Philosophy & Technology (2021) 34:13–32
https://doi.org/10.1007/s13347-019-00387-2

RESEARCH ARTICLE

Foo, Bar, Baz...: The Metasyntactic Variable and the Programming Language Hierarchy

Brian Lennon[1]

Received: 8 August 2018 / Accepted: 12 November 2019 / Published online: 18 December 2019

Abstract
This article argues that the English-language nonsense words “foo,” “bar,” “baz,” and others in a more or less standardized sequence of so-called metasyntactic variables commonly used in computer programming ought to be understood as meta-abstractive, re-representing a linguistically derived code’s abstraction of language and the abstraction of the programming language hierarchy itself, making it legible in a manner that rewards culturally oriented study: for example, of programming as a culture and of cultures of software development or engineering.

Keywords Foo · Bar · Metasyntactic variable · Programming language · Request for Comments (RFC)

1 Introduction: “Anyone Could Say Anything and Nothing Was Official”

On April 1, 2001, Network Working Group RFC (Request for Comments) 3092, authored by Donald Eastlake III of Motorola Inc., Carl-Uno Manros of the Xerox Corporation, and Eric S. Raymond of the Open Source Initiative (OSI), was published with the title “Etymology of ‘Foo’.” “Approximately 212 RFCs so far,” the document’s abstract observed, “contain the terms ‘foo’, ‘bar’, or ‘foobar’ as metasyntactic variables without any proper explanation or definition.” This document rectifies that deficiency (Eastlake et al. 2001, p. 1).

The philological promise of this document’s title might be described as at least partly tongue in cheek, in so far as its occasion was ad hoc and its justification was as revisionist as it was inceptive. As a Request for Comments, “Etymology of ‘Foo’” was added to the technical document series by that name (more properly, by the name Internet Requests for Comments) that had been launched in 1968 along with the first

✉ Brian Lennon
blennon@psu.edu

1 Pennsylvania State University, University Park, PA, USA

US Advanced Research Projects Agency grant supporting the development of a protocol for linking computers across a telecommunications network. In an arrangement that might be considered idiosyncratic today, ARPA's funding supported a loosely coordinated team of researchers composed mostly of graduate students at the Los Angeles and Santa Barbara campuses of the University of California, the University of Utah, and MIT, who conferred with representatives from the RAND Corporation and Bolt, Beranek, and Newman, the high-technology research and services firm headquartered in Cambridge, Massachusetts. The procedural informality of the Network Working Group that began meeting in 1968 reflects the unusual circumstances of the moment, which historians of the early internet have attributed to factors including an early (but typical) focus on hardware, which left the ostensibly less important work on software to students and to the team's ad hoc formation, distributed location, and closure as a research community.[1] By all accounts, the founding student members of the NWG were surprised to find themselves left to their own devices: Stephen D. Crocker would later reflect that "[m]ost of us were graduate students and we expected that a professional crew would show up eventually to take over the problems we were dealing with." When the Bolt, Beranek, and Newman team arrived for their first meeting with the fledgling NWG, they "found themselves talking to a crew of graduate students they hadn't anticipated […], while BBN didn't take over the protocol design process, we kept expecting that an official protocol design team would announce itself" (Reynolds and Postel 1987, p. 2). That never happened: as Andrew L. Russell has put it, "Crocker slowly realized that no such team was going to arrive. Much to his surprise, the team of graduate students and contractors in NWG had uncontested jurisdiction over protocol development for the ARPANET" (Russell 2014, p. 169).

What is interesting about the early history of the NWG is the process by which its improvised expertise was deliberately formalized as informal communication and record-keeping. "[A]fter a particularly delightful meeting in Utah," Crocker recalled in the same memoir,

> it became clear to us that we had better start writing down our discussions. We had accumulated a few notes on the design of DEL[2] and other matters, and we decided to put them together in a set of notes. I remember having great fear that we would offend whomever the official protocol designers were, and I spent a sleepless night composing humble words for our notes. The basic ground rules were that anyone could say anything and that nothing was official. And to emphasize the point, I labeled the notes 'Request for Comments'. (Reynolds and Postel 1987, p. 2–3)

As the story of how the RFCs became "the vehicle for the Network Working Group to publish consensus statements and technical standards for the ARPANET—even though they were specifically intended *not* to become standards" (Russell, 2014: 169) is now fairly well known, I will not review it here, though its less well-documented implications for a genre theory of technical communication might also be noted. Janet Abbate

[1] See Russell (2014), p. 168–169; Braman (2011), p. 296–298; Abbate (2000), p. 73–74.
[2] DEL stood for Decode-Encode Language, according to Crocker the first version of a host protocol language. See Reynolds and Postel (1987), p. 2.

reflected that while the RFCs were initially printed on paper, they were stored in electronic form and accessed via the ARPANET as soon as it was possible to do so and that their role in evolving "formal standards informally" cannot be delinked from their then-peculiar mode of production, distribution, and access, notwithstanding Crocker's statement that he had "never dreamed these notes would [be] distributed through the very medium we were discussing in these notes."[3]

The informality of the process by which RFCs were first published, preserved, and used for reference ensured that the documents retained traces of the deliberative process that would normally be removed from the record, or which the creation of a record would normally suppress altogether. Sandra Braman notes that for some time "[t]here appears to have been no expectation that anyone outside of that community would ever be looking at the documents" (Braman 2011, p. 298)—one of several possible reasons that anger, humor, speculative metacommentary on the deliberative process or its direct narration, and other such social byproducts of group decision-making were retained in early RFCs. Though Brian E. Carpenter and Craig Partridge's claim for RFCs as scholarly publications may rest on an idealized understanding of "open" review and a deliberately enervating contrast with other "scholarly publication systems," they are right to emphasize the effects of a later, conventional formalization of the RFCs as the collection's purpose evolved from exploratory communication to authoritative publication, during the later 1970s (Carpenter and Partridge 2010, p. 31).

RFC 3092 "Etymology of 'Foo'" appeared long after the document series embraced this more conventional, authoritative mode. In its reflectiveness on the past and its stated intention to redress some specific, now less desirable consequences of the in-group informality of that past, RFC 3092 reflects the outward-directed perspective of the RFC as a mature form—that is, as a publication in the conventional sense, directed to some reading public not all of whose members were necessarily known in advance. To a limited yet genuine extent, its purpose, to provide an etymology of the English-language word "foo," can be described as scholarly. And yet in so far as its topic is a philological topic, quite clearly more cultural and historical than technical in nature, RFC 3092 inhabits an eccentric minor subcategory of RFCs, perhaps closer to those in category 1G "Request for Comments Administrative," as catalogued in RFC 1000 "Request for Comments Reference Guide,"[4] than to others now carrying the standardized "informational" designation assigned to many RFCs that earn that descriptor, but also to the parodic RFCs often published on April Fool's Day each year. In this and in other ways that make it neither an essential technical document, nor one clearly segregatable with openly or hyperbolically humorous and "joke" RFCs, "Etymology of 'Foo'" might be understood as representing the substantive, rather than glib or flippant persistence of informality within the formality of the RFC series of today. I want to suggest that the putative topic of "Etymology of 'Foo'," the metasyntactic or deliberately meaningless word given in its title, is as good a marker of this persistence as any, and that in like manner, in their specific contexts in program code in so-called higher-level programming languages, the term "foo" and associated terms are specifically meta-abstractive, abstracting the abstraction of higher-level languages, which is to say rendering their abstraction concrete and tangible.

3 See Abbate (2000), p. 74 and Reynolds and Postel (1987), p. 3.

4 See Reynolds and Postel (1987), p. 7.

My approach here is broadly *philological* in three distinct senses of that term, none of them reducible to its quotidian associations with either historical linguistics or literary historiography. First, it emphasizes the histories of both natural and formal languages, in their individual specificities, over their abstract formal or structural characteristics—if not as more important than the latter, than as equally interesting and deserving of attention. Second, it regards individual natural and formal languages as carriers and sometimes shapers of (specific) cultural histories. Third, it aims to integrate knowledge from different disciplines without rejecting the difference of disciplines, imagining that difference is a mere artifact of bureaucracy, or demanding that that difference be discarded or overcome.

2 Reserved and Unreserved Words

"Approximately 212 RFCs, or about 7% of RFCs issued so far, starting with RFC 269," Eastlake, Manros, and Raymond began,

> contain the terms 'foo', 'bar', or 'foobar' used as a metasyntactic variable without any proper explanation or definition. This may seem trivial, but a number of newcomers, especially if English is not their native language, have had problems in understanding the origin of those terms. (Eastlake et al. 2001, p. 1)

RFC 3092 "Etymology of 'Foo'" concluded with an appendix containing a table of occurrences of the words "foo," "bar," "foobar," and "fubar" in RFCs 269 through 3092 itself. In the document's five intervening pages Eastlake, Manros, and Raymond provided brief definitions of the words "bar," "foo," "foobar," and "foo-fighter," in that order, drawing disproportionately on Raymond's own *The Jargon File*[5] but also on the *Oxford English Dictionary*, the *Free On-Line Dictionary of Computing*, and Ron Goulart's *Encyclopedia of American Comics*, among other sources. Only two of the six definitions given for "foo" defined the word as it had been used in existing RFCs to date. The first of these two, brief and reproduced in its entirety from *The Jargon File*, was "Used very generally as a sample name for absolutely anything, esp. programs and files (esp. scratch files)" (Eastlake et al., 2001, p. 1).

The contemporary English word "foo," for which the *Oxford English Dictionary* provides no definition, appears to date to the 1930s. It can be found in editions of *The Tech*, the student newspaper at the Massachusetts Institute of Technology, as early as 1937, in usage contexts that remain unclear.[6] It may have originated in Bill Holman's comic strip *Smokey Stover,* created in 1935, where it appeared as a nonsense word and in mostly nonsensical catchphrases like "foo fighter," "foomobile," "firefighter of foo," "fire chief of foo," and "where there's foo, there's fire."[7] The migration of "foo fighter" into United States Air Force jargon during the Second World War[8] has encouraged the imagination of a genealogical relationship to other US military slang terms of the same

[5] For the most recent version of *The Jargon File*, see Raymond (2004b).
[6] See, for example, "Mother of Five" (1937), *The Tech* (1938a), and *The Tech* (1938b) — all examples provided in Tim 2011.
[7] See, for example, Holman (1938).
[8] See *Time* (1945).

period, including the acronyms “snafu” (“situation normal, all fouled/fucked up”) and “fubar” (“fouled/fucked up beyond all recognition”).[9] Students and others in M.I.T.’s Tech Model Railroad Club, formed immediately after the war, included it in their club “dictionary” with a jocular definition consistent with its cryptic usage in *The Tech.*[10]

To all appearances, it was not until the early 1960s that the latter usage, which conforms to the first part of Eastlake, Manros, and Raymond’s initial definition (“a sample name for absolutely anything”) was extended into that definition’s second part (“…esp. programs and files”). Today, the only speakers and writers of US English who can be counted on to recognize the word “foo” are academic computer scientists and professional software developers, who will also almost certainly also know much of the “standard list of metasyntactic variables” provided in the second of Eastlake, Manros, and Raymond’s two RFC-specific definitions. “Foo” and “bar,” along with “baz,” “qux,” and “quux,” in that order, followed less strictly by “corge,” “grault,” “garply,” “waldo,” “fred,” “plugh,” “xyzzy,” and “thud,” among others, are used as placeholder names in program code that may be sketched quickly and may be intended to be provisional, demonstrative, illustrative, or otherwise temporary. *As* a specification, the specification of a programming language (that is, its formal definition in a document intended for use as a guide to implementation) is intrinsically constrained: a basic and obvious characteristic that is nonetheless frequently elided by excitable analogies binding code to language, including the concept of a “programming language” itself.[11] As a designed system, a programming language is determined by constraints incommensurable with the arbitrary historical finitude of a natural language. Only its reserved words or identifiers—a small number of syntactic objects given precise semantics—are lexed and parsed, or compiled, usually in the form of blocks of instructions (including functions, classes, and methods): that is, as code to be run. Other text in a program represents data, is ignored (for example, program comments, which are demarcated using a border or barrier symbol), or violates rules of syntax, producing compiler error. Finally, there are identifiers chosen by the program writer as names for classes, functions or methods, and constants or variables. Provided that they are created correctly, according to specified (and rigid) orthographic rules, and that they are not identical to reserved words or identifiers, such names might be words in any natural language whose natural language meaning, as presented by a dictionary, clarifies a program segment's intent, purpose, or structure. A programmer writing a program to compute a financial transaction might, for example, choose the names *price*, *tax*, *shipping*, and *total* for variables assigned the relevant values at stages of the transaction; such a programmer might give a compound name like *calculateTax* or *calculateShipping* to functions or methods that perform subtasks supporting the final result.

[9] On “snafu” and “fubar,” see Elkin (1946), *Oxford Essential Dictionary of the US Military* (2002), Burns and Novick (2007), Blackmore (2009), and *Oxford English Dictionary Online*, (2016).

[10] See Samson (1959).

[11] On the language metaphor in early computing, see Nofre et al. (2014). In practice, programming language specifications can vary widely, and in some cases a standardization document, reference document, or reference implementation serves in place of a specification. For examples representing the most widely used languages today, see Gosling et al. (2019), *ISO/IEC 9899:2018 Information Technology, Programming Languages, C* (2018), *ISO/IEC 14882:2017 Programming Languages, C++* (2017), *Standard ECMA-262: ECMAScript® 2019 Language Specification, 10th Edition* (2019), and *The Python Language Reference* (2019).

It may seem dramatic to suggest that in a domain of such radical constraint, this freedom to choose names becomes a site of struggle, even a kind of suffering, as the programmer is faced with elaborating ad hoc a second, entirely separate, exclusively human-readable task- and program-specific semantics interlayered with that of strictly determined machine-readable syntax, in those portions of the program that run. But if this freedom were not also a curse, it would not be the object of so much legislation in programming language style guides and manuals of software craftspersonship.[12] In practice, such conventions tend to reflect the domain of use, from the single-letter names ("a," "b," "x," "y") common in academic computer science, which reflect the latter's kinship with mathematics, to the very long compound names ("InternalFrameInternalFrameTitlePaneInternalFrameTitlePaneMaximizeButtonWindowNotFocusedState") found in a large enterprise software project.[13] But controversy and conflict over naming conventions can be found even within the most orthodox and best-regulated domains of use.

The choice of "foo," "bar," and so on as names thus has two quite distinct contexts. In one context, it represents a deliberate choice of the asemantic abstraction of a so-called nonsense word—as is common in computer science research and instruction, or in the sample code provided in software library or API (Application Programming Interface) documentation, where a high level of abstraction follows the requirement to provide generalizable examples. Relieved of ready or apparent reference such tokens stand in for domain- or application-specific signifiers whose specification would be premature in such contexts, obscuring the pedagogical generality and availability of the example. In principle, the data that "foo" or "bar" makes available to some later procedure might represent anything from the integer value 1 to the memory address of an intricate data structure.

In another context, however, the choice of "foo" or "bar" represents a deferral of the choice of a semantically (more) meaningful word (for example, "calculateTax," "calculateShipping," or "InternalFrameInternalFrameTitlePaneInternalFrameTitle PaneMaximizeButtonWindowNotFocusedState"), where the latter choice is necessary and required eventually. The latter is the context of professional software development, where the primary emphasis is on tasks and purposes rather than concepts and where both software creation and software maintenance is performed by teams, accounting for the premium placed on code standards including standard styles and other measures of readability. Here, too, "foo" or "bar" holds a place for later specification, but that specification is anything but premature; indeed, it serves a designated domain and design. The only thing standing in its way is the temporality and the sustainability of the programmer's labor in their culmination in a readable, maintainable, tested, and documented internal or external release of code.

In both of these technical usages as a metasyntactic variable in program code, the word "foo" can be found in programming textbooks and other instructional and non-instructional documentation from the mid-1960s onward.[14] "Foo" and "bar" appear both separately and in conjoined form, though invariably in sequence (a characteristic

[12] See, for example, Kernighan (1974), McConnell (1993), and Goodliffe (2007).

[13] The latter name reflects the complexity of graphical user interface (GUI) application components written in a programming language that strictly enforces an object-oriented programming style, such as Java. See, for example, Chartier (2004), Grouchnikov (2007), and Ynda-Hummel (2013).

[14] See, for example, Berkeley and Bobrow (1964), Hart and Levin (1964), and Samson (1966).

encouraging the imagined link to the aforementioned acronym “fubar”). John Everett recalls that when he began working at Digital Equipment Corporation in 1966,

> foobar was already being commonly used as a throw-away file name. [...] Foo and bar were also commonly used as file extensions. Since the text editors of the day operated on an input file and produced an output file, it was common to edit from a .foo file to a .bar file, and back again. It was also common to use foo to fill a buffer when editing. (Everett 1996)

Of the examples of metasyntactic variables provided in *The Jargon File*, most of which date to the 1970s, Raymond noted that “The word *foo* is the canonical example. To avoid confusion, hackers never (well, hardly ever) use ‘foo’ or other words like it as permanent names for anything. In filenames, a common convention is that any filename beginning with a metasyntactic-variable name is a scratch file that may be deleted at any time” (Raymond 2004a).

Raymond catalogued variations on the series, adding notes on their supposed provenance, such as “foo, bar, thud, grunt” (“This series was popular at CMU [Carnegie Mellon University]”); “foo, bar, bletch” (“Waterloo University. We are informed that the CS [Computer Science] club at Waterloo formerly had a sign on its door reading ‘Ye Olde Foo Bar and Grill’”); “foo, bar, fum” (“This series is reported to be common at XEROX PARC [Palo Alto Research Center]”); “foo, bar, baz, bongo” (“Yale, late 1970s”); and “foo, bar, zot” (“Helsinki University of Technology, Finland”). Noting also ostensibly British (“fred, jim, sheila, barney”), Dutch (“aap, noot, mies”), French (“toto, titi, tata, tutu”), Italian (“pippo, pluto, paperino”), New Zealand (“blarg, wibble”) variations, among others, Raymond suggested that “Of all these, only foo and bar are universal” (Raymond 2004a).

3 Name, Variable

In both programming language design and programming practice, the word “name” is used for the human-readable linguistic token chosen by the programmer and stored in an encoded form including a location address in system memory, where data associated with the name will be stored. A programmer chooses the name, but not the memory address. The latter is assigned by the programmer’s software tool called an interpreter, compiler, or assembler, which associates the name with the address in a process or event called *binding*.

“Variable” is one of several possible terms for the composite computational object formed by binding a programmer-chosen, human-readable name to a memory address holding associated data. In many programming languages, almost any association between a name and a memory address of stored data, from a single binary value to an entire software package or namespace, can be described as a binding: here, the distinction that matters is between a binding that can be changed (that is, which is mutable in character) and a binding intended to be constant, which cannot be changed. Used more strictly, “variable” refers to (mutable) bindings to discrete, elementary data objects—usually non-compound objects, often the smallest and least important to the general purpose of the program, explicitly restricted in purpose or lifetime, even outright disposable. The classic example is the iterator variables used in loop constructs, frequently given the single-character names *i*, *j*, *k*, and so on.

```
for (int i = 0; i < 10; i++) {
  printf("%d\n", i);
}
```

Code example 1: C language for loop, using an iterator variable to print the integer sequence from 0 to 9.

Properly speaking, variables are a central feature of only one of the four major categories of programming languages (imperative, functional, logic, and object-oriented). The design of imperative languages—the most widely used of which include Fortran, Cobol, variants of Algol, Pascal, Ada, Perl, and Python, as well as the entire so-called C family (including C, C++, Objective-C, C#, and arguably also Java, JavaScript, PHP, Go, Rust, and Swift), but exclude Lisp, Prolog, Scheme, Haskell, and Clojure—can be understood as an abstraction of the so-called von Neumann architecture, or more generally the stored-program computer design, which integrates a processor with memory storing both data and instructions. Understood as a name and memory address bound in both stateful and mutable relationship, a variable represents an abstraction of a computer memory address or range of cellular addresses at which data can be stored, arranged in the sequential units called bits, nibbles, bytes, and so on.[15] The number of such units used to store data associated with a variable name will vary with memory and other aspects of hardware design as well as specifications provided by a firmware instruction set and a software operating system, among other factors, but the informal use of the term "variable" to denote a primitive data structure is licensed by the association with single units and the bounded memory segments allocated for them—for example, the one- to eight-byte numeric data types char (character), short (short integer), int (integer), long (long integer), float (floating-point number), and double (extended floating-point number) in the C language.

Though the programmer enjoys semantic freedom in choosing a variable name that is limited only by the issue of (human) readability, most languages impose orthographic restrictions that include confinement to alphanumeric characters plus a subset of other "special" characters based on those commonly used for punctuation in English, prohibiting a name from beginning with a numeric character and prohibiting spaces within names (early versions of Fortran are a notable exception), a requirement to which the various "casing" schemes (for example "snake case," in which underscores represent spaces, and "camel case," in which word boundaries are marked by capitalization) are responses.[16] Imposed to facilitate automated parsing of program text, such restrictions have an oblique relationship to the dilated intelligibility required of placeholder words like "thingamajig," "whatchamacallit," and "whatsit," used in other language contexts where semantic precision is unavailable for whatever reason. Quite unlike the speaker of English who resorts to one of these words in a forgetfulness or ignorance momentary or otherwise, a programmer in most languages is free to choose as a variable name a non-pronounceable and utterly meaningless sequence like xxxxxxxxxx or lkjsdflkjsdflk. That the pronounceable semi-semantic words "foo"

[15] See Sebesta (2010), p. 208, 211.
[16] See Sebesta (2010), p. 209-210.

and “bar,” very words in the linguistic sense, are conventionally chosen instead, and that they are preferred to the direct if unspecific reference and indeed, referentiality of “whatsit,” “whatchamacallit,” and the like, leaves us with all sorts of interesting questions.

4 Four Subdomains

Ian Watson has described as “stand-ins” a wide range of linguistic phenomena in which direct but unspecific reference serves a useful or necessary role, such as the English-language use of the proper names “Jane Doe” and “John Doe,” “Joe Sixpack,” “Anytown,” “Podunk,” “XYZ Corp.” and many others, including equivalents in other languages. Stand-ins, Watson argues, operate principally as “exemplifiers,” generalizing an unspecified or unknown individual case or substituting a general case, such as a generalized social identity or role, for an unspecified or unknown individual. Other examples mentioned by Watson include the surname Roe used to refer to the then-unidentified plaintiff (Norma Leah McCorvey) in *Roe v. Wade*; the name Johnny, in the rhetorical question “If Johnny told you to jump off the Empire State Building, would you listen to him?”; the word “widget” as used in economic theory, as a stand-in for “all possible products”; and the range of US telephone numbers beginning with 555, left mostly unassigned and used to create fictional numbers used in film and television[17]; but one could also mention the words “thingy,” “thingamabob,” “thingamajig,” “gizmo,” “gadget,” “John Hancock” (referring to anyone’s signature), “Advent Corporation” (as used in legal documents), “Main Street,” and “Peoria.”

All stand-ins, Watson suggests, are also exemplifiers in this sense; but he identifies a special class of stand-ins (and thus a special class of exemplifiers) that perform an additional function. Calling these “metasyntactic signifiers,” Watson explains that “[i]n computer science, ‘foo’ and ‘bar’ are among the conventionalized stand-ins for ‘any algorithmic variable,’ and my use of the term ‘metasyntactic signifier’ is based on computer scientists’ use of the term ‘metasyntactic variable’ for these stand-ins” (Watson, 2005: 145–146). Metasyntactic signifiers, Watson suggests, are fundamentally algebraic in character, belonging to a set of signifiers alienated from their original or conventional referents and available for redirection to a protean class of other referents specified for the occasion. Importantly, metasyntactic signifiers serve their function only because their alienation is fully conventionalized: “[i]f your name really were John Doe,” Watson observes, “you would have to constantly emphasize to people that that really is your name and not a pseudonym” (Watson, 2005: 148).

Of course, such conventionalization is a historical process, meaning that it develops over time (and can be reversed). It is subject to the geographically determined constraint of regional variation, as well. Writing in *The American Mercury* in October 1924, Louise Pound offered the following sample list of what she called “American indefinite names” catalogued in the midwestern USA: “Thingumbob, thingumabob, thingumajig, thingumajiggen, thingumadoodle, dingus, dingbat, doofunny, doojumfunny, doodad, doodaddle, doogood, dooflickus, dooflicker, doojohn, doojohnny, dooflinkus, doohickey, doobobbus, doobobble, doohinkey, doobiddy,

[17] See Watson (2005), p. 142–152.

doohackey, gadget, whatyoumaycallit, fumadiddle, thinkumthankum, dinktum, jigger, fakus, kadigin, thumadoodle, optriculum, ringumajig, ringumajing, ringumajiggen, boopendaddy, thumadoodle, and dibbie" (Pound, 1924: 236–237). The word "kadigin," to take one example from this list, was given a one-word definition ("thingamajig") in the first and second editions of Harold Wentworth and Stuart Berg Flexner's *Dictionary of American Slang*, published in 1960 and 1975 respectively; but it was dropped from its successor, the *New Dictionary of American Slang* edited by Robert L. Chapman.[18]

By contrast with the Midwestern usage context of Pound's list, "many or most" members of which, she noted, "may be general over the USA," the context in which the metasyntactic variables "foo," "bar," et al. are used is highly constrained: a professional (and to a smaller extent, a parallel amateur) technical culture access to which requires high levels of technical ability and/or education as well as a level of privilege commensurate with it. At the same time, we might want to say that "foo," having long since reached the end of its life as a stand-in (in Watson's sense) or "indefinite name" (in Pound's sense) in conversational or written English, owes its continued life or afterlife to that restricted context, in which it is classified as a "metasyntactic variable."

The terms "metalinguistic formula" and "metalinguistic variable" can be found in technical literature as early as 1959, in John W. Backus's early formulation of what is now called the Backus-Naur Form (BNF) of notation for programming language syntax, initially created for the Algol 60 specification. The following year, Martin Davis and Hilary Putnam used the term "syntactic variable" in a paper titled *A Computing Procedure for Quantification Theory,* while phrasing incorporating the term "metasyntactic" appeared in other reports and documents associated with Algol 60.[19] By the 1970s, when the first versions of *The Jargon File* appeared on systems at Stanford and MIT, "metasyntactic variable" was widely used in descriptions of syntax notation and related formalisms, and no longer automatically associated with Algol alone.[20] From that point forward, the metasyntactic variable sequence *foo*, *bar*, *baz*, *qux…* developed into a characteristic feature of at least four distinct subdomains of the practice of programming.

4.1 Pseudocode

The first of these four subdomains is the writing and reading of pseudocode. Defined descriptively, as an artifact, pseudocode is a generic term for English-language (or other natural language) writing whose syntax resembles the syntax of a particular programming language or family of programming languages but is not identical to it. Importantly, pseudocode is *not* (yet) executable: that is, it is written for human reading only. In this sense, pseudocode is an abstraction of code in a particular programming language or family of languages and even of program code generally. One of its main uses is to specify an algorithm, in any pedagogical, referential, or theoretical context

[18] See Wentworth and Flexner (1960), Wentworth and Flexner (1975), Chapman and Wentworth (1986), and Chapman et al. (1995).

[19] See Backus (1959). For Davis and Putnam's use of "syntactic variable" see Davis and Putnam (1960). The phrase "metasyntactic classes" and "metasyntactic formulae" appear in *ALGOL Bulletin* (1961) and Knuth and Merner (1961), both in discussing the Algol 60 specification. For a catalog of other early examples of these and similar usages, see Bron (2017).

[20] See, for example, Hewitt et al. (1973), Nicholls (1975), and Galley and Pfister (1979).

where the core form and idea of a procedure is usefully distinguished from its implementations.

```
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \,.\,.\, j - 1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$
```

Code example 2: Insertion sort algorithm described in pseudocode. Source: Cormen et al. (2009), p. 18.

Pseudocode can also be understood as the product of an early, preliminary stage in the process of writing a program. This is trivially implicit in the use of pseudocode to specify algorithms, in so far as algorithms have no value apart from their implementation—so that the abstraction of their specification in a textbook is best grasped as practically motivated, meant to encourage promiscuous implementation. In practice, program comments may begin their life as pseudocode, as a programmer sketches out a program, then comments out lines of pseudocode while replacing them with executable syntax. But pseudocode is otherwise quite distinct from fully developed and permanent program comments, whose purpose is not to anticipate executable code but to explain it, by describing the intent of the programmer who wrote the code to another programmer reading the program.[21] (Note the comment "Insert $A[j]$ into the sorted sequence $A[1\,.\,.\,j—1]$." inserted into the pseudocode in Code example 2, which makes this distinction clear.) We might therefore define pseudocode as human-readable text that provides a facsimile of the machine-readability of program code as well as the non-parseable human readability otherwise reserved for program comments, whose purpose is exclusively human communication.

As an extended form of metasyntax, in this sense, pseudocode's abstraction of program code operates outside the global scope of any working program while retaining unspecific reference to *some* program or programming language or to programmability in general. By contrast, the token-based or tokenized abstraction of metasyntactic variables like "foo" and "bar" produces neither compile time nor runtime errors. The use of "foo," "bar," etc. in pseudocode itself, rather than in a program, might then be understood as a trace of this partial extrusion of parsing and interpretation (or parsability and interpretability).

4.2 Documentation

The second subdomain is the writing and reading of program documentation. By the mid-1970s, Jon Sachs had distinguished three forms of documentation used in software engineering: the external documentation written for users of a software application, the operational documentation written for technical service providers serving users, and the

[21] See Sachs (1976).

internal documentation used by programmers themselves.[22] External and operational documentation may or may not be produced by programmers (they may instead be produced by a professional technical writer whose primary task is to write documentation of an application, not to create or maintain it), but internal documentation is produced exclusively by programmers, for programmers.

An example of the latter is the documentation of application programming interfaces or APIs. Strictly speaking, an API is a modular unit of code created by one group of programmers for use by another group of programmers, either as a software library or as a network-accessed service (or both). Its core concept, the exposure of key functionality and the abstraction of all supporting and ancillary operations, is associated with the object-oriented programming paradigm, which makes a fundamental distinction between public and private data (or state) and encourages or enforces the various forms of modularization indexed by the umbrella terms "encapsulation" and "information hiding." Today the term "API" has been generalized by the start-up culture of Silicon Valley, where it might be used, purposefully loosely, to refer to almost any software product or service at all, or, somewhat confusingly, the documentation of any software product or service, especially when documentation is generated by another software tool designed for that purpose.

Regardless of whether it carries the former connotation or the latter, however, API documentation is a form of internal documentation, intended for use by other programmers rather than for the end user of a software product or service, for whom a manual or user's guide would be prepared instead. It would be as unusual to find "foo" and "bar" used in examples in such a manual or user's guide or even in some of what Sachs called operational documentation (depending on its level of formality), as it would be to *not* find "foo" and "bar" used in internal documentation. Such usage can be found, for example, in the API documentation of Instructure's Canvas learning management system, widely used in higher education in the USA (Code example 3); of MediaWiki, the PHP application that supports most of the best-known wiki websites, including Wikipedia, Wiktionary, Wikimedia Commons, and WikiLeaks (Code example 4); of both Plone and Drupal, two other widely used content management systems; of Amazon Web Services, Amazon.com's cloud computing platform and service; and of GitHub, the most widely used web-based hosting service for code and document repositories.[23] (Code example 3 is clearly oriented toward unit testing, which we might consider a separate subdomain in which metasyntactic variables are commonly used, or alternately treat as another form of documentation, used by other programmers only.)

```
{ "name": "test name",
  "file_ids": [1, 2],
  "sub": { "name": "foo", "message": "bar" },
  "flag": true }
```

Code example 3: Use of "foo" and "bar" in Instructure Canvas API documentation. Source: *Canvas*.

[22] See Sachs (1976).

[23] See *Canvas*, *MediaWiki*, *Plone Documentation*, *Drupal API*, *Amazon Web Services,* and *GitHub Developer*.

If you make a request for api.php?....titles=Foo|Bar|Hello and cache the result, then a request for api.php?....titles=Hello|Bar|Hello|Foo will not go through the cache—even though MediaWiki returns the same data!

Code example 4: Use of "foo" and "bar" in MediaWiki API documentation. Source: *MediaWiki*.

4.3 Computer Science and Programming Pedagogy

The metasyntactic variable sequence *foo*, *bar*, *baz*, *qux…* can also be found in documents and other material created for instruction in computer science and programming or software engineering. Though such usage is closely related to the reading and writing of pseudocode, as discussed above, a distinction can be made between pseudocode used for transmitting knowledge from expert to expert, for example in a reference work devoted to algorithms,[24] and pseudocode used in the instruction of novice programmers. It might be argued that the latter usage should be discouraged, in so far as a student's or other novice programmer's encounter with conventional metasyntactic variables lacks the conventionalized abstraction of their reference necessary for Watson's "stand-ins" to function as they do, which a community of experts can take for granted.

"Rant incoming, walk the other way," began an anonymous participant in Reddit's "learnprogramming" forum, devoted to discussion programming instruction and the experience of novices, in 2014. That participant then continued:

> OK, so, every time I look up an example or ask a question for some horrible reason the answer involves "Foo" and "Bar".
> When I first started out learning programming seeing those two words added an enormous amount of confusion and their repeated use was distracting. I didn't understand their significance and was trying to relate them to the coding problem, but that was only at first.
> Afterward, it made my mind run in circles because they don't mean ANYTHING AT ALL. (*Reddit* 2014)

As the final sentence above makes clear, the author of the post "'Foo', 'Bar'—Does this actually help anyone—Ever?" has understood quite well the intended purpose of "foo" and "bar" as metasyntactic variables: that is, to serve as "stand-ins" in Watson's sense. It is just as clear that the author rejects "foo" and "bar" as yet insufficiently conventionalized stand-ins—which have thus not yet become stand-ins in Watson's sense at all. Many responses to the original Reddit poster's reflection effectively redescribed this discrepancy, either by affirming it or rejecting it. A typical example of the latter was submitted by the Reddit user "Stormflux":

> Ok, but Foo and Bar are pretty standard placeholders for anyone who's been programming professionally for any amount of time. They're part of the culture. I don't see why we should change them. Do they really add that much confusion for new people?"

[24] See, for example, Knuth (1997).

A response to this response (which may have been submitted by the original poster) reads:

> Honestly, I think they do mostly because there is nothing familiar about those words. There is nothing in foo or bar that makes one think “Yes, that’s a placeholder” or “I see how that relates to a dummy example.” When you’re just starting out and you see single words like int, double, short, float, or def that have a special meaning, well then the first time you see foo/bar you’re going to think they must have one too.

As the latter participant suggests, a novice programmer might well take such metasyntactic variables—which are similarly compressed and thus similarly epigrammatic, even gnomic—for elements of programming language syntax such as the data type specifiers int (integer) and double (a type of floating-point number) or the function declaration keyword def.

Though such complaints from learners are numerous and frequent, they are almost invariably resisted by more experienced programmers, who can often be found countering with the reasonable argument that “foo” and “bar” make code easier to read than the single-letter tokens “a,” “b,” “x,” “y,” used in mathematics (a usage also common in academic computer science instruction), or the less persuasive argument that “foo” and “bar” are usefully less culturally specific than referential nouns or proper names, to which one might otherwise be tempted to resort.[25]

4.4 Software Craftspersonship

Finally, the use of the metasyntactic variable sequence *foo*, *bar*, *baz*, *qux*..., along with a discussion of its appropriate usage and of the appropriateness of its usage at all, can be found in the numerous popular manuals of software craftspersonship that both preserve and refine the lore and tradecraft of professional software development, from Brian W. Kernighan and P. J. Plauger’s *The Elements of Programming Style*, which dates to the 1970s, to Steve McConnell’s *Code Complete: A Practical Handbook of Software Construction* and more recent works like Pete Goodliffe’s *Code Craft: The Practice of Writing Excellent Code.*[26] Surafel Lemma Abebe et al. have focused the normative drives of such discourse, which can range from issues of code style to the psychology of project management and of programming itself, on the specific issue of names and naming. Among the “lexicon bad smells” (an extension of the common term “code smell,” used to identify programming mistakes) they identify are “[m]eaningless metasyntactic variables [...] used in an identifier. Example: foo, bar” (Abebe et al. 2011, p. 127). I will return to Abebe et al.’s arguments shortly.

5 Meta-abstraction and the Programming Language Hierarchy

At a more general level than we have considered thus far, we might say that in all of these domains of programming, the token-based linguistic abstraction of the

[25] For another relevant discussion in this form, see Copeland (2010). For another example of a complaint, see Reisner (2012).

[26] See Kernighan (1974), McConnell (1993), and Goodliffe (2007), as cited previously.

metasyntactic variable sequence *foo*, *bar*, *baz*, *qux*... is also meta-abstractive, abstracting the abstraction of higher-level programming languages themselves or raising it to another level. What does this mean?

Recall that the phrase “higher-level language,” now widely used to describe the notation systems that emerged in the mid-1950s combining algebraic expressions with English-language keywords, refers to such systems’ design for software abstraction of hardware-dependent numeric and alphanumeric operation codes. That is to say that the hierarchically higher level that such models establish supersedes a preceding abstraction, the hardware-dependent assembly language operation codes that represented binary digital instructions using alphanumeric mnemonics. Purely numeric so-called machine code, representing hardware processes for storing and incrementing or decrementing simple register values (achieved using electrical polarization or other means, but not so far removed from counting with sticks or other tokens), is thus abstracted by the alphanumeric mnemonic abbreviations of an assembly code created by a processor’s manufacturer for convenience in writing machine instructions. Assembly code is in turn abstracted by higher-level languages that provide a programmer with a lexicon of everyday English-language words mapped to ranges or aggregates of instructions in machine code.

If it is to be meaningful to speak of a supersession of higher-level languages themselves, in such a hierarchical schema, we might do well at this point to reject the science fiction of the usual suspects, from fifth-generation projects to generalized artificial intelligence, and admit the enclosure of computation within natural language that programming languages represent and that metasyntactic variables might be said to index.[27] In a way, we might say that in context, the meta-abstraction of the sequence *foo*, *bar*, *baz*, *qux*... makes the abstraction of the programming language hierarchy tangible, lending it a concreteness it cannot possess, but which is useful as a mode or artifact of (self-) observation. “Foo” functions as a non-semantic keyword within a domain of English-language reserved words whose semantics are not those of the English language but of a programming language design. That is to say that it is a token name, belonging to the category of names the choice of which is granted the programmer, such choice being limited not by the minuscule bounds of a programming language’s syntax (its reserved words), but by the notably vast bounds of the English language itself. And at the same time, it is the token name that the programmer chooses in that subcontext in which English-language meaning is not only not required but is unwanted.

“Experienced programmers,” Abebe et al. observe, “choose identifier names carefully, in the attempt to convey information about the role and behavior of the labeled code entity in a concise and expressive way. In fact, during program understanding, the names given to code entities represent one of the major sources of information used by developers” (Abebe et al., 2011: 125). Abebe et al.’s taxonomy of what they call “*lexicon bad smells* in software includes categories of outright programmer error, such as misspelled identifiers or identifiers clearly used in the wrong context (such as a module or package other than the one to which a segment of code belongs). It also includes categories of effects produced by carelessness—inconsistent use of identifers,

[27] The classification of programming languages includes an imagined fourth and fifth generations; see Martin (1982).

unusual grammatical structure, and redundancy in integrating data type indicators into names when a language requires them to be declared explicitly anyway—or by haste, such as extreme contraction (e.g., using single-letter identifiers). Deeper problems are indicated by the overloading of identifers or other forms of synonymy, which conflate multiple roles or functions that would be better separated, or by elided hyponymy and hypernymy in class hierarchies, in an object-oriented design.

About the use of metasyntactic variables, Abebe et al. have little to say beyond categorizing them as "meaningless terms" used hastily or carelessly. The lack of sustained attention suggests that in their view, the odor of *foo*, *bar*, *baz*, *qux*... is relatively mild. Though the meta-abstraction of the metasyntactic variable sequence is left unexplored, it is arguably key to the nature of the distinction between the activity of writing both readable and executable program code, on one hand, and activities that involve reading that code with human understanding, on the other—a distinction at the heart of Abebe et al.'s study of how one of these domains impacts the other. *Foo*, *bar*, *baz*, *qux*... are "meaningless" in both the domain of natural language and the domain of programming language syntax, and their use as names chosen freely by the programmer reinscribes and reminds us of the difference between natural language and code, just as does any other freely chosen name. If their conventionalized use as metasyntactic signifiers tends to attenuate that difference, for example in encouraging novice programmers to mistake them for reserved words, it is also true that, as we have seen, "foo" has its own specific and determined, if not unambiguous or especially well-documented natural linguistic history and derivation. To echo the title of RFC 3092, we have an etymology of "foo."

6 Cultural Pointers

Another way to understand the cultural function of the metasyntactic variable sequence *foo*, *bar*, *baz*, *qux*... is by analogy to the technical function of the pointer. In programming language design, a pointer is an object storing a memory address. Where data is linked to variable names, a pointer variable is used to point to a memory location other than the location that would otherwise be linked to a name, when assigning data representing primitive values, when linking a name to a data structure distributed in memory, a variable may be a pointer variable by default. (While it was by no means the first to formalize the pointer as a feature, the C language is of all the most widely used programming languages the most strongly identified with it, very much for better and for worse.) Many forms of data processing require that data not be changed in or by the process of evaluating it, if its results are to be relied on. While such a scenario might be imagined unusual, the truth is the opposite: in that the stored-program paradigm expressed by all general-purpose computing architectures today, such operations are intrinsic and extensible even to program code itself, treated as just one more aggregation of data, and early computers routinely modified program code while running a program, without any of the many safeguards later developed to reduce its inherent risk. Pointer variables, understood as aliases, preserve something of this volatility in their abstraction of the memory addressing expressed by variable assignments: a virtualization that in setting the reference of a token free offers the useful illusion of direct memory access.

Pointers provide efficient access to memory contents by multiplying the paths that can be followed to access them. Risky yet handy in the small-scale, improvisational systems programming for which the venerable C language, for example, was designed (that is, to implement the Unix operating system); they are also widely used in large systems written mostly or entirely in C++, C's successor. In addition to being difficult to use properly, they are easy to exploit to gain access to areas of memory storage that need protection, such as the areas used by an OS and the startup programs that load it. For this reason, pointers are features of relatively few languages in use today (in addition to C, this includes Objective-C, C#, Go, and Swift, as well as C++). By contrast with the general concept of *reference*, meaning any controlled and authorized path to extant data used instead of a new object and allocation of memory, pointers live up their name, providing access to a "point" in memory—perhaps especially "pointed," even "pointy" access—whether that point is safe to access or not. A pointer is purely functional in this sense, ordinary or extraordinary depending on context, like an exterior door on an upper floor of a house.

If not in their alienation from what original natural-linguistic referents they once had, then in the extensible alienability in which they remain available for redirection, the metasyntactic variables *foo*, *bar*, *baz*, *qux*… function as *cultural* pointers: specifically introspective instances of a more general concept of reference that governs the contact (or no contact) zone in which natural languages and programming languages appear to touch. In the incomplete but realizable conventionalization that encourages novice learners to confuse them with reserved words, they are not "meaningless," as Abebe et al. would have it, but meta-abstractive—not unlike Watson's "self-destructive" stand-ins, whose conventionalization is a second-order, rather than a primary process. "Designers who create stand-ins and pseudonyms meant to be convincing," Watson observes,

> can, if they wish, build a 'self-destruct' feature into this convincingness, so that it becomes obvious after a little reflection that the stand-in does not actually exist. For example, any American road atlas will confirm that there is no interstate highway 13 in the United States (perhaps for the same reasons that many tall buildings in America have no 13th floor). (Watson 2005)

Where the absence of a road or a building floor numbered 13 represents the modification of a planned technical material system to accommodate something potentially resistant or adversative (here, the ethnographic concept of superstition) in its culture of use, the residual lexicality of the metasyntactic variable is a cultural space in the planned technical system of a code. Retaining both semantic history and semantic availability, as the always already dereferenced sequences xxxxxxxxxx or lkjsdflkjsdflk cannot do, though they may with as much permission be chosen as names, *foo, bar, baz, qux*… point simultaneously or bicamerally to human language, as a technical system and a culture, and to the ordinally mapped virtual space of stored data on which pointer variables operate in redirection. And whereas when dealing with code, the term "metasyntactic" serves to separate such tokens from the specified, thus valid syntax of a programming language, when dealing with language, they are better understood as re-representing a linguistically derived code's abstraction of language. Watson's two examples do not actually illustrate the same thing: any building more than twelve stories high *has* a numeric thirteenth floor, regardless of whether that floor is labeled as such, whereas we certainly can say that no US Interstate Highway System 13 exists (although one was proposed by North Carolina state senator Robert Lee Humber in 1964). It is perhaps less useful to schematize the difference, here, than to take it as demonstrating the volatility of exemplarity itself, in this context.

Consider a similar artifact of the culture of programming, found just as often in software documentation and instructional materials, if not in pseudocode. The phrase "Hello world" refers to a demonstration program that does nothing beyond outputting the English-language phrase "Hello world" (or its common variations: "Hello, world," "Hello world!," etc.). A "Hello world" program is what we might call purposively purposeless, in that its only use is to serve as a minimal demonstration of programming language syntax or a minimally executable program. As such, it shares with the metasyntactic variable that "meaninglessness," in Abebe et al.'s sense, that we have suggested is better understood as a higher-order or meta-abstraction. Call it "meta-procedural," in this case, since unlike the metasyntactic variable, which performs its technical function by remaining asyntactic, a "Hello world" program has no value if it cannot be successfully compiled or directly interpreted: that is, if it fails to output the string "Hello world."

"Hello world" is also very often the first program written by a novice. This gives it a second characteristic that distinguishes its function quite decisively from that of the metasyntactic variable sequence and its presumption of expertise even neatly inverts it. For the string "Hello world" functions as a greeting—a greeting produced by an agent who is deliberately ambiguous or confused. The text "Hello world" is written by the computer, so to speak (more precisely but not quite as exactly, by the compiler or interpreter of the program), as a greeting to the world, as if the computer had just been born or had just acquired the ability to write. But in its pedagogical function as a novice programmer's first program, "Hello world" also speaks for that novice programmer, acquiring and exercising the newfound power to (apparently) directly control a machine. In linguistic terms, the English-language phrase "Hello world" is phatic or performative, establishing a social relation rather than communicating a message, whereas in technical terms, the "Hello world" program produces only a so-called side effect—both literally, in its direction of output to an output device, and figuratively (or generalizably) in what I have called its purposive purposelessness. Grasped in a still broader context, the context established by sociolinguistic theory, "Hello world" is dialogic, meaning not so much the turn-taking of two communicators focused on transmitting and receiving a message, as the polyphony of two speakers speaking at once, and saying the same thing, to an interlocutor who is in each case one's self, the other and an implied third party ("the world") at the same time.

```
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Code example 5: "Hello world" program in C.

If "Hello world" thus enacts a kind of primal scene of computing, in the novice programmer's embrace of power over an electromechanical servant, even a prosthetic slave, the metasyntactic variable "foo" stands in some way for that power's conventionalization and for the programmer's habituation to it. "Foo," we might say, marks the authority, even the authoritarianism of computing in a manner homologous with "snafu," the widely used and fully conventionalized token of specifically US military slang from which some have imagined the former derived. The function of "snafu," Frederick Elkin has written, is to *compliantly* caricature military authority: it is used by the US soldier in a resigned acceptance of both his requirement to obey *and* the disarray of the authority that requires his obedience. This "caricaturing of both the Army and himself," Elkin concludes, "evidences an adjustment in which the soldier accepts his subordinate position in his own mind but does not completely adopt the subordinate role" (Elkin 1946, p. 422).

This may seem to take us far afield from RFC 3092 and its proposed etymology. But if we accept, as I am suggesting that we accept, that the etymology of "foo" leads us not only deep into the history of programming but back out, again, so to speak, into its broadest possible, indeed meta-abstractive context, then we may have to conclude that much about the power to "speak code," as that power is promoted and pursued today, still demands closer examination.

References

Abbate, J. (2000). *Inventing the Internet*. Cambridge, MA: MIT Press.

Abebe, S. L., Haiduc, S., Tonella P., et al. (2011). The Effect of Lexicon Bad Smells on concept location in source code. In *IEEE 11th International Working Conference on Source Code Analysis and Manipulation,* 125–134.

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *In Proceedings of the International Conference on Information Processing, UNESCO* (pp. 125–132).

Berkeley, E. C. and Bobrow, D.G., eds. (1964). *The programming language LISP: its operation and applications*. Information International.

Blackmore, D. S. T. (2009). Fubar. *The seafaring dictionary: terms, idioms and legends of the past and present*. Jefferson, N.C: McFarland & Co.

Braman, S. (2011). The framing years: policy fundamentals in the Internet design process, 1969–1979. *The Information Society, 27*(5), 295–310.

Bron, D. (2017). *What is the history of the term "metasyntactic variable"? Stack Exchange*: *English Language and Usage*.

Burns, K., & Novick, L. (2007). *FUBAR*. *The War*: Public Broadcasting Service.

Canvas. Welcome to the Canvas LMS API Documentation.

Carpenter, B. E., & Partridge, C. (2010). Internet requests for comments (RFCs) as scholarly publications. *ACM SIGCOMM Computer Communication Review, 40*(1), 31–33.

Chapman, R. L., & Wentworth, H. (1986). *New dictionary of American slang*. New York: Harper & Row.

Chapman, R. L., Kipfer, B. A., & Wentworth, H. (1995). *Dictionary of American slang* (3rd ed.). New York: HarperCollins.

Chartier, R. (2004). Longest & shortest types in .NET. *Contemplation*.

Copeland, J. (2010). *Stop using Foo Bar! Ruby Forum*: *Ruby on Rails*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., et al. (2009). *Introduction to algorithms* (3rd ed.). Cambridge, MA: MIT Press.

Davis, M., & Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM, 7*(3), 201–215.

Eastlake, D., Manros, C. and Raymond, E. (2001). *Etymology of "Foo."* Request for Comments (RFC) 3092.
Elkin, F. (1946). The soldier's language. *American Journal of Sociology, 51*(5), 414–422.
Everett, J. (1996). Foobar. *alt.folklore.computers*.
Galley, S. W., & Pfister, G. (1979). *The MDL programming language*. Massachusetts Institute of Technology: Laboratory for Computer Science.
Goodliffe, P. (2007). *Code Craft: The practice of writing excellent code*. San Francisco: No Starch Press.
Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., 2019. *The Java® Language Specification: Java SE 12 Edition.*
Grouchnikov, K. (2007). And the longest JRE class name is…. *Pushing Pixels*.
Hart, T. P. and Levin, M.I. (1964). LISP Exercises.
Hewitt, C., Bishop, P. and Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 235–245.
Holman, B. (1938). *Smokey Stover and the Fire Chief of Foo*. Racine, WI: Whitman Publishing Co..
ISO/IEC 9899:2018 Information technology — Programming Languages — C. 2018.
ISO/IEC 14882:2017 Programming languages — C++. 2017.
Kernighan, B. W. (1974). *The elements of programming style* (First ed.). New York: McGraw-Hill.
Knuth, D. E. (1997). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed.). Reading, MA: Addison-Wesley.
Knuth, D. E., & Merner, J. N. (1961). Algol 60 Confidential. *Communications of the ACM, 4*(6), 268–272.
Martin, J. (1982). *Application development without programmers*. Englewood Cliffs, N.J: Prentice-Hall.
McConnell, S. (1993). *Code complete: a practical handbook of software construction*. Redmond, WA: Microsoft Press.
MediaWiki. MediaWiki API.
Mother of Five. (1937). *Letter to the Editor*. Massachusetts Institute of Technology: *The Tech*.
Nicholls, J. E. (1975). *The structure and design of programming languages*. Reading, MA: Addison-Wesley.
Nofre, D., Priestley, M., & Alberts, G. (2014). When technology became language: the origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture, 55*(1), 40–75.
Oxford English Dictionary Online. (2016). *Fubar, adj*. Oxford University Press.
Oxford Essential Dictionary of the U.S. Military. (2002). *Fubar*. Oxford University Press.
Pound, L. (1924). Notes on the vernacular. *The American Mercury*, 233–237.
The Python Language Reference. 2019.
Raymond, E. S. (2004a). Metasyntactic variable. *The Jargon File*.
Raymond, E. S. (2004b). *The Jargon File, version 4.4.8.*
Reddit (2014). "Foo", "Bar" — Does this actually help anyone — Ever?
Reisner, A. (2012). Eff You Foo Bar. *Alex Reisner.*
Reynolds, J., & Postel, J. (1987). *The Request for Comments Reference Guide*. *Request for Comments (RFC), 1000*.
Russell, A. L. (2014). *Open standards and the digital age: history, ideology, and networks*. Cambridge: Cambridge University Press.
Sachs, J. (1976). Some comments on comments. *ACM SIGDOC Asterisk Journal of Computer Documentation, 3*(7), 7–14.
Samson, P. (1966). PDP-6 LISP.
Samson, P. R. (1959). Foo. *Tech Model Railroad Club Dictionary.*
Sebesta, R. W. (2010). *Concepts of programming languages* (9th ed.). Boston: Addison-Wesley.
Standard ECMA-262: ECMAScript® 2019 Language Specification, 10th Edition, 2019.
The Tech (1938a). On Foo-ism. Massachusetts Institute of Technology.
The Tech (1938b). Temperance Poll Shows 87 Percent of Voters Imbibe. Massachusetts Institute of Technology.
Tim. (2011). *What does "foo" mean? Stack Exchange*: *English Language and Usage*.
Time (1945). Science: Foo-Fighter. *Time* 45(3), p. 72.
Watson, I. (2005). *Cognitive design: creating the sets of categories and labels that structure our shared experience*. Department of Sociology: Rutgers University.
Wentworth, H., & Flexner, S. B. (Eds.). (1960). *Dictionary of American slang*. New York: Thomas Y. Crowell.
Wentworth, H., & Flexner, S. B. (1975). *Dictionary of American slang. 2d* (supplemented ed.). New York: Thomas Y. Crowell.
Ynda-Hummel, I. (2013). What is the longest method name you've ever seen? *Quora.*