similarly, the rise in accuracy of ubiquitous digital translation services has meant that translation, as Rita Raley suggests, “has become an ordinary, everyday practice” available at the push of a button.[[4]](#endnote-4) Transpilation points to an interesting conversion between these two fields: translation is becoming more automatic as compilation becomes more translational.

This convergence is also particularly suggestive of the changes wrought to the temporality of translation. Vilém Flusser has argued that technological progress appears to move in a future-to-present fashion after the completion of the Enlightenment (which appeared to move past-to-present). Flusser further argues that this turn to the future is the product of the decline of writing and the rise of what he calls coding for the apparatus. These processes of thinking with machines represent new modes of communication and thought that occur in concert with and are ultimately addressed to machines.

As I expand on all these above points below, I also consider the role of standardization in the authoring of JavaScript. Programming languages diverge from human languages in that they are managed, having some relationship to planned languages such as Esperanto in this way.[[5]](#endnote-5) However, because they are managed, we can trace how this strange time-travelling translation arrived by considering the negotiations that went into the standardization of JavaScript.

Ultimately, this essay concludes that translation done by machines has already produced strange temporalities and will continue to produce even stranger ones, if ES6 to ES5 transpilation in JavaScript is any indication. Surveying the near-future of machine translation in *Translation in the Digital Age*, Michael Cronin suggests that “if we are moving to ‘the unsteady symbol’ of the future, chief among those symbols will be the ever-changing, self-renewing figure of translation.”[[6]](#endnote-6) Within this futurity, Cronin discusses the relationship between automated translation systems and the rise of what Milad Doueihi calls “digital humanism,” suggesting that translation will, as it has in analog versions of humanism, play a key role in shaping temporal and spatial constructions of power.[[7]](#endnote-7) As Flusser shows, however, this new, digital humanism involves an increasing proliferation of non-human agents and, to understand these temporal and spatial constructions of power as they are shaped by translation, we must also consider how translation is reshaping non-human codes.

## The Longtail Of Web Time

JavaScript is a programming language originally designed to run within a web browser. It was first released in 1995 for the Netscape browser. Brendan Eich wrote the first version of the language in ten days—which, for a computer language, is an *extremely* small team working *extremely* fast.[[8]](#endnote-8) The language has always struggled with the limitations created by this compressed timeline, even as JavaScript has matured and gained new features and capabilities. Despite often being buggy and quirky, programmers who wish to build complex applications online—such as Google Mail or Facebook—*have* to use JavaScript and therefore end up developing a variety of libraries, tools, patches, and quick-fixes to supplement the language’s often flawed syntax. This ecosystem of tools and patches has made JavaScript into a kind of testing ground for novel or experimental ideas in computer science. Such experimentation inspires languages such as TypeScript, CoffeeScript, and Clojure that compile into JavaScript as well as the Node.js project, which uses the open-source JavaScript interpreter from Google’s Chrome browser to run JavaScript in environments outside the web browser, allowing web developers to write the server-side (the backend of an application) and the client-side (the front-end of an application) in the same programming language. Such innovations have radically altered the nature, scope, and complexity of web development in the present. They have also raised JavaScript’s profile as a serious computer language, after having been dismissed as a “toy” for much of its early existence.

Beyond experimentation, patching and extending JavaScript is necessary because JavaScript developers grapple with what Susan Leigh Star calls “the inertia of the installed base.”[[9]](#endnote-9) Studying the development and standardization of infrastructure, Star finds that “infrastructure does not grow de novo;” instead, it “struggles” with already existing technologies that precede it. This is why, for instance, “optical fibers run along old railroad lines.”[[10]](#endnote-10)[[11]](#endnote-11) Despite the breathless rhetoric of rapid progress that accompanies the changing nature of online life and the pace at which web technologies advance, JavaScript has a surprisingly strong inertial pull while simultaneously evolving at a rapid clip. In the particular case of JavaScript, this infrastructural struggle is prominent because, unlike most programming languages, any computer with a web browser is in possession of a copy of JavaScript’s compiler.

The tools to convert human-readable programming language into executable, machine-readable digital code are normally only installed on the computers of IT professionals, either programmers or the engineers and technicians who maintain information systems. These professionals tend to be familiar with the need to regularly upgrade their tools to make sure they maintain compatibility. Moreover, if code breaks for such skilled users, they will know that it may be a result of outdated tools.

JavaScript is different because every web browser is also equipped with a version of the program that converts JavaScript code into digital instructions for the computer to process. Code that is written for a newer version of JavaScript than is present on a user’s computer will not run. Moreover, there is no easy way to signal that such a break is being caused because the user’s browser is old. Thus, unlike other programming languages, Star’s “inertia of the installed base” is more pronounced with JavaScript as out-of-date implementations tend to accrue online.

This inertia of the installed base is most evident in the seemingly interminable discussion in JavaScript developer communities about whether or not to continue supporting Microsoft Internet Explorer 8 (IE8). Released in 2009, IE8 is a browser widely noted for its inability to support many basic web standards, due to its emerging when Microsoft and Netscape were using browser implementations to fight over the future of web standards. The browser, which is also full of security holes, is officially no longer supported by Microsoft as of 2016. Problematically, the browser is also as far as Internet Explorer can be updated on the Windows XP operating system. IE8 continues to be used by around 17% of all web users.[[12]](#endnote-12) As of 2013, 36% percent of Chinese Internet users run IE8, which is enough for it to be the most popular browser in China.[[13]](#endnote-13) The continued use of Internet Explorer 8 and Windows XP, especially in Asia, means that web developers often have to balance using new features of the language with supporting a potentially large segment of their audience for whom these features will cause the site to work incorrectly.

As a result, many new features introduced into JavaScript are often met with trepidation by JavaScript developers. *This will be great when I can support it in 10 years, after the majority of my users have upgraded their browsers*, has been the overwhelming attitude. However, to combat some of these issues, developers now use “polyfills,” programmer Remy Sharp’s term for bits of code that patch missing pieces in a browser’s JavaScript implementation.[[14]](#endnote-14) As an example, the most famous polyfill is a library called html5shiv, initially developed by Sjoerd Visscher. In versions of Internet Explorer before IE9, certain new HTML entities (such as <aside> and <header>) would not be styled by CSS. html5shiv uses JavaScript to clone these elements so that they will receive style in old versions of Internet Explorer. Thus a website that would not display correctly on an old browser is patched by JavaScript so that it works correctly.

The ability to polyfill a missing feature is possible because JavaScript has been upgraded in ways that do not change the language’s syntax. Most changes to the language instead come as new methods added to existing objects, such as the .map() method that was added to Array in ES5. If a developer used the Array.map() method in their code and a user with an older browser attempted to execute the code, the .map() method would be missing and the code would fail to run. However, because of the way JavaScript works, a programmer could write their own implementation of the .map() method and add it to the Array object if it is missing. Patching the Array object in this way would be another example of a polyfill. Thus new changes can be added to code on an as-needed basis.

To maintain compatibility this ability to polyfill legacy browsers, the JavaScript standard-making body has favored changes that do not alter the language’s syntax. However, many of the features in ES6 do make these sorts of changes. Syntax changes cannot be polyfilled and will not execute correctly with older versions of JavaScript because of how computer languages are compiled into machine instructions.

When the Array.map() method is used but not present, an execution error will occur. This type of error occurs after the program has been compiled from human-readable JavaScript to machine-readable digital codes. Polyfills, as code also has to be compiled before it can work its solution, can only fix execution errors. An error in syntax, say using a feature of ES6 before it has been implemented, occurs instead during the process of compiling JavaScript to these machine-readable commands. To use human language as an analogy, a polyfill can fix errors in interpretation, where my audience does not understand what I am saying after I have said it. A syntax error would be analogous, instead, to a sentence I spoke or wrote that violated the grammar of my spoken or written language. In the case of an execution error, I can fix things by providing more information; there is nothing I can do about my syntax errors. While they are not errors, as such, older syntax analyzers will not be able to understand, on a grammatical level, the ES6 syntax and will mark it as an error, which makes the whole program behave erratically.

Hence the need for a transpiler. Unlike a traditional compiler, the output from a transpiler is not machine-readable code but is, instead, a different human-readable language. In the programming language Python for instance, the release of version 3 changed syntax in ways that meant a lot of code that worked with the version 2 interpreter would now produce syntax errors at compilation. A transpiler was used to facilitate the translation of new code to work with older libraries that had not yet switched to Python 3. Despite being incompatible with one another’s interpreters, the rules by which Python 3 differs from Python 2 are logical and can translation between them can be algorithmically implemented.

Transpilers are novel because, unlike compilers which produce machine-readable digital codes as output, these translating compilers produce another human-readable program, thus making it possible for computer programmers to automatically translate between two versions of a language. As part of a major overhaul of the language, ES6 is intended to implement a variety of features suggested by both modern programming languages that have emerged since JavaScript was devised in 1995. These new features, however, are syntax changes that cannot be patched with polyfills.

As an example, ES6 introduces a feature common in modern programming languages called “default function parameters.” A function in computer programming is a conceptually linked block of code that can be called repeatedly from different locations in the program and can be provided with different input variables to change the output. Take for instance a simple function in JavaScript to say hello to a person:

function sayHello(name) {  
 return 'Hello there, ' + name;  
}

In this function the name parameter is concatenated to the string “Hello there,” such that running sayHello('Andrew') will return the output “Hello there, Andrew”.

If we run sayHello() without the parameter, the function returns “Hello there, undefined” because undefined is the default value for any missing data in JavaScript. If we were writing a more complex function, this behavior would probably be a problem. To solve it, many modern programming languages rely on what are called default function parameters to declare in the function definition itself what each parameter will have as its value if the user does not provide one.

JavaScript has not had this feature before ES6. Instead, developers had to use the following confusing and strange looking command to set default values for parameters:

function sayHello(name) {  
 name = name || 'default person';  
 return 'Hello there, ' + name;  
}

In this example, name is re-assigned at the first line of the function. Its value is being set to the result of a logical test that could be translated to English as “myself OR the string ‘default person’.” If name is undefined, the first clause of the test (“myself”) will fail and name will receive the second value in the logical OR operation (in this case, 'default person'). Despite being a solution that works in currently implemented versions of the language, this compromise introduces readability problems for more complicated functions. To solve this widespread problem, the developers of ES6 introduced default function parameters into the language standard.

In ES6, the sayHello function can be declared:

function sayHello(name='default person') {  
 return 'Hello there, ' + name;  
}

Here, sayHello() will return 'Hello there, default person', as the person being said hello to has the name of “default person,” the default parameter value in the function declaration.

In an out-of-date browser, this function will produce a syntax error because the compiler does not understand the default parameter declaration—='default person'—as grammatically valid. To solve this for browsers that have not implemented it (which at the moment include Opera and Microsoft Edge), a transpiler can algorithmically process the ES6 code above into something that resembles the ES5 compatible version with the uglier syntax (name = name || 'default person').

While ES6 transpilers are useful for solving legacy compatibility problems like the one above, why they are so widely used in development is that some of the most advanced features of ES6 (such as classes) are not implemented in *any* production browsers. With the most ambitious feature of ES6—the importing of external modules—browser engineers are not even sure *how* the feature will work, despite it having been approved in the standard. While features such as classes and module imports will arrive in the future someday, both are used in production applications today, because the futuristic code is being transpiled by Babel or Traceur.

This futurity, as I discuss in the next section, potentially radically refigures the obsession with legacy compatibility that has been the historical driver of practices of industrial standardization in general, the term for the set of principles by which corporations manufacture economic stability and consensus in the post-national era of geopolitics. Though programming languages are one facet of infrastructure standardization, the new temporal rhythms (from future to present, instead of past to future) unlocked by transpilation have the potential to radically change how standards are made and how they function as codifications of power in the present.

## Futures in the Empire of Standards

In *Optical Media*, German media philosopher Friedrich Kittler explains that “after 1880 we find ourselves in an empire of standards.”[[15]](#endnote-15) He uses “standards” in contrast to “norms” and “laws.” For Kittler, standards “distinguish those aspects of the regulations that are intentional from the accidental or the contingent.”[[16]](#endnote-16) Kittler’s distinction between standards and earlier forms of norming behavior can be exemplified by the difference between the metric and the British measurement systems. The older British system (still used in the US) is defined generally in reference to custom and often in vague reference to the scale of the human body or conventions that developed organically dating back to Rome. This would be a norm in Kittler’s formulation: systems that are in no way coherent (“How many cups are in a gallon?”) but that have been established through consensus and practice. Metric, now properly the International System of Units (SI), is a coherent, decimal-based set of units. Units are defined in terms of one another and scale based on fixed, powers-of-ten based prefixes (milli-, kilo-, etc.). This system is a standard; where the British system is governed by social convention, SI is governed by a body and is meant to facilitate the work of measuring a variety of factors in a consistent manner.

The function of standards, as Kittler alludes to in naming our era “the empire of standards,” is to govern inter-corporate relations in an age of post-national capital and weakened nation states. Media studies scholar Jonathan Sterne strongly supports this argument in *MP3: The Meaning of a Format*, where his account of the MPEG working group and the application of perceptual coding to the creation of a standard for audio compression becomes, also, a story of how “the politics of standards may eclipse the governmental regulation of broadcast or telecommunications as a crucial site where policy happens.”[[17]](#endnote-17) This is because standards-making bodies set the patterns by which international exchange happens but often do so by managing a variety of competing corporate interests, usually without the force of any geopolitical clout.[[18]](#endnote-18) Within this empire of postnational corporatism, three main themes have developed in the growing body of work on the social and historical ramifications of international standardization: the desire for control, the quest for conformity, and the inevitability of compromises.

This desire for control manifests as economic control. As Sterne writes, in the emerging digital audio marketplace, “manufacturers worried about a bottomless soup of competing standards and protocols creating a market that was too *unpredictable*.”[[19]](#endnote-19) The classic, oft-repeated example was the costly and protracted industrial war between the Betamax and VHS video casette standards in the 1980s that cost all parties involved a significant amount of money and, more importantly, consumer goodwill. In their ground-breaking essay on the historical study of standards, Amy Slaton and Janet Abbate intensify the link between standards and power, arguing that “standards bring economic control to their users” and that the study of their creation and use is an extension of both the history of technology *and* the history of labor.[[20]](#endnote-20) Such standards give their authors (and corporate sponsors) considerable “control of revenues and profits” and “of the condition of production” through “the deliberate design of work processes.”[[21]](#endnote-21)

Similarly, standards enforce conformity not just of products and processes but of people as well. As Slaton and Abbate show, the history of standards is the history of favorable re-articulations of the “divisions of manual and intellectual labor.”[[22]](#endnote-22) Their two case studies (of the conversion from brick to concrete construction at the turn-of-the-20th-century and the rise of TCP/IP as the standard for Internet communication) trace how, in both cases, standards were used to favorably manipulate labor conditions around the contemporaneous labor blockages faced by corporate authors, for instance concrete construction was favored because it shifted construction from skilled (and unionized) bricklayers to unskilled concrete pourers. Susan Leigh Star’s work in STS on standards and infrastructure also consistently highlights this conformity but in a variety of contexts beyond labor. She suggests that standards are always involved in the “distribution of the conventional” and mandate the creation of a fixed population who count, distinct from those who are rendered invisible by failing to conform. She first documents this phenomenon by capturing her difficulties in ordering at McDonald’s due to an onion allergy before moving on to issues of gender identity, disability, and physical suffering; throughout, Star documents how standards recede into the background (if we let them) and pull their invisible barriers to entry out of sight as well.[[23]](#endnote-23)

In addition to control and conformity, the social scientific and humanistic literature on standards highlights the messy and bad futures standards produce through their inevitable function as objects of compromise. Discussing the fidelity to reality projected by optical media, Kittler remarks that “media standards are still a commercial compromise that reveals deficits, such as black-and-white images, no stereoscopic effects, or missing colors like the American NTSC television system.”[[24]](#endnote-24) As Kittler draws out, a standard—whether for a media format, a computer language, or the size of a box—is always the product of market-driven compromises and therefore never perfect. Similarly, Sterne documents this phenomenon in tracing the needlessly byzantine complexity of aspects of MP3 that resulted from the standard-making body’s compromise between two competing formats to satisfy two consortiums involved in the creation of the MPEG standard.[[25]](#endnote-25) Perhaps most shockingly, Donald MacKenzie has traced how the standard for computer arithmetic “had to be negotiated, rather than deduced from existing human arithmetic” due to a variety of preexisting arithmetic units that all interpreted basic math slightly differently.[[26]](#endnote-26)

Trevor Pinch summarizes these points and offers a sentiment often repeated in the literature on standards:

Standards are rarely simply technical matters; they are powerful ways of bringing a resolution to debates that might encompass different social meanings of a technology. Standards are set to be followed; they entail routinized social actions and are in effect a form of institutionalization.[[27]](#endnote-27)

Moreover, the three factors show up in computer science and business literature as factors affecting the quality of standards. Computer scientist Kai Jakobs concludes that “common wisdom has it that consortia move faster, are more flexible and more business-oriented, and that they are thus destined to come up with really useful solutions very quick” but, in actuality, due to the social and political factors frequently elaborated in the social scientific and humanistic accounts of standardization, contemporary technical standards often “never really lived up to the high expectations that initially surrounded [them].”[[28]](#endnote-28) Due to competing industrial interests, the restrictive nature of *any* standard, and the need for compromise in the absence of government arbitration, technical standards often underwhelm.

In many ways, this account of standardization is the history of JavaScript. For instance, considering the Document Object Model API (DOM) that JavaScript provides developers to interact with an HTML document, computer textbook author Elliotte Rusty Harold explains that

There’s a phrase, “A camel is a horse designed by committee.” That’s a slur on a camel. A camel is actually very well adapted to its environment. DOM, on the other hand, is the sort of thing that that phrase was meant to describe.[[29]](#endnote-29)

Harold enumerates the variety of things DOM was meant to do (and arguably does not do very well) before concluding that “within the constraints that they were operating under, they failed.”[[30]](#endnote-30)

Within an environment in which standards attempt to create exciting and new technologies but instead generate unexciting, unambitious, difficult-to-use products, ES6 is potentially unique. Unlike other standards-making exercises, such as the creation of the MPEG standard (which was initially a contest between fourteen pre-existing technologies backed by a variety of corporate alliances), the working group responsible for standardizing JavaScript, Technical Committee #39 (TC39), approached the ES6 with an eye toward adding oft-requested features that might break the syntax, even if this sacrificed legacy compatiblity.[[31]](#endnote-31) Where previous versions of the language resulted from competition between corporate interests such as Microsoft and Netscape, ES6 is rhetorically situated by its creators as being *for* users, through time-saving and innovative features. The ES6 standard documents a new future for the language rather than a series of corporate compromises.

While there is some research on the relationship between future creation and the process of standardization, most of it is tied to the larger critique of corporate power in the post-national era of neoliberalism. Sterne notes that the documentation of the MPEG standard “gives a clue to the level of confusion […] as to what, exactly, the transmission of digital audio was *for*.”[[32]](#endnote-32) The committee creating the standard had to imagine applications for the technology and these projected futures mostly conformed to contemporaneous uses of audio (radio broadcast and commercial audio production). They never guessed at MP3’s widespread usage for computer audio.[[33]](#endnote-33) Similarly, Lawrence Busch’s account of the empire of standards and its role in “collectively creating a future that consists of standardized differentiation” shows how businesses generally encounter the future in terms of risk prediction and through errors in their predictive models.[[34]](#endnote-34)

Standards are designed to manage the risk of the future. Such risk management often comes at the cost of meaningful innovation. ES6 represents a potentially novel moment in the history of standardization: a standards-making body attempting to make a radical break from the inertia of its own installed base. Breaking from legacy compatibility is possible because of transpilation and the ability to translate a language from its future version into the present.

## Future Code / Code Futures

While we might question whether any computer program actually exists,[[35]](#endnote-35) ES6 is novel for not even meeting the minimal existence of any computer program. TypeScript, which transpiles into JavaScript exists as a programming language. There is a specification and documentation for it, in addition to a set of tools to turn it into executable code. As I have been stressing in this piece, ES6, unlike TypeScript, does not exist in the same way.

While in the process of developing ES6, the naming convention of the language changed. Prior to 2015, JavaScript was versioned in sequential numbers (EcmaScript 1, EcmaScript 2, etc.). In 2015, TC39 decided to release part of the specification as EcmaScript 2015 (abbreviate ES2015), thereafter switching to yearly releases. Yearly releases allow working features to be standardized when they are ready, rather than waiting for the entire feature set to be approved (which often takes years). Despite the new versions of JavaScript being released as ES2015, ES2016, and ES2017, the old naming convention, “ES6,” is still used by developers to refer to all three of these proposed and released standards (as well, presumably, as ES2018 and ES2019). “ES6” vs “ES5” represents a massive sea-change in the mentality of JavaScript (and also necessitates the use of transpilation in everyday programming). “ES5” vs “ES2015” vs “ES2016” speaks to the success of this sea-change and the newly emerging futurity of JavaScript development. So while ES6 does not designate a version of the language that will ever be released, it is still helpful to think of all the yearly releases that introduce syntax-altering changes as ES6 to conceptually link them.

Bearing this naming change in mind, part of the proposed ES6 specification was released as ES2015 in June 2015. Some ES2015 and ES2016 (which was released in June 2016) features are still unimplemented in browsers. Additionally, the features planned for ES2017 and ES2018 only exist as proposals. Yet, at the same time, thanks to transpilers such as Babel and Traceur, there are thousands of developers all over the world writing code and using these proposed features to make production web applications. This pattern of use complicates the question about the real existence of computer languages, given that real production code is being written right now in a language that does not officially exist yet.

For instance, recent versions of Facebook’s popular React framework recommend the use of ES6 to author code (and Facebook does this on their site, as well). React is particularly interesting for thinking about the existence of ES6 as certain features—specifically, static class variables—are not in the recently approved ES2015 standard; they have been merely proposed for inclusion in future versions of the language (specifically in the draft ES2017 standard). For convenience, many tutorials for React recommend enabling a feature of the Babel transpiler called “stage-0.” Stage-0 is the straw-man stage of approval by TC39; features at this stage are on a list TC39 might like to consider at some point in the future. Though not seriously considered or even voted on, applications can use these features through Babel, though the documentation offers an ominous warning (under a red “Subject to Change” banner): “These proposals are subject to change so *use with extreme caution*.”[[36]](#endnote-36) Babel not only allows for the early implementation of ES2015 and ES2016, it allows the use of features that may never actually appear in any future JavaScript release. ES6 transpilers are speculative programming tools and ES6 only exists, at least in parts, in the future.

Walter Benjamin argues that translation is a “mode,” a way of thinking with the text and the two languages to be connected in the act of translating.[[37]](#endnote-37) In thinking with a transpiler, ES5, and ES6, I want to think more directly about the role of speculation in that mode. A mode of guesswork has always operated within translation. Benjamin, writes that “any reference to a certain public or its representatives [is] misleading, but even the concept of an ‘ideal’ receiver is detrimental in the theoretical consideration” of translation.[[38]](#endnote-38) While translation is never fruitfully considered from the perspective of an ideal reader, the translator, as Benjamin unpacks, must still speculate about the nature of readers (for Benjamin these guesses are guided by the shape of the text). Thus translation for Benjamin involves speculation about the future and the readers it might contain. Similarly, Michael Cronin has traced a history of translation amongst Irish nationalists in the 1840s that used translation to imagine Ireland as a homeland-in-the-future *and* to position it as a part of a broader, imaginary European intellectual and revolutionary foment.[[39]](#endnote-39) Translation in both cases is constructed as an activity that is partly in dialog with the future.

Extending Benjamin’s translational mode, Vilém Flusser views translation as “a specific way of thinking, writing, and living,” according to Finger, Guldin, and Bernado’s *Introduction*.[[40]](#endnote-40) For Flusser, who published in four languages during his life, and whose theory of translational nomadism analogized his own nomadic biography, translation was the process of continuous re-translation, a form of “explication” in which “meaning is homeless and itinerant.”[[41]](#endnote-41) Flusser would often translate his work himself between the languages he wrote in as a process of continually exploring the limits of what he meant. Finger et al describe this as a process of “slowly circling around…, taking a series of distinct pictures” in an attempt “to describe the object from as many viewpoints as possible.”[[42]](#endnote-42) For Flusser, languages are particular lenses through which to view the world, with each language providing new insights into the topic at hand.

However, as Flusser developed his theory of translation, he came to see language as a limited boundary for translation. In “What is Communication?,” “human communication is an artificial process. It relies on … symbols ordered into codes.”[[43]](#endnote-43) These codes—he mentions the “code of gesture” alongside spoken and written language—produce “the codified world in which we live,” removing us from “‘first nature.’”[[44]](#endnote-44) Human communication then happens by translating thought into and between a variety of code systems, not just language.

In *Does Writing Have a Future?*, Flusser enfolds the non-human into his theory of translation between code systems. In describing a process throughout Flusser’s *oeuvre* that I have taken to calling “thinking with,” Flusser describes communication as “a complex feedback loop between technology and the people who use it. A changing consciousness calls for a changing technology, and a changing technology changes consciousness.”[[45]](#endnote-45) This loop between non-human technology and human consciousness happens through both human and non-human code systems and is coming to prominence in the present (even more today than when Flusser was writing in 1987) because the written project of the Enlightenment, explication, is complete and nothing new remains to be explained.[[46]](#endnote-46)

For Flusser, this completion marks a radical change in the nature of time as well as the nature of inscription. Specifically, in the era of code, our concept of progress changes: society moves “no longer from the past toward the future but from the future toward the present.”[[47]](#endnote-47) In this new mode of thinking, “*future* turns into multidimensional compartments of possibilities” and “digital codes are a method of making these compartmentalized possibilities into images” in the present.[[48]](#endnote-48) If we hope to avoid “a descent into illiterate barbarism” as technical image replaces written inscription, Flusser suggests that we must develop “a theory and philosophy of translation” that can translate our current mode of thinking with language into a broader accounting of thinking with technical apparatuses in order to make “a conscious step beyond current conditions of thought and life.”[[49]](#endnote-49) Flusser concludes that if human civilization is to survive, the linguistic nomadism at the core of his theory of translation has to be extended to non-human codes.

I see in ES6 transpilers a first glimpse of the new theory and philosophy of translation Flusser calls for. These transpilers bring a language from the future into the present through the power of digital codes. They disrupt the normal rhythms and associated practices of translation but retain the fundamental nature: the movement of thought between code systems.

In discussing algorithmic translation in art and machine translation in business, Rita Raley argues that “fidelity to the original in the instance of an algorithmic translation … is a fidelity to the virtual, a fidelity to the *idea* of the original, rather than the thing itself.”[[50]](#endnote-50) The concept of a virtual original is especially suggestive to the transpilation of ES6, where even the idea of an original is virtual. There is no possibility, as I write this, to find an original for ES6 other than the virtual version defined in the standard. When I write ES6 code, I am programming for an apparatus that may yet exist in the future. I am engaging in the temporality of the technical image as Flusser understands it.

We see these technical apparatuses appear in a number of considerations of the machine translation of human language. Raley considers the variety of web artists using “Google as an authoring environment” for its potentials to deform language in novel ways (due to the idiosyncratic nature of machine translation).[[51]](#endnote-51) She concludes this analysis by asking whether we ought to consider these machinic tools as coauthors for the art they have helped create. Similarly, Cronin tells the story of Martin Kay’s “translation amaneunsis” as an example of new approaches to machine translation. Rather than build systems to do the full work of translation, Kay argued that researchers should pursue expert systems that could support translators’ work rather than trying and often failing to automate the entire process of translation.[[52]](#endnote-52) For Cronin, Kay’s critique praises machine translation for “the entirely reasonable ambition to automate certain sub-routines in the translation process” while critiquing existing systems for their “indifference to details.”[[53]](#endnote-53) Raley’s consideration of machinic co-authors and Kay’s amaneunsis imagine translation as a dialog between human and machine, a human-machine hybrid translator in which questions of detail are dealt with by experts while the routine tasks are automated by the machine.

Flusser extends this idea of a cyborg translator to *all* notions of communication through apparatuses. For Flusser, writing is the process of rendering linear the nomadic cycles of thought. He imagined in 1987 that in the future, “word processing” will come to mean the algorithmic processes that render our thought linear for us. He suggested that word processors will someday be “grammar machines, artificial intelligences that take of this order on their own,” freeing humans from the burden of rendering thoughts into lines.[[54]](#endnote-54) He imagined that communicating thoughts in the future will always involve a version of Kay’s amanuensis, and that writing in the future will be done by “those who manipulate the apparatus, setting the new signs into electromagnetic fields.”[[55]](#endnote-55) Such writers will “write with and for apparatuses,” that “writing has changed for these people; it is another writing, in need of another name: programming.”[[56]](#endnote-56)

This human-machine continuum of translation is, according to Raley, “the new linguistic doxa” of our age.[[57]](#endnote-57) Cronin similarly connects human-machine collaborative translation to the idea, similar to Flusser, that something profound has shifted in the nature of language. For Cronin, translation is the fundamental logic of the digital world:

The possibility that text, image, and sound can be converted/translated into digital code means that representations, identities, and objects become inherently unstable as they can potentially be converted/translated into anything else, emerging as new objects or circulating in next contexts.[[58]](#endnote-58)

As with Raley’s example of the “virtual original,” Cronin shows how *everything* today is a virtual original because so much of what once counted as discreet genres of human output (sound, text, image, video, etc.) are just electrons swirling on spinning platters somewhere. When everything is just a stream of bits, we find ourselves “in an age of potentially endless translation.”[[59]](#endnote-59)

Despite the potentially endless translation of digital code, there is still something uniquely odd about ES6 transpilation. As Robin Boast vividly explains in *The Machine in the Ghost*, digital code, unreadable by humans and software alike, is constantly transferred between software and human agents as it is progressively processed for display and editing.[[60]](#endnote-60) However, rather than moving from a human reader to a machine reader, the transpiler converts from one human readable code to another using the principles of compilation, which traditionally converts human code to machine code. This extra loop through human-readable language makes transpilation a singular digital practice.

Additionally, without this extra act of translation, ES6 would be not be usable by the computer, given that the language only exists in the future. By translating from the future into the present, ES6 is a widely used professional programming language. This shift to a speculative, future-oriented translation of human code has also altered the temporal rhythms by which JavaScript is standardized.

## Standard Times, Standard Futures

A recent article in *Smashing Magazine* outlining experimental work being done on CSS asks its readers to “consider how things have been working recently in JavaScript” before outlining the transpiler/polyfill approach to JavaScript development I have been discussing.[[61]](#endnote-61) “I’m already using the async/await functions in production, and that feature hasn’t been implemented in even a single browser!” enthuses the article’s author, before detailing efforts to make CSS similarly future-oriented.[[62]](#endnote-62) That CSS is attempting to become more like JavaScript shows the degree to which transpilation and polyfilling have transformed JavaScript development. In this final section, I explore more fully the ramifications of this new temporal normal.

As discussed above, while working on ES6, TC39 announced a series of changes to the way JavaScript would be both designed and versioned. Moving from a sequential numbers (1 through 6) to yearly releases (ES2015, ES2016, and ES2017), TC39 decided to change the pace at which new features are standardized. Axel Rauschmayer, one of the foremost authorities on the design and evolution of JavaScript, explains the reasoning behind the switch: “the most recent release of ECMAScript, ES6, is large and was standardized almost 6 years after ES5 (December 2009 vs. June 2015).”[[63]](#endnote-63) He identifies “two main problems with so much time passing between releases”:

* Features that are ready sooner than the release have to wait until the release is finished.
* Features that take long are under pressure to be wrapped up, because postponing them until the next release would mean a long wait. Such features may also delay a release.[[64]](#endnote-64)

The logic is that a smaller release, will allow the features that are currently working to be standardized for use by developers, while unfinished ideas can be pushed into the future without having to wait many years for a release. Though Rauschmayer cites the delay between ES5 and ES6 as an example, the more significant delay was in releasing ES5 itself, occurring 10 years after the release of ES3 in 1999.

Rauschmayer’s point about the problem with JavaScript’s previous history of standardization (with roughly 5-10 years between new versions) suggests even further the degree to which transpilation has changed the temporal approach to JavaScript’s standardization. Instead of waiting several years (for browser makers to catch up their implementations to the new standard), TC39 can design new language features and approve them when they are ready. This shift suggests it is no longer browser makers, but the computer scientists who design the language, that are driving the future of JavaScript. Now that the browser makers are no longer beholden to immediately implementing the standard, there is less pressure to maintain a conservative version of the language.

To accomplish this more rapid and experimental approach, the new process also allows members of TC39 to advance aspects of the language through five stages of proposal (from zero to four) using the version control platform Git.[[65]](#endnote-65) This new approach to standard-making is the direct result of the changing temporal nature of JavaScript. As we saw in the previous section, the always-on, realtime translation provided by a variety of online algorithmic systems is changing both the timescale for translation work and, arguably, the temporal scales of the social itself. TC39’s move to Git[^git] also exemplifies this temporal shift. Beyond formalizing the steps for each stage, these new processes distribute the work of the committee in time and space. Rather than wait for infrequent meetings to discuss the work of evolving JavaScript, Git provides a series of tools for copying code to team members, tracking each member’s local changes, and handling any conflicts that may result when different versions of the same file are placed in the shared repository by individual developers. By using these tools as a platform for managing a standard, TC39 has made informal and asynchronous the often heavily formalized and scripted process of standardization itself.

This new development model directly results from the use of transpilation and polyfilling in JavaScript development. Features proposed for future versions of the ECMAScript standard have been implemented in Babel’s “type-0” preset within days of their first being introduced for consideration. Similarly, Mozilla’s documentation of the JavaScript language often includes open source polyfills for new features that developers might want to use as soon as they are approved for the standard. In both cases, the JavaScript developer community that has long struggled against the inertia of its own installed base has leveraged the conversion of human and machine codes, new understandings of translation and the new temporal rhythms implied by these systems to change their entire approach to the creation of the language’s future.

By considering the emergence of this new standardization process and the use of transpilers to translate future versions of JavaScript into the present, we can see how transpiling produces the same temporal distortions as well as the same changes in the social as ubiquitous machine translation. Thus it is possible to conclude that machine translation is moving beyond the translation of human languages. Further, ES6, which is translated from the future into the present, has a number of features Flusser associated with the new theory of translation he saw as a needed explanation of the new digital codes, their apparatuses, and the practices of programming that manipulate both. As Cronin suggests, “the ever-changing, self-renewing figure of translation” will be chief among our symbols of the future, even as the strange new horizons for digital translation trouble our very core assumptions of writing, language, and ourselves.[[66]](#endnote-66)

# Works Cited

“A Short History of JavaScript.” W3C - Web Education Community Group, June 27, 2012. <https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript>.

Benjamin, Walter. “The Task of the Translator.” In *Illuminations: Essays and Reflections*, edited by Hannah Arendt, translated by Harry Zohn, 69–82. New York: Schocken, 1969.

Boast, Robin. *The Machine in the Ghost: Digitality and Its Consequences*. London: Reaktion Books, 2017.

Busch, Lawrence. *Standards: Recipes for Reality*. Cambridge, MA: MIT UP, 2011.

Cronin, Michael. *Translation in the Digital Age*. New York: Routledge, 2012.

Finger, Anke, Rainer Guldin, and Gustavo Bernardo. *Vilém Flusser: An Introduction*. Minneapolis, MN: Minnesota UP, 2011.

Flusser, Vilem. *Writings*. Minneapolis: Minnesota UP, 2004.

Flusser, Vilém. *Does Writing Have a Future?* Translated by Nancy Ann Roth. Minneapolis: Minnesota UP, 2011.

Hu, Tung-Hui. *A Prehistory of the Cloud*. Cambridge, MA: MIT UP, 2015.

Hui, Yuk. *On the Existence of Digital Objects*. Minneapolis, MN: Minnesota UP, 2016.

Jakobs, Kai. “Information Technology Standards, Standards Setting and Standards Research.” Cotswolds, UK, 2003. <http://web.archive.org/web/20131126232302/http://www.stanhopecentre.org/cotswolds/IT-Standardisation_Jakobs.pdf>.

Kittler, Friedrich. *Optical Media*. Translated by Anthony Enns. Cambridge, UK: Polity, 2009.

Krajewski, Markus. *World Projects: Global Information Before World War I*. Translated by Charles Marcrum II. Minneapolis: Minnesota UP, 2015.

Lazzarato, Maurizio. “Immaterial Labor.” In *Radical Thought in Italy: A Potential Politics*, edited by Paolo Virno and Michael Hardt, 1st ed., 132–47. Minneapolis, MN: Minnesota UP, 2006.

MacKenzie, Donald. “Negotiating Arithmetic, Constructing Proof: The Sociology of Mathematics and Information Technology.” *Social Studies of Science* 23, no. 1 (1993): 37–65. <http://www.jstor.org.lib-ezproxy.tamu.edu:2048/stable/285689>.

Moss, Ben. “IE8 Is Back from the Dead.” Webdesigner Depot, January 13, 2016. <http://www.webdesignerdepot.com/2016/01/ie8-is-back-from-the-dead/>.

Pinch, Trevor. “Technology and Institutions: Living in a Material World.” *Theory and Society* 37, no. 5 (July 10, 2008): 461–83. <https://doi.org/10.1007/s11186-008-9069-x>.

“Plugins.” Babel, 2016. <https://babeljs.io/>.

Popa, Bogdan. “36 Percent of Chinese Internet Users Still Running Internet Explorer 8.” softpedia, December 10, 2013. <http://news.softpedia.com/news/36-Percent-of-Chinese-Internet-Users-Still-Running-Internet-Explorer-8-407470.shtml>.

Raley, Rita. “Algorithmic Translations.” *CR: The New Centennial Review* 16, no. 1 (April 1, 2016): 115–37. <http://escholarship.org/uc/item/9p08q4wq>.

Rauschmayer, Axel. “The TC39 Process for ECMAScript Features.” 2ality, November 15, 2015. <http://www.2ality.com/2015/11/tc39-process.html>.

Salter, Anastasia, and John Murray. *Flash: Building the Interactive Web*. Cambridge, MA: MIT UP, 2014.

Sharp, Remy. “What Is a Polyfill?” January 8, 2010. <https://remysharp.com/2010/10/08/what-is-a-polyfill>.

Slaton, Amy, and Janet Abbate. “The Hidden Lives of Standards: Technical Prescriptions and the Transformation of Work in America.” In *In Technologies of Power: Essays in Honor of Thomas Parke Hughes and Agatha Chipley Hughes*, edited by Michael Thad Allen and Gabrielle Hecht, 95–143. Cambridge, MA: MIT UP, 2001. <http://citeseerx.ist.psu.edu/showciting?cid=649498>.

Star, Susan Leigh. “Power, Technology, and the Phenomenology of Conventions: On Being Allergic to Onions.” In *Boundary Objects and Beyond: Working with Leigh Star*, edited by Geoffrey C. Bowker, Stefan Timmermans, Adele E. Clarke, and Ellen Balka, 263–89. Cambridge, MA: MIT UP, 2016.

———. “The Ethnography of Infrastructure.” In *Boundary Objects and Beyond: Working with Leigh Star*, edited by Geoffrey C. Bowker, Stefan Timmermans, Adele E. Clarke, and Ellen Balka, 473–88. Cambridge, MA: MIT UP, 2016.

Sterne, Jonathan. *MP3: The Meaning of a Format*. Durham, NC: Duke UP, 2012.

TC39, Ecma. “The TC39 Process,” 2016. <https://tc39.github.io/process-document/>.

Venners, Bill. “The Good, the Bad, and the DOM.” *Artima Developer*, October 16, 2003. <http://www.artima.com/intv/dom.html>.

Walton, Philip. “Houdini: Maybe the Most Exciting Development in CSS You’ve Never Heard Of.” *Smashing Magazine*, March 24, 2016. <https://www.smashingmagazine.com/2016/03/houdini-maybe-the-most-exciting-development-in-css-youve-never-heard-of/>.

1. JavaScript’s rise in prominence has been meteoric. For instance, a decade ago Macromedia’s Flash was considered the gold standard for adding interactivity to websites, while JavaScript was regarded as a half-baked toy. Advances in JavaScript’s tool ecosystem and the concomitant revelation of serious performance and security concerns with Flash have reversed these fortunes in a compressed time period. Thus, we can conclude that JavaScript is at once slowed by inertia and rapidly evolving. For more on the history of Flash’s rise and fall, see Anastasia Salter and John Murray, *Flash: Building the Interactive Web* (Cambridge, MA: MIT UP, 2014). [↑](#endnote-ref-1)
2. JavaScript, like many modern languages, is *interpreted* rather than *compiled*, so to be completely accurate, the term I use here should be “interpreter.” In compiled languages, the human-readable language is converted to machine instructions all at once. In interpreted languages, machine instructions are produced one-at-a-time, as the program executes. Though this is a *huge* distinction for computer scientists, the difference between interpreter and compiler are moot for this article. As this article deals with compilation in the abstract and an interpreted language, I refer to what should be called the JavaScript “interpreter” as a “compiler” throughout to avoid confusing readers unfamiliar with computer programming. [↑](#endnote-ref-2)
3. JavaScript was originally so named when browser-maker Netscape implemented Sun Microsystem’s Java in its browser, as part of a cross-branding effort to compete with Microsoft. JavaScript was originally intended to be a faster, lighter language to complement and support Sun’s Java in the browser. [↑](#endnote-ref-3)
4. Rita Raley, “Algorithmic Translations,” *CR: The New Centennial Review* 16, no. 1 (April 1, 2016): 122, <http://escholarship.org/uc/item/9p08q4wq>. [↑](#endnote-ref-4)
5. Markus Krajewski, *World Projects: Global Information Before World War I*, trans. Charles Marcrum II (Minneapolis: Minnesota UP, 2015), 229. [↑](#endnote-ref-5)
6. Michael Cronin, *Translation in the Digital Age* (New York: Routledge, 2012), 141. [↑](#endnote-ref-6)
7. Qtd. in Cronin, 131. [↑](#endnote-ref-7)
8. “A Short History of JavaScript,” W3C - Web Education Community Group, June 27, 2012, <https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript>, n.p. [↑](#endnote-ref-8)
9. Susan Leigh Star, “The Ethnography of Infrastructure,” in *Boundary Objects and Beyond: Working with Leigh Star*, ed. Geoffrey C. Bowker et al. (Cambridge, MA: MIT UP, 2016), 478. [↑](#endnote-ref-9)
10. Star, 478. [↑](#endnote-ref-10)
11. See Tung-Hui Hu, *A Prehistory of the Cloud* (Cambridge, MA: MIT UP, 2015). for a more detailed account of why it is that fiber optic cables follow old rail lines in the United States. [↑](#endnote-ref-11)
12. Ben Moss, “IE8 Is Back from the Dead,” Webdesigner Depot, January 13, 2016, <http://www.webdesignerdepot.com/2016/01/ie8-is-back-from-the-dead/>, n.p. [↑](#endnote-ref-12)
13. Bogdan Popa, “36 Percent of Chinese Internet Users Still Running Internet Explorer 8,” softpedia, December 10, 2013, <http://news.softpedia.com/news/36-Percent-of-Chinese-Internet-Users-Still-Running-Internet-Explorer-8-407470.shtml>. [↑](#endnote-ref-13)
14. Remy Sharp, “What Is a Polyfill?” January 8, 2010, <https://remysharp.com/2010/10/08/what-is-a-polyfill>, n.p. [↑](#endnote-ref-14)
15. Friedrich Kittler, *Optical Media*, trans. Anthony Enns (Cambridge, UK: Polity, 2009), 37. [↑](#endnote-ref-15)
16. Kittler, 37. [↑](#endnote-ref-16)
17. Jonathan Sterne, *MP3: The Meaning of a Format* (Durham, NC: Duke UP, 2012), 136. [↑](#endnote-ref-17)
18. Sterne, 136–7. [↑](#endnote-ref-18)
19. Sterne, 131–2, emphasis mine. [↑](#endnote-ref-19)
20. Amy Slaton and Janet Abbate, “The Hidden Lives of Standards: Technical Prescriptions and the Transformation of Work in America,” in *In Technologies of Power: Essays in Honor of Thomas Parke Hughes and Agatha Chipley Hughes*, ed. Michael Thad Allen and Gabrielle Hecht (Cambridge, MA: MIT UP, 2001), 95, <http://citeseerx.ist.psu.edu/showciting?cid=649498>. [↑](#endnote-ref-20)
21. Slaton and Abbate, 95. [↑](#endnote-ref-21)
22. Slaton and Abbate, 96. [↑](#endnote-ref-22)
23. Susan Leigh Star, “Power, Technology, and the Phenomenology of Conventions: On Being Allergic to Onions,” in *Boundary Objects and Beyond: Working with Leigh Star*, ed. Geoffrey C. Bowker et al. (Cambridge, MA: MIT UP, 2016), 277. [↑](#endnote-ref-23)
24. Kittler, *Optical Media*, 37. [↑](#endnote-ref-24)
25. Sterne, *MP3*, 145–46. [↑](#endnote-ref-25)
26. Donald MacKenzie, “Negotiating Arithmetic, Constructing Proof: The Sociology of Mathematics and Information Technology,” *Social Studies of Science* 23, no. 1 (1993): 38, <http://www.jstor.org.lib-ezproxy.tamu.edu:2048/stable/285689>. [↑](#endnote-ref-26)
27. Trevor Pinch, “Technology and Institutions: Living in a Material World,” *Theory and Society* 37, no. 5 (July 10, 2008): 472, <https://doi.org/10.1007/s11186-008-9069-x>. [↑](#endnote-ref-27)
28. Kai Jakobs, “Information Technology Standards, Standards Setting and Standards Research” (Cotswolds Conference on Technology Standards and the Public Interest, Cotswolds, UK, 2003), <http://web.archive.org/web/20131126232302/http://www.stanhopecentre.org/cotswolds/IT-Standardisation_Jakobs.pdf>, n.p. [↑](#endnote-ref-28)
29. Qtd. in Bill Venners, “The Good, the Bad, and the DOM,” *Artima Developer*, October 16, 2003, <http://www.artima.com/intv/dom.html>, n.p. [↑](#endnote-ref-29)
30. Qtd. in Venners, n.p. [↑](#endnote-ref-30)
31. Sterne, *MP3*, 145. [↑](#endnote-ref-31)
32. Sterne, 139, emphasis original. [↑](#endnote-ref-32)
33. Sterne, 139–41. [↑](#endnote-ref-33)
34. Lawrence Busch, *Standards: Recipes for Reality* (Cambridge, MA: MIT UP, 2011), 189. [↑](#endnote-ref-34)
35. Yuk Hui’s *On the Existence of Digital Objects* (2016) treats this question from a philosophical perspective using Martin Heidegger and Gilbert Simondon. Also of interest is work deriving from Maurizio Lazzarato’s “Immaterial Labor” (1997) that attempts to account for cognitive and affective work as labor from a Marxist perspective. See Yuk Hui, *On the Existence of Digital Objects* (Minneapolis, MN: Minnesota UP, 2016). and Maurizio Lazzarato, “Immaterial Labor,” in *Radical Thought in Italy: A Potential Politics*, ed. Paolo Virno and Michael Hardt, 1st ed. (Minneapolis, MN: Minnesota UP, 2006), 132–47. [↑](#endnote-ref-35)
36. “Plugins,” Babel, 2016, <https://babeljs.io/>, n.p. [↑](#endnote-ref-36)
37. Walter Benjamin, “The Task of the Translator,” in *Illuminations: Essays and Reflections*, ed. Hannah Arendt, trans. Harry Zohn (New York: Schocken, 1969), 70. [↑](#endnote-ref-37)
38. Benjamin, 69. [↑](#endnote-ref-38)
39. Cronin, *Translation in the Digital Age*, 26–33. [↑](#endnote-ref-39)
40. Anke Finger, Rainer Guldin, and Gustavo Bernardo, *Vilém Flusser: An Introduction* (Minneapolis, MN: Minnesota UP, 2011), 45. [↑](#endnote-ref-40)
41. Finger, Guldin, and Bernardo, 49–50. [↑](#endnote-ref-41)
42. Finger, Guldin, and Bernardo, 53. [↑](#endnote-ref-42)
43. Vilem Flusser, *Writings* (Minneapolis: Minnesota UP, 2004), 3. [↑](#endnote-ref-43)
44. Flusser, 3–4. [↑](#endnote-ref-44)
45. Vilém Flusser, *Does Writing Have a Future?*, trans. Nancy Ann Roth (Minneapolis: Minnesota UP, 2011), 17. [↑](#endnote-ref-45)
46. Flusser, 151. [↑](#endnote-ref-46)
47. Flusser, 150. [↑](#endnote-ref-47)
48. Flusser, 150–1. [↑](#endnote-ref-48)
49. Flusser, 155. [↑](#endnote-ref-49)
50. Raley, “Algorithmic Translations,” 119. [↑](#endnote-ref-50)
51. Raley, 133–4. [↑](#endnote-ref-51)
52. Cronin, *Translation in the Digital Age*, 116–22. [↑](#endnote-ref-52)
53. Cronin, 117. [↑](#endnote-ref-53)
54. Flusser, *Does Writing Have a Future?*, 2011, 6. [↑](#endnote-ref-54)
55. Flusser, 55. [↑](#endnote-ref-55)
56. Flusser, 55. [↑](#endnote-ref-56)
57. Raley, “Algorithmic Translations,” 134. [↑](#endnote-ref-57)
58. Cronin, *Translation in the Digital Age*, 131. [↑](#endnote-ref-58)
59. Cronin, 131. [↑](#endnote-ref-59)
60. Robin Boast, *The Machine in the Ghost: Digitality and Its Consequences* (London: Reaktion Books, 2017), 178–82. [↑](#endnote-ref-60)
61. Philip Walton, “Houdini: Maybe the Most Exciting Development in CSS You’ve Never Heard Of,” *Smashing Magazine*, March 24, 2016, <https://www.smashingmagazine.com/2016/03/houdini-maybe-the-most-exciting-development-in-css-youve-never-heard-of/>, n.p. [↑](#endnote-ref-61)
62. Walton, n.p. [↑](#endnote-ref-62)
63. Axel Rauschmayer, “The TC39 Process for ECMAScript Features,” 2ality, November 15, 2015, <http://www.2ality.com/2015/11/tc39-process.html>, n.p. [↑](#endnote-ref-63)
64. Rauschmayer, n.p. [↑](#endnote-ref-64)
65. Ecma TC39, “The TC39 Process,” 2016, <https://tc39.github.io/process-document/>, n.p. [↑](#endnote-ref-65)
66. Cronin, *Translation in the Digital Age*, 141. [↑](#endnote-ref-66)