What Is Copyrightable in Software?

*Charles Duan**

Abstract

For five decades, copyright law has been internally inconsistent. Computer code receives copyright protection—the statute says so. But the statute also excludes methods of operation from copyright. Given that computer code is a method of operating a computer, these two blackletter doctrines contradict each other. To avoid this doctrinal conflict, courts and commentators have for years controversially disregarded or at least bent one of the doctrines. The Supreme Court has twice ducked the issue as too difficult to resolve, most recently in the 2021 Google v. Oracle *case, and the circuits continue to be split.*

Yet there is a simple solution. Computer science concepts, so far not explored in the legal literature, show that software contains numerous elements with no effect on a computer's operation: comments, syntactic alternatives, bound variable names, and ordering of variable declarations and subroutines. These elements instead serve a purely communicative purpose, helping other humans understand computer code. Computer scientists have long valued these communicative elements as vehicles for expression.

Communicative elements in code reconcile the doctrines, fully satisfying the exclusion of methods of operation while nevertheless providing a basis for copyright protection in software. And they are the strongest basis for copyright in computer code. Protection based on these elements is consistent with copyright rationales and doctrines. It offers a solid foundation for answering the trickiest questions in software copyright law, including application programming interfaces and generative artificial intelligence. And it would resolve, after half a century, the internal contradiction in the Copyright Act.

---

* © 2026 Charles Duan. Assistant Professor, American University Washington College of Law. J.D., Harvard Law School, 2007; A.B. in Computer Science, Harvard College, 2004. I would like to thank Clark Asay, Bob Brauneis, Annemarie Bridy, Chris Buccafusco, Mike Carroll, James Grimmelmann, Paul Ohm, Blake Reid, Michael Risch, Pam Samuelson, Josh Sarnoff, John Whealan, and participants at the 2024 Intellectual Property Scholars Conference, 2024 George Washington University Center for Law and Technology Speaker Series, 2023 Junior Intellectual Property Scholars Association Summer Workshop, and 2020 Works-in-Progress Intellectual Property Colloquium for their ideas and comments relating to this Article. Thanks as well to the editors of *The George Washington Law Review* for their thoughtful and meticulous editing and comments. I would also like to acknowledge and remember my friend and colleague, Sherwin Siy, who made several insightful suggestions to me when this Article was in its inception, and whose brilliance and thoughtfulness on copyright law have been an ongoing source of inspiration for me. Views and errors in this Article are my own.

TABLE OF CONTENT

Introduction

It is a technological feat seemingly magical; it may also be illegal. A person types instructions into Microsoft's Copilot program in plain English: "Create the code for a snake game in JavaScript."[1] Within seconds, dense lines of code for the classic computer game appear on the screen. The code executes without error and even contains explanatory documentation.[2] For a human programmer, doing the same might take hours of research and coding, additional time for debugging errors, and, of course, years of computer science training.[3] The computer does it instantly and perfectly.

Code-generative artificial intelligence ("AI") systems like Copilot are the locus of tremendous recent controversy due to their interaction with copyright law.[4] This is, of course, part of the larger debate about whether generative AI infringes copyrights—Copilot produces this humanlike code based on having been trained on millions of lines of code actually written by humans, rendering its outputs, and potentially the system itself, an impermissible copy of that human-written training data.[5] Whether generative AI infringes copyrights in that training data has been a difficult question of law and policy.[6]

---

1 *See* Thomas Dohmke, *Web Summit Rio 2023: Building an App in 18 Minutes with GitHub Copilot X*, GitHub: Blog (May 5, 2023), https://github.blog/developer-skills/github/web-summit-rio-2023-building-an-app-in-18-minutes-with-github-copilot-x/ [https://perma.cc/XS39-DSKQ].

2 *See id.*

3 *See, e.g.*, Luke Garrigan, *How to Code Snake*, DEV Cmty. (May 15, 2021), https://dev.to/lukegarrigan/how-to-code-snake-1jeb [https://perma.cc/7C6Y-5MBH] ("I wouldn't personally recommend this game to beginner programmers as Snake certainly has some tough quirks you have to figure out.").

4 *See, e.g.*, Rina Diane Caballar, *Ownership of AI-Generated Code Hotly Disputed*, IEEE Spectrum (Nov. 19, 2022), https://spectrum.ieee.org/ai-code-generation-ownership [https://perma.cc/4LV3-RA7Q]; Emma Roth, *Microsoft, GitHub, and OpenAI Ask Court to Throw Out AI Copyright Lawsuit*, Verge (Jan. 28, 2023, at 19:02 ET), https://www.theverge.com/2023/1/28/23575919/microsoft-openai-github-dismiss-copilot-ai-copyright-lawsuit [https://perma.cc/AX7X-4VJ9] (quoting copyright plaintiffs' attorneys as alleging "software piracy on an unprecedented scale"); Craig Topham, *Publication of the FSF-Funded White Papers on Questions Around Copilot*, Free Software Found. (Feb. 24, 2022, at 17:36 ET), https://www.fsf.org/news/publication-of-the-fsf-funded-white-papers-on-questions-around-copilot [https://perma.cc/2274-8Q35].

5 *See* Mark Chen et al., Evaluating Large Language Models Trained on Code § 3.1 (July 14, 2021) (unpublished manuscript), https://arxiv.org/pdf/2107.03374 [https://perma.cc/ZF8L-RWZ8] ("Our training dataset was collected in May 2020 from 54 million public software repositories hosted on GitHub, containing 179 GB of unique Python files under 1 MB."); Katherine Lee, A. Feder Cooper & James Grimmelmann, *Talkin' 'Bout AI Generation: Copyright and the Generative-AI Supply Chain*, 72 J. Copyright Soc'y U.S.A. 251, 272–74 (2025).

6 *See, e.g.*, Matthew Sag, *Copyright Safety for Generative AI*, 61 Hou. L. Rev. 295, 326–37 (2023) (describing memorization and the "Snoopy problem," in which AI reproduces copyrighted works); Lee et al., *supra* note 5, at 256; Pamela Samuelson, *Generative AI Meets Copyright*, 381 Science 158, 158–59 (2023).

But as new as the issues with code-generative AI may be, the technology raises another question that, at least on a computer technology timeline, is decidedly old. In order for AI-generated code to infringe copyrights, the code must be copyrightable in the first place, but computer code is not obviously copyrightable.[7] Despite over half a century of jurisprudence on computer software and copyright law, there is no consensus on what, if anything, constitutes protectable subject matter.[8] The appellate courts have long been split on the issue—the Second Circuit has rejected the Third Circuit's test,[9] the First Circuit has rejected the Second Circuit's test,[10] and Federal Circuit judges have quarreled over what precedent to follow as recently as 2023.[11] The issue might seem ripe for Supreme Court review, but twice now, most recently in 2021, the Justices have found the question too difficult to answer.[12] Scholarly commentary has come into no greater alignment.[13]

What explains these decades of discord over this copyright question of obvious jurisprudential and economic importance? The short answer is that the application of copyright law to computer software rests on a trilemma of three incompatible propositions:

1. Computer code is copyrightable, per 17 U.S.C. § 117;
2. Methods of operation are not copyrightable, per 17 U.S.C. § 102(b); and
3. Computer code is a method of operating a computer.

The first two propositions are legal doctrines, both with extensive historical, legislative, jurisprudential, and logical backing.[14] Yet in view of the third, which seems almost tautological of the nature of computer

---

7 *See infra* Section II.C.

8 *See infra* Part II.

9 *See* Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 706 (2d Cir. 1992) (rejecting *Whelan Assocs., Inc. v. Jaslow Dental Lab'y, Inc.*, 797 F.2d 1222 (3d Cir. 1986)).

10 *See* Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 815 (1st Cir. 1995) (rejecting *Computer Associates* as "misleading" and applying a 17 U.S.C. § 102(b)—based test instead), *aff'd by an equally divided Court*, 516 U.S. 233 (1996).

11 *Compare* SAS Inst., Inc. v. World Programming Ltd., 64 F.4th 1319, 1331 (Fed. Cir. 2023) (citing Eng'g Dynamics, Inc. v. Structural Software, Inc., 26 F.3d 1335, 1340 (5th Cir. 1994)) (discussing how *Engineering Dynamics* informs the court of steps that must be completed prior to an infringement analysis), *with SAS Inst.*, 64 F.4th at 1338 (Newman, J., dissenting) (contending that *Engineering Dynamics* is irrelevant to copyrightability). *See generally infra* Section II.B (discussing the case law surrounding copyrightability of code).

12 *See* Google LLC v. Oracle Am., Inc., 593 U.S. 1, 20 (2021) ("We shall assume, but purely for argument's sake, that the entire Sun Java API falls within the definition of that which can be copyrighted."); *Lotus*, 516 U.S. at 233.

13 *See infra* Section II.D.

14 *See infra* Section I.B–.C.

programs,[15] one of the doctrines seemingly must be false.[16] Those favoring stronger protection for computer programs thus conclude that Congress must have intended to abrogate or limit the copyright prohibition for methods of operation, while those opposed conclude that software copyrights must be evanescently thin.[17]

Yet it is possible for both of these well-supported copyright doctrines to coexist—computer code can be copyrightable even if methods of operation are not. The error lies in the third proposition. Computer code certainly is a method of operating a computer, but it also contains plenty of material having nothing to do with computer operation and everything to do with the kind of expression that copyright is all about.[18]

By joining the literature on software engineering and on copyright law, this Article answers a decades-old question and determines exactly what is copyrightable in software. The computer science literature shows that code contains numerous "communicative elements": written aspects that enable the programmer to express to other *human* readers what the program does and how it does it.[19] These communicative elements do not change the operation of the computer; programming languages often delineate communicative elements as irrelevant to how the program behaves.[20] Comments interspersed among operating code, spacing and line breaks, ordering of blocks of code, and names of certain variables are all examples of these communicative elements that help humans read computer code but have no effect on operation.[21]

---

15 *See infra* Section I.A.

16 *See* Oracle Am., Inc. v. Google Inc. (*Oracle II*), 750 F.3d 1339, 1367 (Fed. Cir. 2014) ("If we were to accept the district court's suggestion that a computer program is uncopyrightable simply because it 'carr[ies] out pre-assigned functions,' no computer program is protectable. That result contradicts Congress's express intent to provide copyright protection to computer programs . . . ." (alteration in original) (quoting Oracle Am., Inc. v. Google Inc., 872 F. Supp. 2d 974, 1000 (N.D. Cal. 2012))); Lloyd L. Weinreb, *Copyright for Functional Expression*, 111 HARV. L. REV. 1149, 1203 (1998) ("Copyright doctrine contains no rule that can be applied as a ground for decision in [computer software] cases without contradiction by another rule that stands on an equal footing."); Mark A. Lemley, *Convergence in the Law of Software Copyright?*, 10 HIGH TECH. L.J. 1, 25 (1995) ("The rules of section 102(b)—against copyrighting ideas, processes, systems, etc.—have not been strictly enforced in the software context . . . ."); OFF. OF TECH. ASSESSMENT, U.S. CONG., OTA-TCT-527, FINDING A BALANCE: COMPUTER SOFTWARE, INTELLECTUAL PROPERTY, AND THE CHALLENGE OF TECHNOLOGICAL CHANGE 29 (1992), https://ota.fas.org/reports/9215.pdf [https://perma.cc/CAK6-7PSX] ("The functional aspects of computer programs pose difficult questions for application of the copyright law . . . .").

17 *See infra* Section II.D.

18 *See infra* Part III.

19 *See infra* text accompanying notes 245–50.

20 *See, e.g.*, BRIAN W. KERNIGHAN & DENNIS M. RITCHIE, THE C PROGRAMMING LANGUAGE 13 (2d ed. 1988), https://archive.org/details/cprogramminglang00bria [https://perma.cc/3GER-TCCB] ("C compilers do not care about how a program looks . . . .").

21 *See infra* text accompanying notes 256–97.

Communicative elements of computer code fit hand in glove with copyright law's policies and doctrines. Choices of how to use communicative elements of code are often closely tied to an individual programmer's or coding house's style and are thereby a representation of originality and authorial identity.[22] Programmers exhibit creativity through their choices of communicative elements: Judiciously chosen spacing and variable names, for example, can make a program look like a clean pedagogical tool, an inscrutable exhibit of hacker prowess, or even a work of visual art.[23] Because they do not affect computer operation, communicative elements do not trigger the functionality or method-of-operation bars to copyright protection.[24] Premising software copyright protection on the communicative elements of code best reconciles the theory of copyright law and the practice of computer programming.

This exposition substantially advances the literature on copyright law and computer software. Up to now, courts and commentators have largely approached the doctrinal trilemma above with one of four approaches: (1) *abrogation*, in which it is assumed that § 102(b) does not mean what it says it means;[25] (2) *creative functionality*, in which any creative choices are copyrightable regardless of whether they are also operational;[26] (3) *unidentified expression*, in which software contains some copyrightable element that cannot be articulated;[27] and (4) *infinitesimality*, in which software copyright protection is so thin that it protects against little beyond literal copying.[28] The communicative elements theory proposed by this Article resolves the limitations of these previous approaches and offers courts and jurists a stronger framework for identifying the copyrightable elements of software.[29]

The theory has a practical payoff too, answering several difficult questions of contemporary copyright law. It shows, with respect to the *Google LLC v. Oracle America, Inc.*[30] case, that *there was copyrightable matter in the case*—but not the matter the litigants identified.[31] Additionally, the theory adds an important dimension to the copyright questions surrounding code-generative AI,[32] identifying surprising discrepancies between the litigants' arguments and the actual copyrightable subject

22 *See infra* Section III.G.

23 *See infra* text accompanying notes 349–54.

24 *See infra* text accompanying notes 245–50.

25 *See infra* text accompanying notes 121–23.

26 *See infra* text accompanying notes 125–26.

27 *See infra* text accompanying notes 127–28.

28 *See infra* text accompanying notes 129–30.

29 *See infra* text accompanying notes 355–59.

30 593 U.S. 1 (2021).

31 *See infra* Section IV.A.

32 *See infra* Section IV.B.

matter in code[33] while revealing deeper truths about copyright and creativity.[34]

This Article proceeds as follows. Part I provides background on the three prongs of the trilemma: the nature of computer programs, unprotectability of methods of operation, and statutory recognition of software copyrights. Part II reviews the history of software copyright law with a focus on how commentators and courts have grappled with the three prongs. Part III proposes the communicative elements test, enumerates several types of communicative elements, and discusses how programmers use those elements. Finally, Part IV applies the proposed communicative elements theory, both generally and to two cutting-edge problems of software copyright law: (1) protectability of application programming interfaces, and (2) code-generative AI.

I. Prongs of a Trilemma

The motivating problem for this Article arises out of a seeming contradiction in the Copyright Act of 1976 ("1976 Act"),[35] which appears to both require and condemn copyright in computer code. This Part expands upon the three premises of that contradiction. First, this Part provides a brief description of the nature of computer code and computer programming.[36] Next, it considers the long-established premise of the unprotectability of methods of operation, which has been entrenched in copyright law well before the advent of software.[37] Finally, it reviews the other doctrinal prong, statutory recognition of computer code as copyrightable subject matter.[38]

A. How Computer Code Works

Virtually every substantial analysis of software copyrights contains an explanation of what a computer program is;[39] this Article is no

---

33 *See* First Amended Complaint at 16, Doe v. Github, Inc., 672 F. Supp. 3d 837 (N.D. Cal. 2024) (Nos. 4:22-cv-06823, 4:22-cv-07074), Dkt. No. 98; *see also id.* at 26 ("These differences in the code are cosmetic and the code is functionally equivalent; otherwise, this is a verbatim copy.").

34 *See infra* text accompanying notes 376–77.

35 17 U.S.C. §§ 101–1332.

36 *See infra* Section I.A.

37 *See infra* Section I.B.

38 *See infra* Section I.C.

39 *See* Michael Risch, Google v. Oracle *and the Search for an Analogy*, Written Description (Oct. 12, 2020), https://writtendescription.blogspot.com/2020/10/google-v-oracle-and-search-for-analogy.html [https://perma.cc/FS3C-VW3P] (discussing various attempts at explaining components of software by analogy); *see, e.g.*, Stephen Breyer, *The Uneasy Case for Copyright: A Study of Copyright in Books, Photocopies, and Computer Programs*, 84 Harv. L. Rev. 281, 341–43 (1970); Alfred Z. Spector, *Software, Interface, and Implementation*, 30 Jurimetrics J. 79, 80 (1989).

different. Even in this *scène à faire*[40] of copyright scholarship, however, there is a notable gap to be addressed. The traditional explanation of computer programs is top-down, starting with the programmer's initial planning stages and proceeding through the outlining of flowcharts, writing of code in a high-level programming language, and conversion to machine-executable "object code."[41] Proceeding in this order is natural, but it ends up emphasizing the first few steps and leaving the last ones, most importantly, the nature of object code, unexplained.

As such, the following explanation will take a bottom-up approach, starting from what the *computer* does and working toward how one instructs the computer at higher levels of abstraction. Besides offering an alternative framing to the traditional literature, this bottom-up approach helps to define more sharply the methods of operating a computer.

To understand computer programs, it is first necessary to understand computers. Every computer, even the most complex, boils down to two main components: a *memory* and a *processor*.[42] The memory is a list or table where numbers can be stored and retrieved by location, akin to a spreadsheet or a series of addressed mail slots.[43] Each position in the memory has a numerical *address*—analogous to the spreadsheet cell number, or the mail slot number—and can retain a single number as a value, analogous to the contents of the spreadsheet cell.[44]

A processor, often called a central processing unit or CPU, is an electronic circuit device that reads a list of instructions and executes

---

40 A *scène à faire* is a stock element of a genre of literature. *See, e.g.*, Leslie A. Kurtz, *Copyright: The Scenes a Faire Doctrine*, 41 FLA. L. REV. 79, 81 (1989).

41 *See, e.g.*, Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 697–98 (2d Cir. 1992) (citing Steven R. Englund, Note, *Idea, Process, or Protected Expression?: Determining the Scope of Copyright Protection of the Structure of Computer Programs*, 88 MICH. L. REV. 866, 867–73, 868 n.13 (1990)); *cf.* Anthony L. Clapes, Patrick Lynch & Mark R. Steinberg, *Silicon Epics and Binary Bards: Determining the Proper Scope of Copyright Protection for Computer Programs*, 34 UCLA L. REV. 1493, 1512–23 (1987) (intertwining high-level and low-level aspects of computer programming).

42 This is the Von Neumann architecture used in most modern computers. *See* SUZANNE J. MATTHEWS, TIA NEWHALL & KEVIN C. WEBB, DIVE INTO SYSTEMS: A GENTLE INTRODUCTION TO COMPUTER SYSTEMS 219–20 (2022), https://diveintosystems.org/singlepage [https://perma.cc/9YBB-8AL3]. Although any textbook on computer architecture would support the citations in this section, this Article relies on Matthews and colleagues' text because their book is freely available and written at an ideal level of detail for those interested in the topic. The website links provided direct to the online version of the textbook, but all pinpoint citations reference the page numbers in the print version for precision.

43 *See id.* at 221. Most computers have several types of memory devices, in particular random-access memory, caches, and processor registers. *See id.* The distinction is irrelevant for this discussion.

44 How do computers store non-numerical information like words, pictures, or music? By converting them to lists of numbers—for example, one number for each pixel, pixel color, or sound wave position. *See id.* at 171–74 ("Any information can be encoded in binary, including rich data like graphics and audio.").

them.[45] Generally, instructions can direct the processor to: (1) read or write numbers at a location in memory, (2) perform a mathematical operation on one or more numbers, (3) read numbers from an input device like a keyboard, or write numbers to an output device like a screen, and (4) jump to an out-of-order instruction if a condition is met—for example, skip forward ten instructions if the value of memory slot 15 is zero.[46]

Instructions, being stored in the memory, must also be numbers.[47] Thus, instructions are assigned numerical operation codes, or *opcodes*.[48] In one processor architecture, opcode 128 is the instruction to add two numbers together in memory.[49] An instruction may be followed by one or more *arguments*, numerical input values that affect how the processor performs the given instruction.[50] Opcode 128, for example, is followed by a memory address where the sum of the two added numbers should be stored, and then two memory addresses containing the numbers to be added.[51]

Consider a program for calculating taxes. Say that the memory at address location 3 contains the amount of wages, and address 4 stores the amount of interest earned. To place total income in memory slot 5, the programmer would write the instruction:

```
128 5 3 4
```

The processor, upon reading this, would perform the desired addition. Stringing together multiple such instructions produces what legal scholarship calls *object code*, or what is colloquially called an executable file.[52]

This description of object code hopefully dispels the common notion that object code is unintelligible.[53] But because it is not pleasant

---

45 *See id.* at 242–43.

46 *See id.* at 274 ("[E]ach [instruction set architecture] provides similar types of instructions . . . ."). The last of these instructions is often called a "branch," because the path of execution splits depending on the condition. *See id.* at 260.

47 *See id.* at 248–49.

48 *Id.*

49 *See* DANIEL J. ELLARD, HARVARD UNIV. COMPUT. SCI. GRP., THE ANT-32 ARCHITECTURE - VERSION 3.1.0B 14 (2002), https://dash.harvard.edu/server/api/core/bitstreams/7312037d-c3d0-6bd4-e053-0100007fdf3b/content [https://perma.cc/G62T-X4YL] (referring, in hexadecimal notation, to instruction 0x80 for addition). This Article uses Ant-32 as its architecture of choice because it is reasonably simple to understand, and because it is the one this Author used in college.

50 *See id.* at 4, 13–14.

51 *See id.* at 14 (*des*, *src1*, and *src2*, respectively).

52 The file contains other material, such as header information. *See, e.g.*, TIS COMM., *Book I: Executable and Linking Format (ELF)*, *in* TOOL INTERFACE STANDARD (TIS) EXECUTABLE AND LINKING FORMAT (ELF) SPECIFICATION: VERSION 1.2, at 1-1, 1-4 (1995), https://refspecs.linuxfoundation.org/elf/elf.pdf [https://perma.cc/47Y4-B5QU] (describing contents of one object file format).

53 *See* Apple Comput., Inc. v. Franklin Comput. Corp., 714 F.2d 1240, 1248–49 (3d Cir. 1983); *see, e.g.*, Richard H. Stern, *Another Look at Copyright Protection of Software: Did the 1980 Act*

to write in or work with, computer scientists have taken a number of approaches to simplify the problem of writing computer instructions.

1. Instruction Names

Consider a few ways of making the aforementioned tax program easier to write. First, the numeric opcodes can be replaced with names—ADD instead of 128, for example. A programmer can thus write instructions using names[54]:

```
ADD g5, g3, g4
```

and then convert the names to opcodes, perhaps manually but better yet with a computer program that converts the names to opcode numbers. This approach, essentially object code with name translations, is an *assembly language*, and the program that translates it to machine code is an *assembler*.[55]

2. Variables

Second, because it is inconvenient to always refer to memory addresses by number, those too can be given names. Programmers call these named memory locations *variables*, in reference to, but not exactly the same as, the mathematical concept of variables as named placeholders for numbers or other matter.[56] In the tax program from above, the assembly language may allow a programmer to define names for memory addresses like so[57]:

```
.define wages, g3
.define interest, g4
```

Do Anything for Object Code?, 3 COMPUT. L.J. 1, 3 (1981) ("In fact, object code is superficially unintelligible . . . .").

54 *See* DANIEL J. ELLARD & PENELOPE A. ELLARD, HARVARD UNIV. COMPUT. SCI. GRP., ANT-32 ASSEMBLY LANGUAGE TUTORIAL (FOR VERSION 3.1.0B) 25 (2002), https://dash.harvard.edu/server/api/core/bitstreams/7312037d-c048-6bd4-e053-0100007fdf3b/content [https://perma.cc/C737-97Z6]. The 'g's indicate that the numbers are general-purpose memory slots ("registers" in assembly language parlance). *Id.* at 1–2.

55 *See id.* at 1.

56 *See, e.g.*, JAMES GOSLING, BILL JOY & GUY STEELE, THE JAVA™ LANGUAGE SPECIFICATION 43 (1996), https://archive.org/details/javalanguagespec00gosl [https://perma.cc/D3LN-87UE] ("A variable is a storage location . . . .").

57 Unfortunately, the Ant-32 assembly language does not appear to permit this, but other assembly languages do. *See, e.g.*, DEAN ELSNER, JAY FENLASON & FRIENDS, USING AS: THE GNU ASSEMBLER, VERSION 2.14, at 80 (Cygnus Support ed., 2002), https://ee209-2019-spring.github.io/references/gnu-assembler.pdf [https://perma.cc/S887-EY7W] (describing *req* directive).

```
.define income, g5
ADD wages, interest, income
```

Again, a translator program can turn the above text into equivalent machine code.[58]

Higher-level programming languages improve the above syntax in two ways. First, actual mathematical symbols can be used instead of the word ADD.[59] Second, and more importantly, the memory slot addresses can be omitted.[60] There is no particular reason why wages must be stored in slot 3, and a better translator program, typically called a *compiler*, can assign memory addresses to variables.[61] In the C programming language, one would write:

```
int wages, interest, income;
 . . .
income = wages + interest;
```

The word *int* indicates that wages, interest, and income are all variable names intended to store integer values and to be assigned memory addresses.[62] This code is significantly easier to understand than "128 5 3 4," but it is directly translatable to numerical machine code.[63]

3. Subroutines

Finally, computer instructions are more manageable when they are organized into useful groups. There are many of these,[64] but perhaps the most important are *subroutines*, also called functions or methods.[65] A subroutine is a group of instructions associated with a name and

---

58 ELLARD & ELLARD, *supra* note 54, at 25.

59 *See, e.g.*, JAMES GOSLING ET AL., *supra* note 56, at 28.

60 *See id.* at 5–6 (showing an example of Java code that does not include memory slot addresses).

61 *See, e.g.*, ANDREW W. APPEL & MAIA GINSBURG, MODERN COMPILER IMPLEMENTATION IN C 158 (1998) (describing translation of variable *v* into a memory address offset *k*). Some languages are *interpreted* rather than compiled; in these languages, a computer program effectively is translated to machine instructions, and those instructions are executed close together in time. *See What Is an Interpreted Language?*, LENOVO, https://www.lenovo.com/us/en/glossary/interpreted-language/ [https://perma.cc/7B3V-RU9P] (last visited Apr. 19, 2026).

62 *See* KERNIGHAN & RITCHIE, *supra* note 20, at 35.

63 *See* APPEL & GINSBURG, *supra* note 61, at 3.

64 For example, a programmer might write a "loop" of several instructions to be repeated multiple times. *See* GOSLING ET AL., *supra* note 56, at 274–82.

65 *See* MATTHEWS ET AL., *supra* note 42, at 24–30. There are differences among these terms, which are not relevant here. For those wondering about Java or other object-oriented languages, an instance method is a subroutine identified by a name and a taxonomic category called a "class," which are effectively a two-part name. *See* GOSLING ET AL., *supra* note 56, at 128. An instance method also has an additional argument, the object itself. *See id.*

perhaps one or more arguments.[66] The tax computation program might enclose the income computation within a subroutine like so:

```
int compute_income(int wages, int interest) {
   return wages + interest;
}
```

A programmer can *invoke* the subroutine elsewhere, like so:

```
my_income = 50000;
my_interest = 320;
income = compute_income(my_income, my_interest);
```

Again, a compiler mechanically translates these subroutines into machine code instructions.[67]

Jurists have frequently downplayed subroutines as mere shortcuts or optional timesaving elements.[68] However, the real value of subroutines is *modularity*.[69] A programmer can invoke a subroutine by knowing only its name and arguments, without knowing the instructions within it.[70] Subroutines may even be written in another programming language or executed on a distant computer over the Internet—these are examples of subroutines that are not mere shortcuts or time-savers.[71] The ADD instruction from above exemplifies modularity: A programmer, writing even in machine code, needs to know only the relevant opcode

---

66 A subroutine may also have a return value, exception handling mechanisms, and other features. *See* MATTHEWS ET AL., *supra* note 42, at 24–30. For purposes here, it is sufficient to think of these as further arguments. A return value, for example, is equivalent to an additional argument of a memory address where the subroutine can place a computational result, called "passing by pointer" or "passing by reference." *See id.* at 57.

67 How? The compiler arranges to place the subroutine's instructions at an unused memory address and records the address in association with the subroutine name. *See id.* at 50–70. The compiler translates a usage of the subroutine into a jump command directed to the subroutine's address, along with instructions to arrange the arguments in memory appropriately. *See* APPEL & GINSBURG, *supra* note 61, at 127–33 (describing the process of allocating a "stack frame" when calling a subroutine).

68 *See, e.g.*, *Oracle II*, 750 F.3d 1339, 1349 (Fed. Cir. 2014) ("[The subroutines at issue] allow programmers to use the prewritten code to build certain functions into their own programs, rather than write their own code to perform those functions from scratch. They are shortcuts.").

69 *See, e.g.*, D.L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, 15 COMMC'NS ASS'N COMPUTING MACH. 1053, 1054 (1972); PHILLIP A. LAPLANTE & MOHAMAD KASSAB, WHAT EVERY ENGINEER SHOULD KNOW ABOUT SOFTWARE ENGINEERING 146–47 (2d ed. 2023). Modularity is more typically called abstraction. *See* Timothy Colburn & Gary Shute, *Abstraction in Computer Science*, 17 MINDS & MACHS. 169, 177 (2007); DANIEL JACKSON, SOFTWARE ABSTRACTIONS 1–2 (rev. ed. 2012). But because abstraction has an unrelated meaning in copyright law, this Article avoids it to prevent confusion.

70 This generally describes the concepts of "information hiding" and "separation of concerns" in the software engineering literature. *See* EDSGER W. DIJKSTRA, *On the Role of Scientific Thought*, *in* SELECTED WRITINGS ON COMPUTING 60, 63–65 (David Gries ed., 1982); LAPLANTE & KASSAB, *supra* note 69, at 115–21.

71 *See, e.g.*, Colburn & Shute, *supra* note 69, at 179–81.

and arguments, and nothing about the underlying logic gates and transistors that perform the computation.[72] Subroutines in computer code, as named operations that programmers can invoke independent of the underlying instructions, are thus fundamental to the large-scale information society we live in today.[73]

Hopefully, this description of computer code has offered some useful intuitions. Code is a series of instructions, and those instructions operate a computer processor. Even higher-level programming languages are translated into sequential, numerical instructions that a processor and memory can execute. This explanation seemingly confirms that computer code is a method of operating a computer.[74] But consider the motivations of readability, understandability, and appearance that led from machine-executable numeric code to higher-level languages. These changes go beyond mere operation, focusing instead on code as a written, readable text.

B. Copyright and Methods of Operation

From our review of code of computers, we turn to the code of law.[75] No copyright protection, says the Copyright Act, inheres in any "method of operation."[76] Although those words arise out of the 1976 overhaul of the statute,[77] the general principle derives from the celebrated 1880 case *Baker v. Selden*.[78] A review of that case elucidates the doctrine and its application to computer code.[79]

---

72 *See id.* at 179; MATTHEWS ET AL., *supra* note 42, at 228–34.

73 Among other things, they are the basis of cloud computing, in which one computer can invoke computing routines on another remote system. *See, e.g.*, Brian Hayes, *Cloud Computing*, COMMC'NS ASS'N COMPUTING MACH., July 2008, at 9, 9.

74 *See, e.g.*, *Oracle II*, 750 F.3d 1339, 1367 (Fed. Cir. 2014) ("[C]omputer programs are by definition functional—they are all designed to accomplish some task.").

75 *Cf.* LAWRENCE LESSIG, CODE AND OTHER LAWS OF CYBERSPACE 53–54 (1999) (referring to the U.S. Code as "East Coast Code" and computer code as "West Coast Code").

76 17 U.S.C. § 102(b).

77 *See* Copyright Act of 1976, Pub. L. No. 94-553, § 102(b), 90 Stat. 2545 (codified as amended at 17 U.S.C. § 102(b)).

78 101 U.S. 99 (1880); *see id.* at 103.

79 No one has better explained this doctrine, the *Baker v. Selden* case, and their relationship to the idea–expression dichotomy better than Professor Pamela Samuelson. *See generally* Pamela Samuelson, *Why Copyright Law Excludes Systems and Processes from the Scope of Its Protection*, 85 TEX. L. REV. 1921 (2007) [hereinafter Samuelson, *Excludes*] (naming and describing the idea–expression dichotomy); Pamela Samuelson, *The Story of* Baker v. Selden*: Sharpening the Distinction Between Authorship and Invention*, *in* INTELLECTUAL PROPERTY STORIES 159 (Jane C. Ginsburg & Rochelle Cooper Dreyfuss eds., 2006) (explaining the Court's definition of authorship as determined in *Baker v. Selden*). The discussion herein owes much to Professor Samuelson's years of influential research and insights into this subject.

1. Baker v. Selden *Facts and Analysis*

Baker concerned copyright protection over Charles Selden's "system of book-keeping."[80] Contrary to popular belief,[81] Selden's system was not standard T-accounts or double-entry bookkeeping—those were invented in Italy in the 13th century[82]—but a rather ingenious arrangement of the tables that made both the transaction history and multiple account balances visible on a two-page spread.[83] Baker also published a book of forms in this two-page spread format; Selden sued Baker for copyright infringement, and Baker defended on the grounds that Selden's system was not proper subject matter for copyright protection.[84]

Rejecting the copyright claim, the Supreme Court distinguished Selden's "detailed explanations" of his bookkeeping technique from "the art which it is intended to illustrate."[85] "Art," in 1880, referred not to aesthetic works, but to human-initiated activity, particularly skilled trades.[86] The Court's examples of medicines, plows, watches, and perspective drawing techniques further explain "art" as those activities intended to achieve some function or purpose, rather than literary explanations that teach how to perform those activities.[87] This art–explanation dichotomy, said the *Baker* Court, was the proper line demarcating copyrightability, because it both properly separated the domains of copyright and patent

---

80 *Id.* at 100.

81 *See, e.g.*, Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 814 n.6 (1st Cir. 1995) (characterizing Selden's system as "the now almost-universal T-accounts system"); Am. Dental Ass'n v. Delta Dental Plans Ass'n, 126 F.3d 977, 981 (7th Cir. 1997) ("Protecting variations on the forms could have permitted the author of an influential accounting treatise to monopolize the practice of double-entry bookkeeping.").

82 *See* Geoffrey A. Lee, *The Coming of Age of Double Entry: The Giovanni Farolfi Ledger of 1299–1300*, Acct. Historians J., Fall 1977, at 79, 87, https://www.jstor.org/stable/40697544 [https://perma.cc/ZKH7-X2HK].

83 *Baker*, 101 U.S. at 100; *see* Charles Selden, Selden's Condensed Ledger, or, Book-Keeping Simplified (1861), https://tile.loc.gov/storage-services/service/rbc/rbc0001/2011/2011gen155867/2011gen155867.pdf [https://perma.cc/L5K6-PZZL].

84 *See Baker*, 101 U.S. at 101.

85 *See id.* at 102.

86 *See* 1 A New English Dictionary on Historical Principles 467 (James A.H. Murray ed., Oxford, Clarendon Press 1888), https://hdl.handle.net/2027/uva.x001541559 [https://perma.cc/P6NB-9KP4] (observing that "art" as aesthetic production "does not occur in any English Dictionary before 1880, and seems to have been chiefly used by painters and writers on painting, until the present century"); Noah Webster, An American Dictionary of the English Language 78 (Chauncey A. Goodrich & Noah Porter eds., Springfield, Mass., new ed. 1880), https://hdl.handle.net/2027/uiug.30112056431684 [https://perma.cc/2ECM-S5Y8] (defining "art" solely in terms of skills, not aesthetic products).

87 *See Baker*, 101 U.S. at 102.

law[88] and best advanced the goal of copyright law to enhance access to knowledge of science and useful arts.[89]

Modern § 102(b) codifies *Baker*'s art–explanation distinction. The statutory language, particularly "method of operation," draws directly from *Baker*.[90] And although the doctrine has posed difficulties with regard to aesthetic and fictional works,[91] courts considering utilitarian matter have hewed closely to *Baker*'s holding that "the copyright for a work describing how to perform a process does not extend to the process itself."[92]

---

88 *See id.* ("To give to the author of the book an exclusive property in the art described therein, when no examination of its novelty has ever been officially made, would be a surprise and a fraud upon the public. That is the province of letters-patent, not of copyright."); Dennis S. Karjala, *Copyright and Creativity*, 15 UCLA ENT. L. REV. 169, 194 (2008) (characterizing *Baker* as "channeling" ideas into patent law rather than copyright); Lucas S. Osborn, *Intellectual Property Channeling for Digital Works*, 39 CARDOZO L. REV. 1303, 1316–17 (2018) (also describing *Baker* as moving ideas such as accounting forms toward the protections of patents).

89 *See Baker*, 101 U.S. at 103 ("The very object of publishing a book on science or the useful arts is to communicate to the world the useful knowledge which it contains. But this object would be frustrated if the knowledge could not be used without incurring the guilt of piracy of the book.").

90 *Compare* 17 U.S.C. § 102(b) ("In no case does copyright protection for an original work of authorship extend to any . . . method of operation . . . ."), *with Baker*, 101 U.S. at 103 ("The copyright of a work on mathematical science cannot give to the author an exclusive right to the *methods of operation* which he propounds . . . ." (emphasis added)). *See also* H.R. REP. NO. 94-1476, at 57 (1976) ("Section 102(b) in no way enlarges or contracts the scope of copyright protection under the present law.").

91 In this domain, the doctrine is called the "idea–expression dichotomy," which judges have often described as difficult to apply. Peter Pan Fabrics, Inc. v. Martin Weiner Corp., 274 F.2d 487, 489 (2d Cir. 1960) (describing idea–expression dichotomy as "of necessity vague"); *see* Nichols v. Universal Pictures Corp., 45 F.2d 119, 121 (2d Cir. 1930) ("Nobody has ever been able to fix that boundary, and nobody ever can.").

92 Bikram's Yoga Coll. of India, L.P. v. Evolation Yoga LLC, 803 F.3d 1032, 1037–38 (9th Cir. 2015) (yoga pose sequences); *see, e.g.*, Publ'ns Int'l, Ltd. v. Meredith Corp., 88 F.3d 473, 480–81 (7th Cir. 1996) (recipes); Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 817 (1st Cir. 1995) (computer program menu commands); Robert R. Jones Assocs. v. Nino Homes, 858 F.2d 274, 278–79 (6th Cir. 1988) (architectural plans, prior to enactment of Architectural Works Copyright Protection Act, Pub. L. No. 101-650, 104 Stat. 5133 (1990)); RJ Control Consultants, Inc. v. Multiject, LLC, 981 F.3d 446, 455–56 (6th Cir. 2020) (technical drawings); Crume v. Pac. Mut. Life Ins. Co., 140 F.2d 182, 184 (7th Cir. 1944) (insurance company reorganization plan); R.W. Beck, Inc. v. E3 Consulting, LLC, 577 F.3d 1133, 1144–45 (10th Cir. 2009) (legal disclaimers). In earlier cases, courts have afforded copyright protection to functional materials such as telegraph codes. *See, e.g.*, Reiss v. Nat'l Quotation Bureau, Inc., 276 F. 717, 719 (S.D.N.Y. 1921). These cases are best understood as invocations of the erroneous sweat-of-the-brow doctrine, which the Supreme Court eliminated in *Feist Publications, Inc. v. Rural Telephone Service Co. See* 499 U.S. 340, 359–60 (1991).

2. *Application to Computer Code*

Despite predating computers by a century, *Baker* is a key precedent for software copyright law.[93] In particular, the art–explanation distinction precludes copyright protection for elements of a work that are methods of operation. This Article refers to these elements as "operational" elements, and they are sometimes called "functional" or "utilitarian" elements.[94] But what elements are precluded? A strict reading of the case gives the following three principles.

1. *An element of a computer program that affects the program's output, order of instructions, or memory content is a method of operation.* These three tests flow from the nature of the "art" of Selden's accounting forms—they produced specific and valid accounting results (output), were a "method" of bookkeeping (order of instructions), and stored information according to a particular organization (memory content).[95] That "any person may practise and use the art itself," per the Court, suggests that at least output, order of instructions, and memory content are operational matter not subject to copyright protection.[96] And it seems intuitive that outputs, order of instructions, and memory content are the operative elements of a computer program, given that they are the three primary operations of a computer processor.[97]

2. *An element of code may be a method of operation even if the element also expresses readable information or involves creative choice.* The *Baker* Court agreed that Selden's arrangement of headers, lines, and tables on his accounting forms was "peculiar" and nonstandard;[98] many other accounting forms—plus standard double-entry bookkeeping—were available and not preempted by Selden's system.[99] Nevertheless, the Court's constant analytical focus was not on Selden's creative choices in designing the forms, but rather the intended uses and functions of those forms once designed.[100]

---

93 *See infra* Section II.B.

94 *See* Karjala, *supra* note 88, at 180, 194.

95 *See Baker*, 101 U.S. at 100–01, 104.

96 *Id.* at 104.

97 *See supra* text accompanying notes 45–46. To be sure, this is a broad definition of methods of operation, particularly with respect to order of instructions: *(a + b) + c* would be operationally different from the mathematically equivalent *a + (b + c)*, for example. Besides such breadth comporting with the scope of *Baker*'s holding, it is acceptable for purposes here. This Article shows that computer programs contain nonoperational elements under this broad definition, *see infra* Part III, so it follows that they contain nonoperational elements under narrower definitions too.

98 *Baker*, 101 U.S. at 100.

99 *See, e.g.*, Jacob Batchelder, The National Accountant 12–28 (Boston, John P. Jewett & Co. 1847), https://babel.hathitrust.org/cgi/pt?id=mdp.39015069444399&seq=7 [https://perma.cc/JHZ9-PLXN]; Israel Alger, Key to Book-Keeping 12–43, 50–55, 61–73 (Boston, True & Greene 1823), https://babel.hathitrust.org/cgi/pt?id=hvd.hn313e&seq=5 [https://perma.cc/XCT8-A6BZ].

100 *See Baker*, 101 U.S. at 103–04.

3. *The consequence of an element in code being a method of operation is that it may be copied in other code, not just that it may be executed on a computer.* Some commentators have read *Baker* as distinguishing the use of a method of operation from the copying of the elements of the operation, such that copyright protection permits the former but not the latter.[101] If that were the case, though, then *Baker* could not have come out as it did: Selden accused Baker not of using Selden's accounting method for keeping books, but rather copying elements of Selden's forms so that third parties could use the accounting system.[102] The consequence of *Baker* was thus not just to permit uses of an art, but also to permit copying to enable others' use of that art.[103]

C. Copyright and Software

The remaining prong of the trilemma, that computer code is subject to copyright protection, is also well established but of more recent vintage. This recency owes partly to the age of the technology—computer programs were not around in 1880[104]—and is partly because the path to recognizing copyright in computer code was neither uniform nor straightforward.

Initially, it was unclear whether computer code was eligible for copyright protection.[105] In 1965, the Copyright Office acknowledged "doubtful questions" about "whether [a computer] program is such as a 'writing of an author' and thus copyrightable"; the Office was willing to issue registrations to programs but only "in accordance with the policy of resolving doubtful issues in favor of registration whenever possible."[106]

---

101 *See, e.g.*, NAT'L COMM'N ON NEW TECH. USES OF COPYRIGHTED WORKS, FINAL REPORT OF THE NATIONAL COMMISSION ON NEW TECHNOLOGICAL USES OF COPYRIGHTED WORKS 18–19 (1978) [hereinafter CONTU REPORT].

102 *See Baker*, 101 U.S. at 100.

103 Later courts have described this result of *Baker* as "merger," but *Baker* goes beyond how the merger doctrine is typically articulated. Merger is said to apply when "specific instructions, even though previously copyrighted, are the only and essential means of accomplishing a given task." Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 708 (2d Cir. 1992) (quoting CONTU REPORT, *supra* note 101, at 20), *quoted in Oracle II*, 750 F.3d 1339, 1360 (Fed. Cir. 2014). That formulation would not explain the uncopyrightability of Selden's forms, because there were multiple other means of accomplishing bookkeeping. *See, e.g.*, BATCHELDER, *supra* note 99, at 12–28; ALGER, *supra* note 99, at 12–43, 50–55, 61–73. Given that courts have typically applied the merger doctrine this way, the term "merger" seems inapposite and thus is avoided throughout this Article.

104 *But cf.* Augusta Ada Lovelace, *Notes by the Translator* for L.F. Menabrea, *Sketch of the Analytical Engine Invented by Charles Babbage Esq.*, *in* 3 SCIENTIFIC MEMOIRS 691, 724 (Richard Taylor ed., London, Richard & John E. Taylor 1843), https://babel.hathitrust.org/cgi/pt?id=uiug.30112069293139&seq=7 [https://perma.cc/7YPV-VKAH] (attributed to be the first written computer program).

105 *See* Samuelson, *Excludes*, *supra* note 79, at 1946–50.

106 *See* U.S. COPYRIGHT OFF., COPYRIGHT OFFICE CIRCULAR 31D (Jan. 1965), *reprinted in* Duncan M. Davidson, *Protecting Computer Software: A Comprehensive Analysis*, 1983 ARIZ. ST. L.J.

As the legislative debate over revising copyright law began in that same decade, multiple stakeholders questioned whether copyright protection should inhere in computer software, on grounds of legal doctrine, industry policy, and the philosophical nature of machine-executable instructions.[107]

In the 1976 Act, Congress took no position on whether computer code was copyrightable. As explained in the House Report, Congress deemed it "premature to change existing law on computer uses at present," and thus drafted the 1976 Act's provisions "neither to cut off any rights that may now exist, nor to create new rights that might be denied under the Act of 1909 or under common law principles currently applicable."[108] This legislative intent was embodied in § 117 of the 1976 Act, which, as then enacted, "does not afford to the owner of copyright in a work any greater or lesser rights with respect to the use of the work in conjunction with automatic systems capable of storing, processing, retrieving, or transferring information."[109]

To the extent that computer software was copyright protected, however, the legislative history recognized that such protection did not supersede *Baker* and the unprotectability of methods of operation. As stated in the House Report, "[s]ection 102(b) is intended, among other things, to make clear that the expression adopted by the programmer is the copyrightable element in a computer program, and that the actual processes or methods embodied in the program are not within the scope of the copyright law."[110] The report did not explain further what that expression was.

Congress did, however, take an important separate step toward resolving the software copyright question. In 1974, Congress established the National Commission on New Technological Uses of Copyrighted Works ("CONTU").[111] Among other things, the new commission was directed to study the application of copyright law to computer systems, and to "make recommendations as to such changes in copyright law or procedures" based on that study.[112]

Based on hearings, public comments, and commissioned studies, CONTU produced a report and recommendations in 1978.[113] Over the dissents of at least two commissioners, CONTU concluded that computer

611, 652 n.72.

107 *See* Samuelson, *Excludes*, *supra* note 79, at 1946–48.

108 H.R. Rep. No. 94-1476, at 116 (1976).

109 Copyright Act of 1976, Pub. L. No. 94-553, § 117, 90 Stat. 2541, 2565.

110 H.R. Rep. No. 94-1476, at 57.

111 Act of Dec. 31, 1974, Pub. L. No. 93-573, § 201(a), 88 Stat. 1873, 1873.

112 *Id.* § 201(b)(1)(A), (c), 88 Stat. at 1873–74.

113 *See* CONTU Report, *supra* note 101.

code should be covered by copyright protection.[114] The report recommended that Congress implement copyright protection not by explicitly codifying the copyrightability of computer programs, but rather implicitly, by repealing § 117 of the 1976 Act—the provision expressing doubt about software copyrightability—and replacing it with a new provision relating to the scope of copyright infringement in computer programs.[115]

Without substantial debate, Congress enacted CONTU's recommendations into law in 1980.[116] Despite the lack of a clear textual statement and a thin legislative record,[117] courts have almost uniformly treated these amendments as ratifying the CONTU Report in general, and the copyrightability of software in particular.[118] As a result, courts have repeatedly held that "[a]s literary works, copyright protection extends to computer programs."[119]

II. Four Approaches over Five Decades

As discussed above, modern copyright doctrine holds that methods of operation are not copyrightable, but computer software is.[120] On the assumption that computer code is nothing more than a method of operating a computer, these two doctrines are fundamentally irreconcilable. Seemingly, a court must either adhere to § 102(b) and disregard the plain legislative intent in § 117 or provide protection to computer code and somehow bend § 102(b).

---

114 *See id.* at 12 ("Copyright should proscribe the unauthorized copying of these works."). Commissioner John Hersey dissented from this conclusion, joined by Commissioners Rhoda H. Karpatkin and William S. Dix (who passed away prior to the report's publication). *See id.* at 27 (Hersey, Comm'r, dissenting); *id.* at 37 (Karpatkin, Comm'r, dissenting). Vice Chairman Melville B. Nimmer, in a concurrence, "share[d] in a number of the doubts and concerns expressed in Commissioner Hersey's thoughtful dissenting opinion." *See id.* at 26 (Nimmer, Vice Chairman, concurring).

115 *See id.* at 12–13.

116 Bayh-Dole Act, Pub. L. No. 96-517, 94 Stat. 3015, 3028–29 (1980) (codified as amended at 17 U.S.C. § 117). The only substantial change was the replacement of the words "rightful possessor" with "owner." 17 U.S.C. § 117; CONTU Report, *supra* note 101, at 12.

117 For the legislative history of the provision, see Michael S. Keplinger, *Computer Software—Its Nature and Its Protection*, 30 Emory L.J. 483, 502–05 (1981).

118 *See, e.g.*, Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 703–04 (2d Cir. 1992) ("To that end, Congress adopted CONTU's suggestions . . . ."); Apple Comput., Inc. v. Franklin Comput. Corp., 714 F.2d 1240, 1252 (3d Cir. 1983) ("[W]e can consider the CONTU Report as accepted by Congress since Congress wrote into the law the majority's recommendations almost verbatim.").

119 Atari Games Corp. v. Nintendo of Am. Inc., 975 F.2d 832, 838 (Fed. Cir. 1992); *see, e.g.*, Johnson Controls, Inc. v. Phx. Control Sys., Inc., 886 F.2d 1173, 1175 (9th Cir. 1989) ("Computer software is subject to copyright protection."); *Comput. Assocs.*, 982 F.2d at 702 ("It is now well settled that the literal elements of computer programs, i.e., their source and object codes, are the subject of copyright protection.").

120 *See supra* Section I.C.

In the five decades since the 1976 Act, courts, policymakers, and commentators have struggled with the tension between § 102(b) and § 117. They have effectively taken four approaches to deal with this jurisprudential puzzle.

Under the *abrogation* approach, the assumption is that § 102(b) no longer stands for the broad holding of *Baker* as described above.[121] This assumption could be either because § 117, coming later in time, implicitly cut back on § 102(b),[122] or because intervening decisions subsequently limited *Baker*, with a common candidate being *Mazer v. Stein*.[123] The consequence of this approach is that it enables courts to hold that methods of operation are copyrightable, at least in some situations, contrary to § 102(b).[124]

A common subspecies of this abrogation approach is the *creative functionality* argument, under which a method of operation is deemed nevertheless copyrightable because there are multiple ways of performing that operation, or because some creativity went into producing the operational element.[125] As discussed above, this is inconsistent with *Baker*, in which the Supreme Court rejected a copyright claim to a form that was "peculiar" and just one of many options for accounting form design.[126]

Under the *unidentified-expression* approach, by contrast, the assumption is that computer code is a combination of operational and expressive material, and that the consequence of taking § 117 in conjunction with § 102(b) is that only the expressive content is copyrightable subject matter.[127] This is the position that this Article takes. But, unlike this Article, the prior literature generally does not identify what that expressive material actually is, treating the expressive content like an unknown subatomic particle in physics—the theoretical laws require that such a thing exist, though it has never been experimentally observed.[128] As a result, the unidentified-expression approach lacks explanatory power. It suffices to prove infringement in cases of literal copying of software—because everything, including the unidentified

---

121 *See supra* Section I.B.

122 *See, e.g.*, Google LLC v. Oracle Am., Inc., 593 U.S. 1, 45–49 (2021) (Thomas, J., dissenting).

123 347 U.S. 201, 217 (1954); *see infra* text accompanying notes 156–62.

124 *See, e.g.*, Apple Comput., Inc. v. Formula Int'l Inc., 725 F.2d 521, 524 (9th Cir. 1984) (holding computer programs controlling the internal operations of computer are entitled to copyright protection under 17 U.S.C. § 117).

125 *See, e.g.*, *Oracle II*, 750 F.3d 1339, 1361–63 (Fed. Cir. 2014).

126 Baker v. Selden, 101 U.S. 99, 100 (1879); *see supra* text accompanying notes 99–100.

127 *See, e.g.*, H.R. Rep. No. 94-1476, at 57 (1976); *see supra* text accompanying note 90.

128 *See* CONTU Report, *supra* note 101, at 22–23; *cf.* Gianfranco Bertone & Tim M.P. Tait, *A New Era in the Search for Dark Matter*, 562 Nature 51, 51 (2018) (discussing challenges in the search for dark matter particles).

expression, must have been copied—but it cannot provide definitive answers beyond that.

Closely related to unidentified expression is the *infinitesimality* approach. Here, the reasoning is that because § 102(b) excludes methods of operation, and computer code is a method of operation, then whatever copyrightable expression remains is so thin that it can only be infringed by literal copying.[129] In many cases, proponents of this approach likely would find wholesale uncopyrightability of software to be more doctrinally satisfying, but accept this infinitesimal level of protection in view of the established § 117 case law.[130]

TABLE 1. TABLE OF APPROACHES

Approach	Description
Abrogation	*Baker*'s prohibition on copyrightability of methods of operation has been abrogated or limited with respect to software.
Creative functionality	Specifically, *Baker* does not apply where there are multiple or creative options for operational matter.
Unidentified expression	Computer software contains some nonoperational expressive matter, which is not identified particularly.
Infinitesimality	The nonoperational expressive matter in software is so thin and unidentifiable that it can only be infringed by literal copying.

The table shows these four major approaches to resolving the copyrightability of computer code, as found in case law and commentary.

A. CONTU and Its Contemporaries

CONTU and Congress of course had to come to terms with the tension between copyright for software and the uncopyrightability of methods of operation.[131] Certainly, doctrinal consistency was not the sole objective for these policymakers—economics, policy, and politics likely played a larger role.[132] Nevertheless, taking the written record

129 *See, e.g.*, Pamela Samuelson, Randall Davis, Mitchell D. Kapor & J.H. Reichman, *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 COLUM. L. REV. 2308, 2353–55 (1994).

130 *See* Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 820 (1st Cir. 1995) (Boudin, J., concurring).

131 *See* CONTU REPORT, *supra* note 101, at 18–23.

132 *See id.* at 23–26. Indeed, CONTU's primary argument for the copyrightability of computer code was oddly inconsistent. The Commission took it as "clear that those who wrote the Copyright Act of 1976 . . . concur in the position that programs are copyrightable." *Id.* at 16. Yet just nine pages earlier, the same report stated that "the drafters of the statute explicitly stated that it

at face value, Congress and the Commission relied on the unidentified-expression approach. The House Report on the 1976 Act stated that § 102(b) was "intended, among other things, to make clear that the expression adopted by the programmer is the copyrightable element in a computer program, and that the actual processes or methods embodied in the program are not within the scope of the copyright law," but it says nothing about what the "expression adopted by the programmer" is.[133] CONTU similarly observed that it was "difficult, either as a matter of legal interpretation or technological determination, to draw the line between the copyrightable element of style and expression in a computer program and the process which underlies it," leaving the line to be "drawn on a case-by-case basis by the . . . federal judiciary."[134]

The Commission majority's dialogue with the dissenting commissioners further confirms that the unidentified-expression approach was taken. The dissenters essentially followed the prongs of the trilemma to their logical conclusion: "[A] computer program is not designed to be read by anyone";[135] copyright should be limited to works that "communicate the work's means of expression"[136] as opposed to those that are "simply a mechanical substitute for human labor" (i.e., methods of operation);[137] and, therefore, copyright should not subsist in computer programs.[138] In response, the majority asserted that computer programs "are *capable* of communicating with humans," but nevertheless acknowledged in the same paragraph that "how copyright protects [computer programs] is not universally understood," thus suggesting the presence of communicative expression but declining to precisely identify it.[139]

Although the CONTU Report left the expressive elements of software vague, individual commissioners and surrounding commentators were more specific. A National Institute of Standards and Technology report cited by the Commission took the creative-functionality approach, arguing that computer programs are "more than simply a set of instructions used to operate a machine" based on "their operational

---

did not address or deal with computer issues." *Id.* at 7; *see also* Pamela Samuelson, *CONTU Revisited: The Case Against Copyright Protection for Computer Programs in Machine-Readable Form*, 1984 DUKE L.J. 663, 703 (describing CONTU's characterization of the 1976 Act as "misleading").

133 H.R. REP. No. 94-1476, at 57; *see also id.* at 54 (characterizing computer programs as literary works "to the extent that they incorporate . . . expression of original ideas, as distinguished from the ideas themselves").

134 CONTU REPORT, *supra* note 101, at 22–23.

135 *Id.* at 30 (Hersey, Comm'r, dissenting).

136 *Id.* at 29.

137 *Id.*

138 *See id.* at 27.

139 *Id.* at 20–21 (majority).

use, in a variety of real human purposes," for human benefit.[140] Commissioner Melville B. Nimmer, author of a well-known copyright treatise,[141] proffered a disputed theory of abrogation, namely that the Supreme Court in *Mazer* applied the idea–expression dichotomy doctrine of aesthetic works to utilitarian works as well, thereby narrowing or eliminating the method-of-operation principles of *Baker*.[142] Commissioner Arthur R. Miller appears to have evolved from *infinitesimality* to *creative functionality* over time. Miller wrote in 1967 of "difficulties" in copyright protection for software beyond literal "duplicating [of] the set of instructions constituting the program,"[143] but in 1993, he concluded that programs "are expressive" because programmers would not independently write programs with "the same structural details, let alone precisely the same set of instructions."[144] These conflicting views, in the mixing pot of committee compromise, perhaps explain CONTU's ambiguous stance.[145]

B. Case Law

The case law on the copyrightability of software similarly reflects the four different approaches to resolving the trilemma.

---

140 *See* Roy G. Saltman, Nat'l Bureau of Standards, U.S. Dep't of Com., NBS Special Pub. 500-17, Copyright in Computer-Readable Works: Policy Impacts of Technological Change 60 (1977), https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-17.pdf [https://perma.cc/5QYP-WURD], *cited in* CONTU Report, *supra* note 101, at 138; *see also id.* at 61 (suggesting that complexity of function gives rise to a "variety of unique ways of expressing the steps in the performance").

141 *See* Melville B. Nimmer & David Nimmer, Nimmer on Copyright (rev. ed. 2025).

142 *See* Samuelson, *Excludes*, *supra* note 79, at 1956–61 (describing and critiquing Nimmer's interpretation of *Mazer*); Weinreb, *supra* note 16, at 1174 n.103. These articles relied on recent editions of Nimmer's treatise. Editions contemporary to the CONTU Report do not appear to be available, but one can triangulate Nimmer's sentiments about *Mazer* and *Baker* from various quoting sources. *See, e.g.*, CONTU Report, *supra* note 101, at 18–20; Melville B. Nimmer, *The Subject Matter of Copyright Under the Act of 1976*, 24 UCLA L. Rev. 978, 990–91, 994–96 (1977); Scholz Homes, Inc. v. Maddox, 379 F.2d 84, 86 n.1 (6th Cir. 1967).

143 *See* Arthur R. Miller, *Computers and Copyright Law*, Mich. St. Bar J., Apr. 1967, at 11, 16.

144 Arthur R. Miller, *Copyright Protection for Computer Programs, Databases, and Computer-Generated Works: Is Anything New Since CONTU?*, 106 Harv. L. Rev. 977, 983–84 (1993).

145 *See* Samuelson, *Excludes*, *supra* note 79, at 1954 n.212 (characterizing the CONTU Report as "highly ambiguous and remarkably shallow").

1. Cases Favoring Copyrightability

Early cases applied both the abrogation[146] and creative-functionality[147] approaches. *Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.*,[148] the most controversial of these,[149] applied the creative-functionality approach to find copyright infringement when the exact code had not been literally copied.[150] Relying on the "numerous ways the programmer can solve the data-organization problems she or he faces,"[151] the court held that only "the purpose or function of a utilitarian work would be the work's idea, and everything that is not necessary to that purpose or function would be part of the expression."[152] Despite recognizing that many of those programmer choices could have operational effects, the court concluded that the mere existence of "various means of achieving the desired purpose" was sufficient to qualify as protectable expression.[153] Indeed, the court admitted that it was conferring copyright protection upon operational material, contending that the "structure and logic of the program" were the programmer's "most valuable efforts" and merited copyright as a policy matter.[154]

---

146 *See, e.g.*, Autoskill Inc. v. Nat'l Educ. Support Sys., Inc., 994 F.2d 1476, 1495 n.23 (10th Cir. 1993) (holding that *Feist Publications, Inc. v. Rural Telephone Service Co.*'s originality test, *see* 499 U.S. 340 (1991), renders protectable a program's use of the 1, 2, and 3 keys to input a response, despite § 102(b)); Apple Comput., Inc. v. Formula Int'l Inc., 725 F.2d 521, 524 (9th Cir. 1984) (observing that application of § 102(b) to processes within computer program "is contrary to the language of the Copyright Act, the legislative history of the Act, and the existing case law concerning the copyrightability of computer programs").

147 *See, e.g.*, Atari Games Corp. v. Nintendo of Am. Inc., 975 F.2d 832, 840 (Fed. Cir. 1992) (holding that "arbitrary programming instructions . . . arranged . . . in a unique sequence to create a purely arbitrary data stream" are a "creative element" protected under copyright); Johnson Controls, Inc. v. Phx. Control Sys., Inc., 886 F.2d 1173, 1176 (9th Cir. 1989) (agreeing "that some discretion and opportunity for creativity exist in the structure, and that the structure of the [computer program at issue] is expression, rather than an idea in itself"); Eng'g Dynamics, Inc. v. Structural Software, Inc., 26 F.3d 1335, 1345 (5th Cir. 1994) (positing that existence of "competing structural design programs" that were "dissimilar" meant that "[a]s a matter of law, the input formats and output reports do not embody only noncopyrightable 'facts'"); Mitel, Inc. v. Iqtel, Inc., 124 F.3d 1366, 1374 (10th Cir. 1997) (holding numeric command codes for operating computer hardware potentially copyrightable due to the "effort required of [programmers] to devise appropriate values for the wide variety of individual functions").

148 797 F.2d 1222 (3d Cir. 1986).

149 *See* Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 705–06 (2d Cir. 1992) (noting and citing criticism of *Whelan*).

150 *See Whelan Assocs.*, 797 F.2d at 1228–29 (noting that defendant's program was "not a direct transliteration of" the plaintiff's code).

151 *Id.* at 1230.

152 *Id.* at 1236 (emphasis omitted).

153 *Id.* at 1236 & n.28; *cf. id.* at 1230 ("Each solution may have particular characteristics—efficiencies or inefficiencies, conveniences or quirks—that . . . make the overall program more or less desirable.").

154 *Id.* at 1237.

While *Whelan* focused primarily on creative functionality,[155] *Apple Computer, Inc. v. Franklin Computer Corp.*[156] exemplified abrogation. The question there was whether copyright protection extended to the object code of an operating system program.[157] Franklin, the alleged infringer, asserted that an operating system was an uncopyrightable method of operation under § 102(b).[158] The court, however, found that § 102(b) argument "inconsistent" with the agreed view "that application programs are an appropriate subject of copyright."[159] Although *Baker* "could support Franklin's reading," the court said, "that interpretation has been rejected by a later Supreme Court decision," namely, *Mazer*.[160] *Apple* further cited the CONTU Report as having "rejected the expansive view some courts have given *Baker*," and the court deemed the report "accepted by Congress."[161] Accordingly, the court held § 102(b) overridden by the post-CONTU amendments.[162]

2. *Cases Limiting Copyrightability*

Other cases have applied the unidentified-expression and infinitesimality approaches.[163] The most prominent of these, which has given rise to a leading test for software copyrightability, is *Computer Associates International, Inc. v. Altai, Inc.*[164] Akin to *Whelan*, *Computer Associates* involved a computer program where the alleged infringer had produced a "clean-room" rewrite of a program, in which new programmers,

---

155 *Whelan* also touches on abrogation. *Id.* at 1234–38. To deal with the tension between the Third Circuit's broad view of software copyright and *Baker*'s broad statements against copyright in methods of operation—Selden too had "various means of achieving the desired purpose" of bookkeeping, *see supra* text accompanying notes 95–96—the court followed Nimmer's abrogation analysis to deem *Baker* significantly narrowed by subsequent idea–expression case law on aesthetic works. *Whelan Assocs.*, 797 F.2d at 1236; *see id.* at 1234–38.

156 714 F.2d 1240 (3d Cir. 1983).

157 *See id.* at 1245–46.

158 *Id.* at 1250.

159 *Id.* at 1251.

160 *Id.* at 1252.

161 *Id.*

162 *See id.*

163 *See* Gates Rubber Co. v. Bando Chem. Indus., 9 F.3d 823, 836–37 (10th Cir. 1993) (noting that "the main purpose or function of a program will always be an unprotectable idea," and then that unprotectable "processes can be found at any level, except perhaps the main purpose level of abstraction"); Apple Comput., Inc. v. Microsoft Corp., 35 F.3d 1435, 1439 (9th Cir. 1994) ("[T]o the extent there is creative expression left in how the works are put together, as a whole they can receive only limited protection."); Brown Bag Software v. Symantec Corp., 960 F.2d 1465, 1476 (9th Cir. 1992) (affirming district court's determination of "five groups of features found to be unprotected or unprotectable"); Yield Dynamics, Inc. v. TEA Sys. Corp., No. c 06-0331, 2007 U.S. Dist. LEXIS 113956, at *15 (N.D. Cal. Feb. 21, 2007) ("[P]laintiff must produce evidence that defendants' works are virtually identical . . . .").

164 982 F.2d 693, 706–11 (2d Cir. 1992).

unfamiliar with the copyright holder's computer program and having no access to it, wrote a new program with the same functionality as the copyright holder's program.[165] The Second Circuit saw the trilemma: "[C]omputer programs are literary works, as we are told by the legislature," but it is a "fundamental principle of copyright law that a copyright does not protect an idea."[166] The "essentially utilitarian nature of a computer program," the court wryly observed, "further complicates the task" of applying the law.[167]

To reconcile these doctrines, the *Computer Associates* court announced a three-step procedure in which courts were to (1) assess layers of abstractions within a computer program, (2) perform filtration on those layers of abstractions to discard uncopyrightable elements, and (3) compare the remaining, unfiltered elements to determine copyright infringement.[168] Elucidating upon this abstraction-filtration-comparison test, the Second Circuit identified numerous uncopyrightable elements in view of *Baker* and § 102(b), such as elements dictated by efficiency, user comfort, conventional programming techniques, compatibility, industry standards, hardware constraints, and more.[169] But what were the *copyrightable* elements? "To be frank," said the court, "the exact contours of copyright protection for nonliteral program structure are not completely clear," and are to be left "as future cases are decided."[170] In other words, the court left the expressive material in software unidentified—and hinted that such material might be infinitesimally thin, leaving copyright "a relatively weak barrier against public access."[171] Such a limited role for copyright in the software area, thought the court, might be acceptable as a policy matter for "a highly functional, utilitarian component in the larger process of computing."[172]

Lotus Development Corp. v. Borland International, Inc.,[173] decided three years later, pushed the infinitesimality envelope even further. The issue there was whether Lotus's hierarchy of spreadsheet program menu commands was copyrightable when those menu commands also served as subroutine names that programmers could invoke in "macro"

165 *See id.* at 700.
166 *Id.* at 702–03.
167 *Id.* at 704.
168 *See id.* at 706–11.
169 *See id.* at 707–11.
170 *Id.* at 712.
171 *Id.*
172 *Id.*
173 49 F.3d 807 (1st Cir. 1995).

computer programs to control the spreadsheet program.[174] Initially,[175] the First Circuit considered and rejected the abstraction-filtration-comparison test of *Computer Associates*.[176] That test, said the court, assumes "a base level that includes copyrightable subject matter," an assumption with which the court parted ways.[177] Instead, the *Lotus* court applied § 102(b) directly, holding that the Lotus menu command hierarchy, "as the method by which the program is operated and controlled," was "an uncopyrightable 'method of operation.'"[178] The court considered and rejected the creative-functionality approach, acknowledging that "the Lotus developers made some expressive choices" in the menu names but still treating those choices as "essential to operating" the spreadsheet program.[179]

But if the method-of-operation provision of § 102(b) excluded menu commands from copyright protection in their entirety, what was left of software copyright?[180] Applying the unidentified-expression approach, the *Lotus* majority observed that its holding did not apply to "the underlying computer code" of the spreadsheet program and took "no position on whether it is copyrightable."[181] But based on the court's rejection of the view that every computer program has "a base level that includes copyrightable subject matter,"[182] the First Circuit must have viewed the scope of protectable matter in software to be exceptionally thin. Judge Boudin's concurrence to *Lotus* portrayed that protectable nugget as even more infinitesimal, writing that § 102(b), when "taken literally[,] might easily seem to exclude most computer programs from protection."[183] Whether Judge Boudin intended the statute to be taken that way—he left further consideration to future courts—his discussion

---

174 *See id.* at 809–10; *cf. supra* text accompanying notes 64–73 (describing subroutines in computer code). Borland copied Lotus's menu command names literally, so that users switching from Lotus's to Borland's product could reuse their existing Lotus macro programs with as little alteration as possible. *Lotus*, 49 F.3d at 810.

175 Before this, the court deemed *Baker* inapplicable, *Lotus*, 49 F.3d at 813–14, but this is less relevant than it might seem. Borland had argued that the cells of a computer spreadsheet were analogous to the blank forms in *Baker*, but the court correctly observed that the menu commands, not the spreadsheet cells, were at issue. *Id.* at 814. So only the narrower blank-forms rule of *Baker* was at issue in that discussion. *Cf. id.* at 816–17 (applying *Baker* for its methods-of-operation principle).

176 *Id.* at 815.

177 *Id.*

178 *Id.*

179 *Id.* at 816.

180 *Cf. id.* at 817 ("Computer programs . . . are copyrightable as 'literary works.'" (citing 17 U.S.C. § 102(a))).

181 *Id.* at 816 & n.11.

182 *Id.* at 815.

183 *Id.* at 820 (Boudin, J., concurring).

of monopolization and lock-in suggests that he likely would have applied the method-of-operation doctrine broadly.[184]

C. Oracle v. Google

In a sense, the cases outlining these four approaches were a prelude to the "Copyright Case of the Century," the decade-long litigation between computer software giants Oracle and Google.[185] The four approaches were pitted against one another throughout the litigation, and the decisions are a window into the difficulty of applying software copyright law.

1. Facts and District Court Opinion

Oracle, through corporate acquisition, owns the intellectual property rights in the Java programming language and software toolkits originally developed at Sun Microsystems.[186] Among other things, this includes copyrights in "Java SE," a collection of computer subroutine programs that Sun—and later Oracle—made available.[187] Like any other subroutines,[188] those in Java SE had names, taxonomic categories, and argument lists.[189] Computer scientists would typically call these subroutine declarations, prototypes, signatures, or an application programming interface ("API");[190] the attorneys in the *Oracle* litigation devised the name "declaring code" for them.[191] Google, as part of its new Android mobile device software platform, developed its own Java-like language and associated software, including a collection of subroutines with the

---

184 *See id.* at 820–21.

185 Jasmine Garsd, *What Google's Copyright Win Means for Other Industries*, MARKETPLACE (Apr. 6, 2021), https://www.marketplace.org/2021/04/06/what-googles-copyright-win-means-for-other-industries/ [https://perma.cc/3FRB-8PVT]; Scott Graham, *Supreme Court, Finally, Takes Up* 'Google v. Oracle,' NAT'L L.J. (Nov. 15, 2019, at 14:17 ET) (quoting Professor Mark Lemley), https://www.law.com/nationallawjournal/2019/11/15/supreme-court-finally-takes-up-google-v-oracle/ [https://perma.cc/6YQP-L2ZR].

186 The facts of the case are set forth most clearly in the district court decision. *See* Oracle Am., Inc. v. Google Inc. (*Oracle I*), 872 F. Supp. 2d 974, 975, 977–84 (N.D. Cal. 2012), *rev'd*, *Oracle II*, 750 F.3d 1339 (Fed. Cir. 2014).

187 *See id.* at 977–78. The name of the subroutine collection is not clear. The Federal Circuit calls it alternately the Java Standard Edition Platform (Java SE) and the Java API, *Oracle II*, 750 F.3d at 1349, while the district court exclusively uses Java API, *Oracle I*, 872 F. Supp. 2d at 977. Because API has another meaning, Java SE is used in this Article to avoid confusion.

188 *See supra* text accompanying notes 64–73.

189 *See Oracle I*, 872 F. Supp. 2d at 977–78.

190 *See* GOSLING ET AL., *supra* note 56, at 155, 157; KERNIGHAN & RITCHIE, *supra* note 20, at 27; *cf. Oracle I*, 872 F. Supp. 2d at 977–78; *Oracle II*, 750 F.3d at 1349 (using the term "API").

191 *Oracle II*, 750 F.3d at 1349. Following industry convention, this Article will use "declaration."

same names, groupings, and argument lists as parts of Java SE.[192] Oracle contended that this Android software was literal infringement of copyright in the subroutine declarations, although Google responded that the declarations were not copyrightable subject matter.[193]

The district court, agreeing with Google, applied the unidentified-expression approach.[194] Initially, the court drew a distinction between subroutine *declarations* and *implementation*, namely, the computing instructions within a subroutine.[195] That distinction, said the court, corresponded to copyright law and particularly the idea–expression doctrine: "The method specification is the idea. The method implementation is the expression."[196] The court reasoned extensively about why the declaration was uncopyrightable,[197] but what about the implementation? "Each method has a singular purpose or function, and so, the basic function or purpose of a method will be an unprotectable process," said the court in a footnote.[198] The district court thus acknowledged that the copyrightable element of computer code was something in the implementing code, but not everything in it.[199]

2. *Appeal to the Federal Circuit*

Reversing the district court, the Court of Appeals for the Federal Circuit in *Oracle America, Inc. v. Google Inc.* ("*Oracle II*")[200] applied the abrogation approach instead. The court characterized § 102(b) as merely embodying the idea–expression dichotomy, as its citation to *Mazer* for the doctrine was in line with the narrow-view tradition.[201] It then considered how to reconcile § 102(a), which provided copyright protection to literary works including computer code, with § 102(b)'s prohibition on protection extending to methods of operation.[202]

---

192 *Oracle I*, 872 F. Supp. 2d at 978–79. It is worth noting the close similarities between these facts and those of *Lotus*. *Cf. supra* notes 173–84 and accompanying text (discussing the case *Lotus Development Corp. v. Borland International, Inc.*, 49 F.3d 807 (1st Cir. 1995)). Indeed, Google's reason for using Java SE's declarations was akin to Borland's: "Google believed Java application programmers would want to find the same 37 sets of functionalities in the new Android system callable by the same names as used in Java." *Oracle I*, 872 F. Supp. 2d at 978; *cf.* Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 810 (1st Cir. 1995) (observing that Borland copied Lotus's menu command names so users familiar with Lotus's product could reuse existing macro programs on Borland products).

193 *See Oracle I*, 872 F. Supp. 2d at 975.

194 *See id.* at 997.

195 *See id.*

196 *Id.* at 998 (emphasis omitted).

197 *See id.* at 998–1002 (explaining that method declarations are methods of operation).

198 *Id.* at 998 n.8.

199 *See id.* at 998.

200 750 F.3d 1339 (Fed. Cir. 2014).

201 *See id.* at 1354 (citing Mazer v. Stein, 347 U.S. 201, 217 (1954)).

202 *Id.* at 1356–57.

Rejecting Google's analysis that § 102(b) acts as a limit on the broader grant of § 102(a), the Federal Circuit relied on a legislative history statement that § 102(b) "in no way enlarges or contracts the scope of copyright protection," meaning the two provisions had to be read in tandem.[203] Implicit in this analysis was that the prohibition on copyright protection for methods of operation, clearly and broadly delineated in *Baker*, had been abrogated in view of § 102(a) and other legal developments, leaving that prohibition hazily subject to reinterpretation.[204]

Applying this hazy standard, *Oracle II* leaned heavily on the creative-functionality approach to deem the declarations copyrightable. Chiefly, the Federal Circuit repeatedly observed that "Oracle had 'unlimited options as to the selection and arrangement of the 7000 lines Google copied.'"[205] Even if those copied lines were methods of operation, said the Federal Circuit, the presence of creative choices and alternatives sufficed to render them copyrightable.[206]

Toward the end of its copyrightability analysis, the court addressed the trilemma directly, in responding to the district court's holding that the taxonomy of declarations was a method of operation:

> The problem with the district court's approach is that computer programs are by definition functional—they are all designed to accomplish some task. . . . If we were to accept the district court's suggestion that a computer program is uncopyrightable simply because it "carr[ies] out pre-assigned functions," no computer program is protectable. That result contradicts Congress's express intent to provide copyright protection to computer programs, as well as binding Ninth Circuit case law finding computer programs copyrightable, despite their utilitarian or functional purpose.[207]

The only way out of the trilemma, in the Federal Circuit's view, was to bend § 102(b) as it had done.

---

203 *Id.* (quoting Feist Publ'ns, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340, 356 (1991)).

204 *See id.* at 1357 ("Thus, this test eschews bright line approaches and requires a more nuanced assessment of the particular program at issue in order to determine what expression is protectable and infringed.").

205 *Id.* at 1361 (quoting Oracle's brief); *see also id.* at 1363 ("Because Oracle 'exercised creativity in the selection and arrangement' of the method declarations when it created the API packages and wrote the relevant declaring code, they contain protectable expression that is entitled to copyright protection." (quoting Atari Games Corp. v. Nintendo of Am. Inc., 975 F.2d 832, 840 (Fed. Cir. 1992))); *id.* at 1365 ("[I]t is undisputed here that the declaring code and the structure and organization of the API packages are both creative and original.").

206 *See id.* at 1367–68.

207 *Id.* at 1367 (alteration in original).

3. Supreme Court

Following the Federal Circuit's determination of copyrightability, the case returned to the district court for further proceedings on fair use, and then, in a subsequent appeal, the Federal Circuit reversed the district court's fair use determination.[208] The Supreme Court granted certiorari, enabling it to review both Federal Circuit decisions on copyrightability and fair use.[209] Although the majority opinion declined to address copyrightability,[210] its opinion on fair use and Justice Thomas's dissent nevertheless reflect the gamut of approaches to the trilemma.

Justice Thomas openly favored the abrogation approach.[211] After citing § 117 and other statutes for the proposition that copyright law "expressly protects computer code,"[212] his dissent addressed whether § 102(b) excludes declarations as methods of operation.[213] To resolve the conflict, Justice Thomas treated the computer-specific provision as overriding, such that "[w]hen faced with general language barring protection for 'methods of operation' and specific language protecting declaring code, the 'specific governs the general.'"[214] The methods-of-operation provision, in his view, was relegated to preventing copyright protection on the abstract "idea of using declaring code," whatever that code may be.[215] The dissent also hinted at the creative functionality approach, observing that "there were innumerable ways" to write the declarations at issue.[216]

Justice Breyer, no stranger to computer copyright law,[217] took a different view. In discussing the nature of Oracle's declarations, he distinguished them sharply from the implementing code of subroutine instructions, making the assumption that the latter "is copyrightable but was not copied."[218] Specifically, said Justice Breyer, the declarations were "inherently bound together with uncopyrightable ideas," and their

---

208 *See* Oracle Am., Inc. v. Google LLC, 886 F.3d 1179, 1185–86 (Fed. Cir. 2018). For completeness, the fair use doctrine permits acts that otherwise constitute infringement of a copyright when those acts satisfy a statutory and common law balancing test for whether the acts are "fair." 17 U.S.C. § 107.

209 *See* Google LLC v. Oracle Am., Inc., 593 U.S. 1, 19–20 (2021).

210 *Id.* at 20 ("We shall assume, but purely for argument's sake, that the entire Sun Java API falls within the definition of that which can be copyrighted.").

211 *See id.* at 45–49 (Thomas, J., dissenting).

212 *Id.* at 46.

213 *Id.* at 46–48.

214 *Id.* at 48 (quoting RadLAX Gateway Hotel, LLC v. Amalgamated Bank, 566 U.S. 639, 645 (2012)).

215 *Id.*

216 *Id.*

217 As a law professor, Breyer wrote a now-famous criticism of copyright law, particularly as applied to software. *See* Breyer, *supra* note 39, at 341–43.

218 *Google*, 593 U.S. at 27.

value derived in large part from programmer goodwill divorced from copyright incentives.[219] As a result, the majority opinion deemed the declarations "further than are most computer programs (such as the implementing code) from the core of copyright."[220]

Justice Breyer had no occasion to explain what of the implementing code was copyrightable, but throughout the opinion remarked on how all computer code, including implementing code, is "functional in nature" and intended to "accomplish particular tasks."[221] Read in view of *Baker*, which the Court cited positively,[222] it would seem that a large part, if not all, of the implementing code might be uncopyrightable functionality. As such, Justice Breyer's opinion seemed to employ the unidentified-expression approach, with a shade of infinitesimality: Something in software is copyrightable, that something is in the implementing code, and whatever it is must be very thin given the functional nature of software generally.[223]

Like the many decisions that preceded them, the opinions in the Oracle–Google litigation leveraged all four approaches to resolving the software copyright trilemma. The irreconcilability of these approaches left the litigation protracted, uncertain, and hotly contested even to this day.[224]

D. Literature

Like the case law, scholarly literature on software copyright has struggled to resolve the conflict between copyrightable subject matter and explicit protection of computer code.

A notable early attempt at characterizing the copyrightable subject matter of code was by three legal practitioners, including an IBM attorney.[225] Recognizing the differing viewpoints on the relationship between software and copyright law, Anthony L. Clapes and his colleagues

---

219 *Id.* at 28–29 (observing that declarations' "value in significant part derives from the value that those who do not hold copyrights, namely, computer programmers, invest of their own time and effort to learn the API's system").

220 *Id.* at 29.

221 *Id.* at 28–30.

222 *See id.* at 26 ("No one claims that the decisions about what counts as a task are themselves copyrightable . . . ." (citing Baker v. Selden, 101 U.S. 99 (1880))).

223 *Cf. id.* at 21 ("[I]n some circumstances, . . . where copyrightable material is bound up with uncopyrightable material, copyright protection is 'thin.'" (quoting Feist Publ'ns, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340, 349 (1991))).

224 *See* Mark A. Lemley & Pamela Samuelson, *Interfaces and Interoperability After* Google v. Oracle, 100 Tex. L. Rev. 1, 2 (2021) ("But the decision not to address the larger question—whether interfaces are copyright-protectable at all—will produce uncertainty."); SAS Inst., Inc. v. World Programming Ltd., 64 F.4th 1319, 1331, 1335 (Fed. Cir. 2023) (divided opinion on software copyrightability).

225 *See* Clapes et al., *supra* note 41, at 1493 n.*.

sought to resolve the debate through an approach superficially similar to this Article—by reviewing specific elements of software for copyrightable expression.[226] The trio, however, premised their analysis on creativity even in operational elements, leading them to determine that expression in a computer program "includes the detailed design, structure, logic, and flow implicit in the source code."[227] Indeed, whether a program is "written to execute very rapidly in a processor" and whether it will "take up little memory," wrote the authors, are "indicia of individual style."[228] These elements affect the ordering of instructions and arrangement of memory,[229] which makes them methods of operation.[230]

Thus, Clapes and colleagues adopted the creative-functionality approach, as have many other commentators.[231] These authorities also tend to recite the abrogation argument, indicating both that they recognize the doctrinal tension and that they view Congress as having resolved the question by taking one side over the other.[232]

---

226 *See id.* at 1503, 1510 (performing "examination of the way programmers express themselves in programs").

227 *Id.* at 1573.

228 *Id.* at 1536.

229 *See, e.g.*, *id.* at 1528–30 (identifying size of memory blocks and order of instructions as creative elements).

230 *See supra* text accompanying notes 95–97.

231 *See, e.g.*, John M. Griem Jr., Note, *Against a Sui Generis System of Intellectual Property for Computer Software*, 22 HOFSTRA L. REV. 145, 153 (1993) ("While the programming code is a kind of 'useful expression,' unlike traditionally copyrightable works, it is properly the subject of copyright protection because it has an expressive aspect . . . ." (footnote omitted)); Jon S. Wilkins, Note, *Protecting Computer Programs as Compilations Under* Computer Associates v. Altai, 104 YALE L.J. 435, 453 (1994) ("In copyright terms, at the design phase programmers have wide discretion to 'select' particular algorithms, techniques, and structures; to 'arrange' these elements by creating a flow of program and data control; and to 'coordinate' these program parts into a working software whole."); Irwin R. Gross, *A New Framework for Software Protection: Distinguishing Between Interactive and Non-Interactive Aspects of Computer Programs*, 20 RUTGERS COMPUT. & TECH. L.J. 107, 146 (1994) (proposing that copyright inheres in "non-interactive elements" of code, including "orderings of a program's instructions"); Dennis M. Carleton, *A Behavior-Based Model for Determining Software Copyright Infringement*, 10 HIGH TECH. L.J. 405, 418 (1995) ("[P]rotectable expression . . . would include such things as how the user selects the function, how the program collects necessary information from the user, how the program behaves in case of an error, and so on . . . ."); Lisa C. Green, Note, *Copyright Protection and Computer Programs: Identifying Creative Expression in a Computer Program's Nonliteral Elements*, 3 FORDHAM INTELL. PROP. MEDIA & ENT. L.J. 89, 119 (1992) (asserting that organizational constructs of programs are, "at least, potentially protectable as expression" because they are not processes); Keplinger, *supra* note 117, at 485 (suggesting that a computer program, as a "writing which explains in intricate detail the procedure for carrying out a process, idea, or algorithm," is copyrightable expression).

232 *See* Clapes et al., *supra* note 41, at 1549 n.202 ("The argument that because computer programs are 'functional' they are entitled to little or no copyright protection . . . suffers from two quite fatal shortcomings. (1) Congress has decided otherwise."); Morton David Goldberg & John F. Burleigh, *Copyright Protection for Computer Programs: Is the Sky Falling?*, 17 AIPLA Q.J. 294, 320 (1989) ("Treating computer programs merely as a component of 'programmed computers' proves too much: it would nullify copyright protection for programs and render their treatment under

A few years later, a coalition of legal scholars and technologists published a major response to the creative-functionality line of reasoning, titled *A Manifesto Concerning the Legal Protection of Computer Programs* ("*Manifesto*").[233] The principal observation of the *Manifesto* was that computer code is operational, even in its expressive aspects: "Programs have almost no value to users as texts. Rather, their value lies in behavior."[234] Valuable behavior merited some sort of intellectual property protection, according to the authors, but copyright protection was the wrong form given that it "has traditionally excluded machines and technological processes from its domain."[235]

The *Manifesto* thus adopts the infinitesimality approach, finding uncopyrightable subject matter at "virtually all levels of abstraction above the literal text."[236] This more limited view also has wide reception in scholarship: Professor Samuelson, one of the authors of the *Manifesto*, has regularly held to it,[237] as have other commentators.[238] Still others have taken a more generous unidentified-expression view of

---

the Copyright Act mere surplusage."); Raymond T. Nimmer & Patricia Ann Krauthaus, *Software Copyright: Sliding Scales and Abstracted Expression*, 32 HOU. L. REV. 317, 335–36 (1995) ("[The] conclusion that substantially all of the valuable aspects of computer programs are unprotected processes or methods of operation except, perhaps, nonutilitarian images or clearly expressive text . . . contradicts the congressional mandate that computer programs come within copyright protection as literary works.").

233 *See* Samuelson et al., *supra* note 129, at 2311–12 & n.5.

234 *Id.* at 2319.

235 *Id.* at 2348 & n.147.

236 *Id.* at 2353.

237 *See, e.g.*, Pamela Samuelson, *Staking the Boundaries of Software Copyrights in the Shadow of Patents*, 71 FLA. L. REV. 243, 281–82 (2019) ("The 'thin' protection strategy . . . may be the surest way for courts to ensure that copyright is not being used to confer patent-like protection to software innovations."); Pamela Samuelson, *Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement*, 31 BERKELEY TECH. L.J. 1215, 1296 (2016) (proposing a limited role for software copyright infringement to prohibit "minor changes to program texts, such as changing variable names, rearranging instructions, or recompiling the code to disguise infringement"); *see also* Samuelson, *supra* note 132, at 749 ("To the extent that the utilitarian character of programs is now better understood, Congress should reevaluate either copyright protection for computer programs or its policy with respect to utility and promulgate a consistent rule.").

238 *See, e.g.*, Alan L. Durham, *Copyright and Information Theory: Toward an Alternative Model of "Authorship,"* 2004 BYU L. REV. 69, 72 n.18 (noting limited communicative role of computer software); Weinreb, *supra* note 16, at 1168 ("[A]s the program is developed, the expression is progressively eliminated, until the program is embodied in a computer and ceases to be 'expressed' altogether."); Dennis S. Karjala, *Copyright, Computer Software, and the New Protectionism*, 28 JURIMETRICS J. 33, 87–88 (1987) (proposing that copyright infringement for software be limited to "[s]lavish verbatim and near verbatim copying," or at least copies that "give rise to piracy concerns").

the copyrightable subject matter in software, but nevertheless have not identified those elements with particularity.[239]

Despite these opposing views on copyright law, the commentators generally agree that software is meant to be used, not read.[240] To the extent that there is recognition that computer code has value in its readability, reading of code is considered "a distinctly secondary purpose,"[241] and the readable elements are simultaneously the operational parts of the program.[242]

III. Communicative Elements of Code

It is possible, of course, that Congress simply wrote an internally inconsistent statute.[243] Indeed, a technology advisory body to Congress seemed to think as much, positing in 1992 that the "complicated" role of § 102(b) could be reconciled only through legislation.[244]

---

239 *See, e.g.*, Lemley, *supra* note 16, at 27 ("[T]he existence of software patents should make courts less willing to extend the coverage of copyright law to ideas and the functional elements of programs, and more willing to engage in a strict filtration analysis."); Peter S. Menell, *An Analysis of the Scope of Copyright Protection for Application Programs*, 41 Stan. L. Rev. 1045, 1082, 1086 (1989) (suggesting that "legal protection for application program code should not extend much, if at all, beyond protection against literal copying," but identifying potential copyrightable elements as those that are "inefficient or otherwise did not reflect good programming practice"); Marci A. Hamilton & Ted Sabety, *Computer Science Concepts in Copyright Cases: The Path to a Coherent Law*, 10 Harv. J.L. & Tech. 239, 278–79 (1997) (concluding that algorithms, necessary data structures, and computer language grammars are not copyrightable, but suggesting that other aspects might be). Steven R. Englund comes closest to identifying a specific copyrightable, nonoperational element, namely the grouping of instructions into subroutines. *See* Englund, *supra* note 41, at 903 ("If the same process could be implemented efficiently by combining and redividing the functions of a set of modules in a number of substantially different ways, then the selection of a particular set of modules should be protected."). However, a more detailed look at division into subroutines reveals that element to be operational too. *See infra* text accompanying notes 298–305.

240 *See, e.g.*, Weinreb, *supra* note 16, at 1153 ("In their operational form, programs are strictly functional and contain no expression; for all practical purposes, they are simply a sequence of steps in the operation of a machine."); Hamilton & Sabety, *supra* note 239, at 245 (noting "the vexing reality that computer programs are a form of expression not intended to be 'consumed' by a human").

241 Karjala, *supra* note 238, at 38; *see* Samuelson et al., *supra* note 129, at 2318–19; Menell, *supra* note 239, at 1056 ("Though some programmers have distinctive programming styles, what matters in the end (assuming that the assigned tasks have been accomplished) is the accuracy, efficiency, and reliability of the resulting program." (footnote omitted)); Nimmer & Krauthaus, *supra* note 232, at 331 ("But programs of this type have value based on functionality, rather than expressive flair."); Englund, *supra* note 41, at 893 ("Although they have some value simply as unambiguous descriptions of solutions to problems, programs usually are written and are important because they can be used to operate computers to achieve some result.").

242 *See* Clapes et al., *supra* note 41, at 1512 (observing that programs are read not for "bedtime reading," but so they can be "modified, enhanced, or corrected").

243 *But see* Goldberg & Burleigh, *supra* note 232, at 319–20, 320 n.109 (questioning this possibility).

244 Off. of Tech. Assessment, U.S. Cong., *supra* note 16, at 29–30.

But there is a satisfying theory of software copyright that fulfills both doctrines simultaneously—the communicative elements theory proposed by this Article. All that is needed is to identify elements or aspects of computer code that are expressive but definitively not methods of operation. Such elements do not only exist—they are pervasive in software.[245] Code is written not just to be executed by computers, but also to be read by other humans who want to learn from and understand software.[246] Thus, it contains numerous elements to facilitate that human communication.[247] To be sure, even the most utilitarian elements of design also communicate information—a wavy metal structure, for example, communicates aesthetic attractiveness while simultaneously functioning as a bicycle rack.[248] So too can operational aspects of code be communicative.[249] Shown below, instead, are many elements that are *solely* communicative because: (1) They are not methods of operation as defined per *Baker*, meaning that they do not affect the program's outputs, order of operations, or memory content;[250] and (2) they are intended to affect the understanding or experience of human readers.

This Part catalogs several of these "communicative elements" of code. It is not exhaustive,[251] but sufficient to overcome the assumption

---

245 *See, e.g.*, Yield Dynamics, Inc. v. TEA Sys. Corp., No. c 06-0331, 2007 U.S. Dist. LEXIS 113956, at *10–12 (N.D. Cal. Feb. 21, 2007) (observing that defendants copied "order, comments, variable names, and even spacing, line breaks and capitalization"). The briefing in *Yield Dynamics* is remarkably consistent in many respects with the elements identified in this Article, although (for obvious length reasons) the briefs do not fully demonstrate the non-operationality of each element. *See* Plaintiff Yield Dynamics, Inc.'s Notice of Motion and Motion for Partial Summary Judgment at 9, Yield Dynamics, Inc. v. TEA Sys. Corp., No. c 06-0331, 2007 U.S. Dist. LEXIS 113956 (N.D. Cal. Feb. 21, 2007).

246 *See* Clapes et al., *supra* note 41, at 1512.

247 *Id.*

248 *See* Brandir Int'l, Inc. v. Cascade Pac. Lumber Co., 834 F.2d 1142, 1146–47 (2d Cir. 1987).

249 *See supra* text accompanying notes 125–26.

250 *See supra* text accompanying notes 95–97. These communicative elements may affect the speed with which a given processor can execute a given instruction. *See* Dragan Rapić, *Your JavaScript Is Slowing Down Because of… Comments?!*, Medium: Level Up Coding (Aug. 15, 2025), https://levelup.gitconnected.com/your-javascript-is-slowing-down-because-of-comments-f9766a802d61 [https://perma.cc/S5U6-9ZEP]. A jump to a memory address further away, for example, may take more time depending on how the memory is arranged. *See id.* It is difficult to say that such effects, which are often unpredictable, are methods of operating the computer; they are better understood as how well the computer works. *See id.* These elements may also affect the performance or efficiency of translation from source code into machine-executable instructions. *See id.* Any such effects during the translation process do not affect how the computer operates once the translation is done; in any event those effects are inconsequentially minimal. *See id.*; *see, e.g.*, Response by u/captainAwesomePants to question posted by u/Kenghis_Ghan, Reddit (r/LearnProgramming), *Do comments affect compile time at all?* (July 16, 2019), https://www.reddit.com/r/learnprogramming/comments/ce01zj/do_comments_affect_compile_time_at_all/ [https://perma.cc/F87F-DAU9].

251 Furthermore, these elements may not be communicative and nonoperational in all situations. Programming languages and compilers change over time and may optimize for language

that "computer programs are by definition functional,"[252] and indeed to show that nonoperational communicative elements are the rule rather than the exception. It also distinguishes purely communicative elements from examples of simultaneously expressive and operational elements,[253] and draws from the computer science literature to show that purely communicative elements are significant to programmers in ways that serve the purposes of copyright protection.[254]

A. Comments

In the following:

```
/**
 * Set the user's name
 */
void setName(String name) {
   . . .
}
```

the text "Set the user's name" is a *comment*, a narrative explanatory text delimited by the slashes and asterisks.[255] Comments are discarded during translation into machine code, typically by the compiler.[256] Therefore, they have no operational effect and serve solely to communicate explanations to human readers.

Comments are clearly copyrightable matter and have long been recognized as such.[257] But they are unsatisfying as a complete basis for copyright protection in software, both because they are easily removed even by an intentional copyist,[258] and because, being ignored during compilation, comments provide no basis for copyright protection in object code.

---

constructs previously treated as equivalent. *See, e.g.*, *infra* note 280. This may seem odd—why should the copyrightability of computer code depend on system optimization?—but in fact it makes sense. If improving a compiler or computer causes it to differentiate between different programming constructs, then the compiler or hardware has extracted useful functionality out of that differentiation, making the constructs no longer purely expressive.

252 *Oracle II*, 750 F.3d 1339, 1367 (Fed. Cir. 2014).

253 *See infra* Section III.F.

254 *See infra* Section III.G.

255 *See, e.g.*, KERNIGHAN & RITCHIE, *supra* note 20, at 12.

256 *Id.* at 13.

257 *See, e.g.*, Clapes et al., *supra* note 41, at 1534–35.

258 There are automatic tools for stripping comments from source code. *See, e.g.*, Albert Danial, *cloc: Count Lines of Code*, GITHUB (June 24, 2025), https://github.com/AlDanial/cloc [https://perma.cc/8UVH-PVL5] (describing and providing download instructions and files for a tool that will strip comments from source code).

B. Whitespace

In many programming languages, the programmer has a great deal of flexibility in spacing within code, so long as individual words are separable: One space, multiple spaces, and carriage returns are all processed identically upon execution.[259] The following code examples are all operationally equivalent:

1.
```
while(true){print(1);}
```

2.
```
while (true) { print(1); }
```

3.
```
while (true) {
     print(1);
}
```

4.
```
while(true)
{
     print(1);
}
```

Programming languages permit this flexibility in whitespace purely to facilitate readability.[260] Spacing allows programmers to align related elements of the code vertically, group related blocks of code with indentation, space related sets of instructions into "stanzas," and otherwise make code easy to scan and read.[261] Conversely, stinginess in spacing may help obscure the meaning of code, which can serve other aesthetic purposes such as making the programmer appear to be cryptically proficient, or "l33t" as they somewhat derisively call themselves.[262]

C. Bound Symbols

Consider the following subroutine definition[263]:

```
int cylinder_volume(int a, int b) {
    . . .
}
```

---

259 *See, e.g.*, KERNIGHAN & RITCHIE, *supra* note 20, at 168.

260 *See id.* at 13.

261 *See, e.g.*, *id.* ("Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read."); ROBERT C. MARTIN, CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP 86–87 (1st ed. 2009).

262 *See infra* text accompanying notes 349–54.

263 Yes, it is true that for this and all subsequent examples that integer values are being used where floating-point numbers—i.e., number variables that allow decimals, KERNIGHAN & RITCHIE, *supra* note 20, at 13—would make more sense. But *int* was explained earlier, *see supra* text accompanying note 62, and you're learning enough computer science already.

The subroutine takes two input arguments and presumably computes the volume of a cylinder based on them. But how is the subroutine to be invoked, say to compute the volume of a cylinder 8 units high and 12 units in diameter? Should the 8 go first, or the 12? Consider this version instead:

```
int cylinder_volume(int height, int radius) {
    . . .
}
```

Now it is clear that 8 should be the first input, and that the diameter of 12 should be halved, to convert it to a radius.

These names *a*, *b*, *height*, and *radius* are all names for variables.[264] In particular, they are *bound variables*, because their names are entirely internal—that is, bound—to the subroutine instructions.[265] Here is the remaining definition of the cylinder volume subroutine:

```
int cylinder_volume(int height, int radius) {
    return height * pi * radius^2;
}
```

The variables *height* and *radius* are bound because they are fully intrinsic to the subroutine.[266] But *pi* is unbound, or "free," at least with respect to this subroutine,[267] and must be defined somewhere else in the code, presumably as the particular transcendental number[268] that begins with 3.14159.[269]

Bound variables are important because "[t]he names of bound variables are immaterial"—they serve no operational purpose.[270] In many

264 *See supra* text accompanying notes 56–62.

265 *See* Robert Harper, Practical Foundations for Programming Languages 6–7 (2013). They are also called "local" variables because the "scope" in which the variable name is usable is local to the subroutine. *See* Kernighan & Ritchie, *supra* note 20, at 72–73; Gosling et al., *supra* note 56, at 265–69.

266 *See* Harper, *supra* note 265, at 8–9.

267 *See id.*

268 *See* David S. Richeson, *How Math Achieved Transcendence*, Quanta Mag.: Quantized Columns (June 27, 2023), https://www.quantamagazine.org/recounting-the-history-of-maths-transcendental-numbers-20230627/ [https://perma.cc/KR5P-62V3] (explaining the definition and history of transcendental numbers).

269 *See* W. Jones, A New Introduction to the Mathematics 243 (London, Jeff. Wale 1706); 1 Leonhard Euler, Introductio in Analysin Infinitorum 93 (Lausanne, Marc Michel Bousquet & Assocs. 1748); Steven Bogart, *What Is Pi, and How Did It Originate?*, Sci. Am. (May 17, 1999), https://www.scientificamerican.com/article/what-is-pi-and-how-did-it-originate/ [https://perma.cc/P5DJ-E57Q].

270 Harper, *supra* note 265, at 7.

programming languages, there is no way to refer to the bound variables by name outside the block of code where it is used.[271] Indeed, compilers will generally convert bound variables to numerical memory addresses and discard their names.[272] Thus, a programmer can freely change bound variable names without changing the operation of the subroutine. The following two subroutines:

- ```
  int cylinder_volume(int a, int b) {
      return a * pi * b^2;
  }
  ```
- ```
  int cylinder_volume(int airplane, int jelly) {
      return jelly * pi * airplane^2;
  }
  ```

are operationally identical to each other and the earlier example. Yet the difference in human communication is obvious: *height* and *radius* are self-explanatory; these others are confusing at best.[273]

D. Syntactic Sugar

Programming languages often offer a variety of ways to express the same operation, to satisfy a variety of programming styles. These stylistic variants are often called "syntactic sugar" because they simplify programming statements that are frequently used or awkwardly verbose.[274] Yet it is up to the programmer whether to use them. In some cases, the alternate form may seem too terse, so a programmer might choose the more verbose form to enhance readability for program-reading audiences.

In the C language, for example, one can increment a variable's value with $x = x + 1$, or equivalently with $x \mathrel{+}= 1$, or sometimes even $++x$.[275] Perl, famously the "There's More Than One Way To Do It" language, allows for conditional clauses to be written at the beginning or end of a statement,[276] making the following three lines operationally identical:

---

271 *See, e.g.*, GOSLING ET AL., *supra* note 56, at 81–82, 267–68 (describing "scope" of variable names, namely "the region of the program within which the entity declared by the declaration can be referred to using a simple name").

272 *See supra* text accompanying notes 56–62; APPEL & GINSBURG, *supra* note 61, at 158.

273 *See* GOSLING ET AL., *supra* note 56, at 110–11 (providing recommendations on choices of variable names).

274 *See, e.g.*, Donghoon Kim & Gangman Yi, *Measuring Syntactic Sugar Usage in Programming Languages: An Empirical Study of C# and Java Projects*, *in* ADVANCES IN COMPUTER SCIENCE AND ITS APPLICATIONS 279, 279–80 (Hwa-Young Jeong et al. eds., 2014).

275 *See* KERNIGHAN & RITCHIE, *supra* note 20, at 44–45, 47–48.

276 *See* LARRY WALL, TOM CHRISTIANSEN & RANDAL L. SCHWARTZ, PROGRAMMING PERL 10 & n.*, 96–97 (Steve Talbott ed., 2d ed. 1996).

- `if ($x != 0) { return $y; }`
- `return $y if $x != 0;`
- `return $y unless $x == 0;`

For a fuller example, consider a program for controlling a car based on the color of an observed signal light. In C and Java,[277] this could be written as:

```
if (lightColor == RED) {
    stopCar();
} else {
    if (lightColor == YELLOW) {
        slowDownCar();
    } else {
        if (lightColor == GREEN) {
            driveCar();
        }
    }
}
```

Alternatively, the curly braces following if and else can be omitted:

```
if      (lightColor == RED)    stopCar();
else if (lightColor == YELLOW) slowDownCar();
else if (lightColor == GREEN)  driveCar();
```

Note the white space for vertical alignment. These code snippets operate identically, but they have different readability characteristics. The construct without braces is compact and quickly informative; the alternative is more verbose but less error prone.[278]

The Python language offers a particularly striking example of syntactic sugar.[279] Consider the problem of taking a list of "strings," namely sequences of letters forming a phrase, and extracting all the uppercase letters individually into a new list:

```
output_list = []
for string in string_list:
   for letter in string:
      if letter.isupper():
         output_list.append(letter)
```

277 *See* KERNIGHAN & RITCHIE, *supra* note 20, at 52–53; GOSLING ET AL., *supra* note 56, at 269–70 (describing "dangling else" problem).

278 Why? If one wants to add a second statement within a given condition, the braces are mandatory; the second construct with no braces invites a hasty editor to forget this rule. *See* KERNIGHAN & RITCHIE, *supra* note 20, at 53.

279 *See* MARK LUTZ, LEARNING PYTHON 229–33 (Julie Steele ed., 4th ed. 2009).

This code processes each string in *string_list* using the *for* command in the first line. For each such string, the code considers each of its letters with the second *for* command. It then appends any letters matching *isupper()*, a subroutine that tests whether a letter is uppercase, to an output list.

The "double-for" construction is verbose, requiring five lines of code. The construction below is identical in function[280]:

```
output_list = [ l for s in string_list
                for l in s if l.isupper() ]
```

Note the operationally irrelevant changes in bound variable names—*s* for string and *l* for letter.[281]

This compact syntax is called a "list comprehension,"[282] and it is debatable which form is easier to understand. The longer form certainly lays out its operations in more detail, but it buries its purpose—collecting capital letters—inside three layers of indentation. The list comprehension is perhaps cryptically terse, but it makes clear that its purpose is to construct a list, as indicated by the square brackets, and it puts up front the object *l* that will populate the list. Programmers regularly debate whether and when list comprehensions are preferred from a readability perspective,[283] and whether compactness of form leads to programming errors.[284]

---

280 *See id.* The equivalence between list comprehensions and *for* loops appears to have been true as of Python version 2.0. *See* A.M. Kuchling & Moshe Zadka, *What's New in Python 2.0*, Python.org (Mar. 21, 2026, at 18:21 UTC), https://docs.python.org/3/whatsnew/2.0.html [https://perma.cc/EG8S-32N2] (explaining that "a list comprehension is equivalent to the following [*for*-loop] Python code"); Greg Ewing, *Python List Comprehensions Enhancement*, Univ. Canterbury (Sep. 1999), https://www.csse.canterbury.ac.nz/greg.ewing/python/listcomp/index.html [https://perma.cc/LJ4B-QBLE] ("Code [translating a list comprehension] is generated almost exactly as if the above transformation [to a *for* loop] had been carried out on the source."). In more recent versions of Python, list comprehensions have been optimized for performance, so they may no longer be operationally equivalent. *See, e.g.*, Carl Meyer, *PEP 709 – Inlined Comprehensions*, Python.org (Dec. 15, 2023, at 15:06 UTC), https://peps.python.org/pep-0709/ [https://perma.cc/FGE3-ZZYM].

281 *See supra* Section III.C.

282 *See* Lutz, *supra* note 279, at 229; Luis Diego Fallas, *List Comprehensions Across Languages*, Exploring Beautiful Languages (Feb. 18, 2007, at 21:10 ET), http://langexplr.blogspot.com/2007/02/list-comprehensions-across-languages_18.html [https://perma.cc/8CPN-EQH3].

283 *Compare* Peter Grant, *List Comprehension in Python*, Built In (June 2, 2025), https://builtin.com/data-science/list-comprehensions-python [https://perma.cc/EJH4-PBPB] ("List comprehensions are a standard Python tool you can use to make your code simpler to read and easier for your colleagues to understand."), *with* Trey Hunner, *Overusing List Comprehensions and Generator Expressions in Python*, TreyHunner.com (Mar. 26, 2019, at 13:30 ET), https://treyhunner.com/2019/03/abusing-and-overusing-list-comprehensions-in-python/ [https://perma.cc/5A5U-3J8J] (identifying "cases when comprehensions aren't the best tool for the job, at least in terms of readability").

284 *See* Fiorella Zampetti, François Belias, Cyrine Zid, Giuliano Antoniol & Massimiliano Di Penta, *An Empirical Study on the Fault-Inducing Effect of Functional Constructs in Python*, 2022

E. Order of Variable Declarations

Older programming languages required variables to be "declared" before use.[285] Consider, for example, code to compute the current dew point, a value that depends on temperature and humidity. The code might look, in part, like this:

```
int getDewPoint(int temp, int humidity) {
    int b, c, g, d;
    b = 18;
    c = 243;
    g = log(humidity) +
       (b * temp) / (c + temp);
    d = c * g / (b - g);
    return d;
}
```

The arguments *temp* and *humidity* are declared as part of the subroutine declaration, but there are four more bound variables—*b*, *c*, *g*, and *d*—used for internal purposes. The initial line, *int b, c, g, d*, is required in some languages to advise the compiler on how many bound variables will need memory addresses.[286] But as a readability matter, it is awkward to place the declarations so far away from their use—*d* remains a mystery for four code lines between its declaration and use. Thus, modern programming languages allow placing variable declarations more logically[287]:

```
int getDewPoint() {
    int b, c;
    b = 18;
    c = 243;
    int g, d;
    g = log(humidity) +
      (b * temp) / (c + temp);
    d = c * g / (b - g);
    return d;
}
```

IEEE INT'L CONF. ON SOFTWARE MAINT. & EVOLUTION 47, 56 (finding that list comprehensions used in open-source software projects correlated with higher incidences of bug fixes).

285 *See, e.g.*, KERNIGHAN & RITCHIE, *supra* note 20, at 200–01 (requiring that declarations precede statements in a block).

286 *See id.* at 39.

287 *See* GOSLING ET AL., *supra* note 56, at 266 ("Local variable declaration statements may be intermixed freely with other kinds of statements in the block.").

There is no operational difference in either case; the computer assigns memory addresses to all four variables: *b*, *c*, *g*, and *d*.[288] The difference is a matter of explanatory power—in this case, grouping like information in like places.[289]

F. Order of Subroutine Declarations

In the same way that a programmer can choose the positions of variable declarations within a function, a programmer can often also choose the order in which subroutines are defined, without an effect on the operation of the program.[290] Consider, for example, a program that includes three subroutines for computing the area of circles, rectangles, and triangles, respectively:

```
int circleArea(int radius) {
    return pi * radius * radius;
}
int rectangleArea(int width, int length) {
    return width * length;
}
int triangleArea(int width, int height) {
    return rectangleArea(width, height) / 2;
}
```

In Java, programmers can define these three subroutines in any order and intersperse them with declarations of variables or other matter.[291]

Programmers can use this flexibility to arrange subroutine definitions according to taste. Sun Microsystems, the original creator of Java, recommended declaring variables first, followed by subroutines arranged topically.[292] Others recommend placing "high level" functions first, followed by "low level" functions that the former depend upon.[293] Google's style guide is especially indicative of the role of ordering subroutines:

---

288 *See id.* at 266–69.

289 *See* MARTIN, *supra* note 261, at 80 ("Variables should be declared as close to their usage as possible.").

290 *See* GOSLING ET AL., *supra* note 56, at 301–10.

291 *See id.* at 2 ("Java does not require types or their members to be declared before they are used. Declaration order is significant only for local variables and the order of initializers of fields in a class or interface.").

292 *See* SUN MICROSYSTEMS, INC., JAVA CODE CONVENTIONS § 3 (1997), https://www.oracle.com/technetwork/java/codeconventions-150003.pdf [https://perma.cc/C837-SS5J].

293 *E.g.*, MARTIN, *supra* note 261, at 84–85.

> The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways.
>
> What is important is that each class uses *some* logical order, which its maintainer could explain if asked.[294]

In older programming languages like C, a subroutine cannot be used before it is defined: *triangleArea* cannot precede *rectangleArea* because the former uses the latter.[295] To get around this limitation, C allows the programmer to predeclare subroutines with "prototypes"[296]:

```
int circleArea(int);
int rectangleArea(int, int);
int triangleArea(int, int);
int triangleArea(int width, int height) {
    return rectangleArea(width, height) / 2;
}
int rectangleArea(int width, int length) {
    return width * length;
}
int circleArea(int radius) {
    return pi * radius * radius;
}
```

Despite originating from a compiler limitation, prototypes give programmers twice the flexibility, as the prototypes and the subroutine definitions can be arranged in different orders. One might decide, for example, that prototypes should be arranged according to how commonly each will be used, but that definitions should be arranged in order of complexity.[297]

---

294 *Google Java Style Guide* § 3.4.2, GitHub (emphasis in bold omitted), https://google.github.io/styleguide/javaguide.html [https://perma.cc/4JVS-XZSE] (last visited Mar. 21, 2026).

295 *See* Kernighan & Ritchie, *supra* note 20, at 72–73; Martin, *supra* note 261, at 84 & n.2.

296 *See* Kernighan & Ritchie, *supra* note 20, at 27. Notice that the bound variable names may be omitted, further indicating their operational irrelevance. *See id.*

297 *Cf.* Jerry Doland & Jon Valett, Nat'l Aeronautics & Space Admin., C Style Guide 15, 28–29 (1994), https://ntrs.nasa.gov/api/citations/19950022400/downloads/19950022400.pdf [https://perma.cc/R2LM-KAGY] (providing different recommendations for organizing header files containing prototypes and program files containing subroutine definitions).

G. Negative Examples

Although the previous examples have illustrated elements of computer code that are communicative, it is worth noting several elements that are *not*, or at least debatably not, communicative. These elements all do serve communicative purposes, but nevertheless also affect the computer's operation. They are described below both to delineate purely communicative elements sharply, and to explain how prior analyses have not been rigorous about the distinction.

1. Breakdown of Subroutines

As described previously, a programmer can group chunks of instructions into subroutines.[298] At least one commentator has argued that the programmer's decisions on how to divide the instructions of a task into subroutines is copyrightable "structure," because the processor will ultimately execute all of the same instructions regardless of how they are grouped into subroutines.[299] This idea is intuitively appealing, because a programmer could bundle a sequence of instructions into one subroutine or break them into several; in either case the processor would perform all the instructions.[300]

But when a subroutine is invoked, the processor performs memory rearrangements and jump instructions to navigate from invocation to subroutine.[301] Thus, at the level of the computer processor, the choice of whether to separate a set of instructions into one subroutine or several changes the sequence of instructions performed, often in ways that substantially affect efficiency.[302] Arrangement of code into subroutines, then, has operational effect and is not purely communicative.[303]

This analysis of CPU instructions is certainly technically intricate, but it demonstrates an important point. Subroutine structure has a

---

298 *See supra* text accompanying notes 64–73.

299 *See* Englund, *supra* note 41, at 902–03. It is assumed that the names of the subroutines are inaccessible to third-party programmers, or else those names would be operational elements for reasons described a few paragraphs below. *See infra* text accompanying notes 306–13.

300 *See* Englund, *supra* note 41, at 903.

301 *See supra* text accompanying notes 64–73.

302 In fact, programmers sometimes explicitly try to avoid the processing overhead of subroutine calls, for example by use a coding technique called "loop unrolling" in which instructions in a subroutine are repeated multiple times. *See, e.g.*, João M.P. Cardoso, José Gabriel F. Coutinho & Pedro C. Diniz, Embedded Computing for High Performance 151–52 (2017).

303 By comparison, C also supports preprocessor macro substitution, in which subroutine-like instructions are spliced directly into other code by the compiler. *See* Kernighan & Ritchie, *supra* note 20, at 206–09. Because macro substitution and bodily inclusion do in fact produce the same compiled machine code, *see id.* at 207, structure based on preprocessor macros may be purely communicative.

small but detectable effect on a computer's operation,[304] so premising copyright protection on subroutine breakdown requires at least a small concession to copyright protection in methods of operation. Once that concession is made, it is unclear how far it extends—that, of course, is the origin of the creative-functionality approach.[305] The communicative elements described previously, however, appear to require no such concession at all, even at the level of the machine code that operates the processor.

2. Subroutine Names

For similar reasons, subroutine names are not communicative elements. Certainly, these names can convey a great deal of information to readers of code: *playChess* would be a terribly confusing name for a subroutine that computed digits of pi. But subroutine names also affect the operation of computers, at least to the extent that the names of subroutines in a program allow other third-party programs to use those same names to invoke the subroutine.[306]

Say that program *A* includes a more logically named subroutine, *calculatePi*, and program *B* invokes it using that same name. *B*, as a user of *A*'s subroutine, can be understood as an input of *A*, and the expected output of *A* with input *B* is the computed digits of pi.[307] If program *A*'s author were to rename the subroutine *makePiDigits*, then running *A* in conjunction with *B* would now generate an error at the point that *B* invokes the now-nonexistent *calculatePi*. *A* produces different outputs for a given input and performs a different order of instructions, so the subroutine name is a method of operation as defined based on *Baker*.[308]

Certainly, if the subroutine name were changed in both *A* and *B*, then the resulting computer program output would be the same. Nevertheless, the name is operational for two reasons, one technical and one doctrinal. The technical answer is that inside *A*'s executable machine code file is a *symbol table* containing the names of all of its subroutines,

---

304 *See, e.g.*, *id.* at 65–67 (explaining how the structure of declarations in relation to the result returned by a section of code may cause a loss of data due to a change in the variable type).

305 *See supra* text accompanying notes 125–26.

306 *See, e.g.*, KERNIGHAN & RITCHIE, *supra* note 20, at 62, 203–06 (noting possibility of "previously compiled functions from libraries" and describing function definitions as a type of "external declaration").

307 It may seem odd to think of program *B* as an "input," but the idea that computer instructions are also a form of input data is foundational to computer science, *see, e.g.*, A.M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, 42 PROCS. LOND. MATHEMATICAL SOC'Y 230, 240 (1937) (constructing the "description number" for a computing machine (emphasis omitted)), and in any event is the logical consequence of machine code instructions being numbers and thus data. *See supra* text accompanying notes 47–52.

308 *See supra* text accompanying notes 95–97.

which external programs like *B* use to invoke those subroutines.[309] Accordingly, even at the object-code level, subroutine names determine the computer's operation. The doctrinal answer is that copyright protection inheres in a "work."[310] *A* and *B* are not the same "work," so the operationally relevant and thus uncopyrightable elements of *A* should be defined with respect to *A* alone, regardless of what changes *B*'s author might or might not make.[311]

This discussion of subroutine names clarifies the longstanding debate on the intersection of copyright and software compatibility.[312] At a high level, compatibility means using the same names and identifiers as another computer program; as a result, there can be questions about whether those duplicated names and identifiers are creative choices or required functionality.[313] But when it comes to operating a computer processor, compatibility is a question of whether the symbol table lookups in an executable program succeed or fail, showing that compatible names are methods of operation.

---

309 *See* TIS COMM., *supra* note 52, at 1-18; WALL ET AL., *supra* note 276, at 197–98; TIM LINDHOLM, FRANK YELLIN, GILAD BRACHA, ALEX BUCKLEY & DANIEL SMITH, THE JAVA® VIRTUAL MACHINE SPECIFICATION 357–60 (Java SE 14 ed. 2020), https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf [https://perma.cc/492G-G5GZ]. The process of *B* looking up subroutines by name in *A*'s symbol table is called *dynamic linking*, *see* TIS COMM., *supra* note 52, at A-10 to A-11, which may be familiar to Windows computer users who have dealt with Dynamic-Link Libraries, also known as DLL files.

Why is the symbol table necessary—can't *B* just store and then jump to the numerical memory address of the relevant subroutine in *A*? Because at the time that *B* is compiled, the memory locations of subroutines in *A* may be unknown, or it may change with different versions of *A*. So, the compiler constructs *B*'s object code to look up the subroutine by name, not address. *See supra* notes 61, 67 and accompanying text.

310 17 U.S.C. § 102(a); *see* Jani McCutcheon, *Works of Fiction: The Misconception of Literary Characters as Copyright Works*, 66 J. COPYRIGHT SOC'Y U.S.A. 115, 129–32 (2018) (discussing and citing references on meaning of "work" under the Copyright Act).

311 *See* 17 U.S.C. § 101 (defining "joint work").

312 *See, e.g.*, Donald S. Chisum et al., *LaST Frontier Conference Report on Copyright Protection of Computer Software*, 30 JURIMETRICS J. 15, 21–22 (1989).

313 *Compare Oracle II*, 750 F.3d 1339, 1371 (Fed. Cir. 2014) (holding Oracle's Java SE subroutine declarations copyrightable despite Google's argument that its use of the declarations was essential for interoperability with existing computer programs written in Java), *with* Chisum et al., *supra* note 312, at 22 (arguing that functional compatibility is "a legitimate goal for software competitors" and thus "reproduction of . . . program aspects or features . . . for the purpose of creating a compatible interface does not infringe"). *See generally* Charles Duan, *Internet of Infringing Things: The Effect of Computer Interface Copyrights on Technology Standards*, 45 RUTGERS COMPUT. & TECH. L.J. 1 (2019) (analyzing the copyrightability of interfaces and the possibility of infringement when creating compatible interfaces).

3. Arrangement of Data Structures

Data structures are collections of information stored within a computer in a structured format, typically as a list.[314] For example, the programmer of medical device software might devise a data structure for a patient's vital information:

```
structure PatientData {
    char[255] name;
    int height;
    int weight;
}
```

The structure acts somewhat like a blank form, with spaces for the name, height, and weight records. It can be stored in memory, passed around to various parts of the program, saved to a disk, or even transmitted across a network so another computer can use it.[315]

A programmer designing a data structure chooses which records to include and—in some cases—what order to store them.[316] These choices certainly can be expressive and communicative.[317] But data structure arrangement is also operational, because computer programs that use those data structures will produce different outputs if the structures are rearranged.[318] The arrangement of data structures is thus part and parcel of the operation of the computer program, in the same way that the peculiar lines and spaces on a blank form can be integral to the operation of a bookkeeping system.[319]

Although the relative arrangement of data structures is a method of operation, the exact, absolute positioning of data in memory may not be. Multiple programs running on a single computer cannot expect to work off specific memory addresses, because they would overwrite each other's data.[320] As such, computer processors, in conjunction with operating system software, often rearrange chunks of memory while programs are running, recomputing and translating memory addresses accordingly.[321] This helps to explain why the arrangement of subroutines

---

314 *See* MATTHEWS ET AL., *supra* note 42, at 38–46.

315 *See id.*

316 *See* Hamilton & Sabety, *supra* note 239, at 257–58.

317 *See id.* at 260.

318 Insofar as a file format is just a very complex data structure, anyone who has wrestled with incompatible or corrupted files is familiar with the operational consequences of data structure arrangements.

319 *See* Hamilton & Sabety, *supra* note 239, at 261 (arguing that "the structural aspect of data structures . . . that are necessary to particular algorithms should not be granted copyright protection").

320 *See* ANDREW S. TANENBAUM & HERBERT BOS, MODERN OPERATING SYSTEMS 41 (4th ed. 2015).

321 *See id.* at 194–95 (describing this process as "virtual memory").

in a program is not operationally significant—the operating system's automatic memory management routines might end up rearranging them.

H. Importance to Programmers

Communicative elements of code properly navigate the § 102(b) copyrightability limitation while nevertheless comporting with legislative intent to permit copyrighting software.[322] But they do more—in the ways programmers use them, the communicative elements align with the purposes, policies, and doctrines of copyright law more generally.

Copyright protection exists to "promote the [p]rogress of [s]cience and useful [a]rts,"[323] and it "must ultimately serve the cause of promoting broad public availability of literature, music, and the other arts."[324] This overarching public goal gives rise to copyright law's chief doctrine of originality: A copyrightable work is one that is "independently created" by an author, and that "possesses at least some minimal degree of creativity."[325] The "minimal degree of creativity," though not precisely defined in case law, appears closely tied to human will and idiosyncratic personality, explaining why preexisting facts are not creative.[326] Scholars have further imputed a requirement that a copyrighted work have an author who "is a human being who intends to produce one or more mental effects in an audience."[327]

Programmers regularly describe all of the aforementioned communicative elements as important mechanisms for making code readable.[328] This shows that these elements are used to produce intended mental effects in an audience of other programmers. Furthermore, there is no single best way to make code readable, and programmers regularly debate and express strong opinions on their preferred usages of these

---

322 *See supra* text accompanying notes 256–97.

323 U.S. Const. art. 1, § 8, cl. 8.

324 Fogerty v. Fantasy, Inc., 510 U.S. 517, 526 (1994) (quoting Twentieth Century Music Corp. v. Aiken, 422 U.S. 151, 156 (1975)).

325 Feist Publ'ns, Inc. v. Rural Tel. Serv. Co., 499 U.S. 340, 345 (1991).

326 *See id.* at 347–48 (discussing copyright as limited to the "original intellectual conceptions of the author" (quoting Burrow-Giles Lithographic Co. v. Sarony, 111 U.S. 53, 58 (1884))); Annemarie Bridy, *Coding Creativity: Copyright and the Artificially Intelligent Author*, 2012 Stan. Tech. L. Rev. 5, 6–8.

327 Christopher Buccafusco, *A Theory of Copyright Authorship*, 102 Va. L. Rev. 1229, 1260 (2016); *see also id.* at 1266–67 (characterizing copyright as "subject matter involving a relation between persons" (quoting Abraham Drassinower, What's Wrong with Copying? 65 (2015))).

328 *See supra* text accompanying note 20; Kernighan & Ritchie, *supra* note 20, at 22, 35; Martin, *supra* note 261, at 82, 88; Gosling et al., *supra* note 56, at 106; Kim & Yi, *supra* note 274, at 279; Wall et al., *supra* note 276, at 38, 548; Grant, *supra* note 283; Hunner, *supra* note 283; Zampetti et al., *supra* note 284, at 55.

elements.[329] This debate suggests that the usage of these elements is individual to, and expressive of, a programmer's personality, akin to the creativity and originality that copyright protection requires.[330] And insofar as communicative elements enable programmers to make their work more readable and understandable to others, those elements advance the public-access rationale that underlies copyright.

The earlier discussion of individual communicative elements provided some initial discussion of their importance to programmers' authorial individuality and expression. Further evidence is given below.

1. Early Examples

From the earliest days of computer programming, leaders in the field have valued readability and communicative elements in computer code. Brian Kernighan, coauthor of a definitive text on the C language, also coauthored a 1974 book titled *The Elements of Programming Style*, intended as a riff on Strunk and White's writing guide, *The Elements of Style*.[331] After noting widespread recognition of "the importance of readable programs," the book identifies such elements as disambiguating parentheses, variable names, and indentation for improving readability.[332] Leading computer scientist Donald Knuth, in 1984, proposed a paradigm shift in programming: "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do."[333] In Knuth's "literate programming" system, a programmer intertwined prosaic commentary and computer code together, and then used a program called WEB that would automatically rearrange the parts into both formatted documentation and executable source code.[334] Knuth's system thus made extensive use of the ability to arrange subroutines (and other parts of code) to one's tastes—top-down, bottom-up, stream of consciousness, or whatever else the programmer deems "psychologically correct."[335]

---

329 *See, e.g.*, *supra* text accompanying notes 283–84.

330 *Cf.* Spector, *supra* note 39, at 88–89 (suggesting that the need for creativity in developing clever interfaces "may be a type of artistic creativity").

331 *See* Brian W. Kernighan & P.J. Plauger, The Elements of Programming Style (2d ed. 1978); William Strunk Jr. & E.B. White, The Elements of Style, at vii (Macmillan Paperbacks ed. 1962).

332 *See* Kernighan & Plauger, *supra* note 331, at ix, 15, 43.

333 Donald E. Knuth, *Literate Programming*, 27 Comput. J. 97, 97 (1984).

334 *See id.* at 98.

335 *Id.* at 107.

2. GoTo *Considered Expressive*

And then there was an infamous debate—no less vigorous than writers' debates around the Oxford comma—over a syntactic sugar construction. Because a common part of computer program operation is to go to a different part of the machine code, many programming languages offer an instruction for doing so, typically called *goto*.[336] Alternatively, one can define "blocks" of instructions, perhaps to be repeated or skipped.[337] The latter can generally substitute for the former,[338] and either will ultimately be translated into machine-code jump or branch instructions.[339]

Choosing between *goto* and block-based flow control is thus a matter of readability and expression, and not necessarily of operation. In 1968, computer scientist Edsger W. Dijkstra published a provocative letter to the editor of the computer science journal Communications of the Association for Computing Machinery, titled *Go To Statement Considered Harmful*, arguing in favor of blocks.[340] His concern with *goto* statements, in essence, was that a programmer reading the text of a program would have difficulty mentally explaining how execution had reached any given point in the program, a problem lessened with block-based code.[341] Knuth, as someone obviously concerned about code readability, took issue with Dijkstra's view, arguing that in at least some cases, the real problem was meaningfully naming the location where a *goto* should go to.[342] Others contended that use of *goto* produced shorter, simpler code.[343] That programmers would so extensively debate a matter

---

336 *See, e.g.*, KERNIGHAN & RITCHIE, *supra* note 20, at 202.

337 *See, e.g.*, *id.* at 200–01.

338 This is the "structured program theorem," or the Böhm–Jacopini theorem. *See* Corrado Böhm & Giuseppe Jacopini, *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules*, 9 COMMC'NS ASS'N COMPUTING MACH. 366, 366, 368 (1966), https://dl.acm.org/doi/pdf/10.1145/355592.365646 [https://perma.cc/7BNC-Z49Y].

339 *See* ALFRED V. AHO & JEFFREY D. ULLMAN, PRINCIPLES OF COMPILER DESIGN 284–86 (Michael A. Harrison ed., 1977), https://archive.org/details/principlesofcomp0000ahoa/ [https://perma.cc/UA8J-DVF7] (describing procedure for converting block-based code into *goto* statements).

340 *See* Edsger W. Dijkstra, Letter to the Editor, *Go To Statement Considered Harmful*, 11 COMMC'NS ASS'N COMPUTING MACH. 147, 147–48 (1968).

341 *See id.* at 147 ("The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress." (emphasis in bold omitted)).

342 *See* Donald E. Knuth, *Structured Programming with Go To Statements*, 6 COMPUTING SURVS. 261, 294 (1974). The target label for a *goto* statement, incidentally, is another communicative element, because a compiler or assembler will turn the label into a numeric address. *See supra* text accompanying notes 56–57.

343 *See* Frank Rubin, Letter to ACM Forum, *"GOTO Considered Harmful" Considered Harmful*, 30 COMMC'NS ASS'N COMPUTING MACH. 195, 196 (1987); *see also* Donald Moore, Chuck Musciano, Michael J. Liebhaber, Steven F. Lott & Lee Starr, Letters to the ACM Forum, *"'GOTO*

of readability and expression suggests that at least some aspects of code, such as syntactic sugar, go beyond operation of a computer.

3. Style Guides

Further showing the importance of communicative elements of code to readability and expression are coding style guides. Coding style guides are books that specify how programmers should standardize the appearance of their code so that other programmers at the same company or using the same guide can easily understand it.[344] Topics often covered in these books include the communicative elements identified above: whitespace, indentation, positioning of braces, bound variable names, comments, and uses of various forms of expressions and syntactic sugar.[345]

Style guides demonstrate the importance of code as readable expression in two ways. They impose corporate standards to facilitate readability of code among employees.[346] But they also serve to restrain programmers from being too individualistic in their coding styles.[347] Style guides exist, in other words, because software firms recognize the personal, creative aspects of programming, and hope to restrain them by imposing standardization of the communicative elements of code.

4. Code Art and Code Golf

At the extreme end, manipulation of the communicative elements of code can turn cryptic instructions into displays of programming prowess or works of art.

Considered Harmful' Considered Harmful" Considered Harmful?, 30 COMMC'NS ASS'N COMPUTING MACH. 351, 351–55 (1987) (further letters in response to the debate).

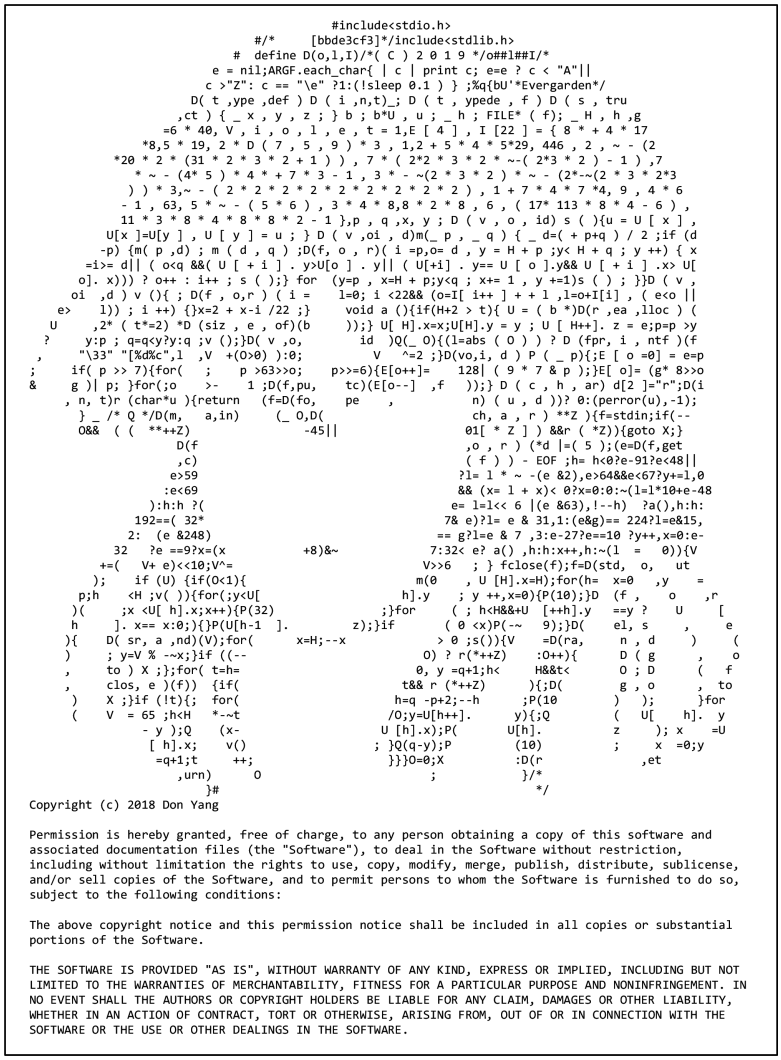
344 *See, e.g.*, *Google Java Style Guide*, *supra* note 294; DOLAND & VALETT, *supra* note 297; SUN MICROSYSTEMS, INC., *supra* note 292.

345 *See, e.g.*, SUN MICROSYSTEMS, INC., *supra* note 292, §§ 4–5, 6.2, 7.2, 8–9 (instructing on use of indentation, comments, placement of declarations, brackets, white space, and naming conventions in coding).

346 *See, e.g.*, *id.* § 1.1 ("Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.").

347 *See* Nicholas C. Zakas, *Why Coding Style Matters*, SMASHING MAG. (Oct. 25, 2012), https://www.smashingmagazine.com/2012/10/why-coding-style-matters/ [https://perma.cc/RAK9-D72C] ("Some see team-defined style guides as a way of forcing all developers to be the same.").

FIGURE. ENCRYPTION PROGRAM BY DON YANG



```
                                    #include<stdio.h>
                                #/*     [bbde3cf3]*/include<stdlib.h>
                              #  define D(o,l,I)/*( C ) 2 0 1 9 */o##l##I/*
                            e = nil;ARGF.each_char{ | c | print c; e=e ? c < "A"||
                           c >"Z": c == "\e" ?1:(!sleep 0.1 ) } ;%q{bU'*Evergarden*/
                         D( t ,ype ,def ) D ( i ,n,t)_; D ( t , ypede , f ) D ( s , tru
                       ,ct ) { _ x , y , z ; } b ; b*U , u ; _ h ; FILE* ( f); _ H , h ,g
                     =6 * 40, V , i , o , l , e , t = 1,E [ 4 ] , I [22 ] = { 8 * + 4 * 17
                  *8,5 * 19, 2 * D ( 7 , 5 , 9 ) * 3 , 1,2 + 5 * 4 * 5*29, 446 , 2 , ~ - (2
              *20 * 2 * (31 * 2 * 3 * 2 + 1 ) ) , 7 * ( 2*2 * 3 * 2 * ~-( 2*3 * 2 ) - 1 ) ,7
                 * ~ - (4* 5 ) * 4 * + 7 * 3 - 1 , 3 * - ~(2 * 3 * 2 ) * ~ - (2*-~(2 * 3 * 2*3
                ) ) * 3,~ - ( 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 ) , 1 + 7 * 4 * 7 *4, 9 , 4 * 6
               - 1 , 63, 5 * ~ - ( 5 * 6 ) , 3 * 4 * 8,8 * 2 * 8 , 6 , ( 17* 113 * 8 * 4 - 6 ) ,
              11 * 3 * 8 * 4 * 8 * 8 * 2 - 1 },p , q ,x, y ; D ( v , o , id) s ( ){u = U [ x ] ,
            U[x ]=U[y ] , U [ y ] = u ; } D ( v ,oi , d)m(_ p , _ q ) { _ d=( + p+q ) / 2 ;if (d
          -p) {m( p ,d) ; m ( d , q ) ;D(f, o , r)( i =p,o= d , y = H + p ;y< H + q ; y ++) { x
        =i>= d|| ( o<q &&( U [ + i ] . y>U[o ] . y|| ( U[+i] . y== U [ o ].y&& U [ + i ] .x> U[
       o]. x))) ? o++ : i++ ; s ( );} for  (y=p , x=H + p;y<q ; x+= 1 , y +=1)s ( ) ; }}D ( v ,
      oi  ,d ) v (){ ; D(f , o,r ) ( i =    l=0; i <22&& (o=I[ i++ ] + + l ,l=o+I[i] , ( e<o ||
    e>    l)) ; i ++) {}x=2 + x-i /22 ;}     void a (){if(H+2 > t){ U = ( b *)D(r ,ea ,lloc ) (
   U     ,2* ( t*=2) *D (siz , e , of)(b     ));} U[ H].x=x;U[H].y = y ; U [ H++]. z = e;p=p >y
  ?     y:p ; q=q<y?y:q ;v ();}D( v ,o,        id  )Q(_ O){(l=abs ( O ) ) ? D (fpr, i , ntf )(f
 ,     "\33" "[%d%c",l  ,V  +(O>0) ):0;          V   ^=2 ;}D(vo,i, d ) P ( _ p){;E [ o =0] = e=p
;     if( p >> 7){for(   ;    p >63>>o;    p>>=6){E[o++]=    128| ( 9 * 7 & p );}E[ o]= (g* 8>>o
&     g )| p; }for(;o    >-    1 ;D(f,pu,    tc)(E[o--]  ,f   ));} D ( c , h , ar) d[2 ]="r";D(i
     , n, t)r (char*u ){return   (f=D(fo,    pe    ,          n) ( u , d ))? 0:(perror(u),-1);
       } _ /* Q */D(m,   a,in)     (_ O,D(                    ch, a , r ) **Z ){f=stdin;if(--
       O&&  ( (  **++Z)              -45||                  01[ * Z ] ) &&r ( *Z)){goto X;}
                    D(f                                    ,o , r ) (*d |=( 5 );(e=D(f,get
                    ,c)                                    ( f ) ) - EOF ;h= h<0?e-91?e<48||
                  e>59                                    ?1= 1 * ~ -(e &2),e>64&&e<67?y+=1,0
                 :e<69                                    && (x= 1 + x)< 0?x=0:0:~(1=1*10+e-48
               ):h:h ?(                                  e= l=l<< 6 |(e &63),!--h)  ?a(),h:h:
             192==( 32*                                  7& e)?1= e & 31,1:(e&g)== 224?1=e&15,
            2:  (e &248)                                 == g?l=e & 7 ,3:e-27?e==10 ?y++,x=0:e-
          32   ?e ==9?x=(x          +8)&~                7:32< e? a() ,h:h:x++,h:~(1  =   0)){V
        +=(   V+ e)<<10;V^=                             V>>6   ; } fclose(f);f=D(std,  o,   ut
       );    if (U) {if(O<1){                           m(0    , U [H].x=H);for(h=  x=0   ,y    =
     p;h    <H ;v( ){for(;y<U[                        h].y    ; y ++,x=0){P(10);}D  (f ,    o    ,r
    )(     ;x <U[ h].x;x++){P(32)                   ;}for     ( ; h<H&&+U  [++h].y   ==y ?    U     [
    h     ]. x== x:0;){}P(U[h-1  ].         z);}if       ( 0 <x)P(-~   9);}D(    el, s     ,     e
   ){    D( sr, a ,nd)(V);for(      x=H;--x            > 0 ;s()){V    =D(ra,     n , d     )     (
   )     ; y=V % ~~x;}if ((--                        O) ? r(*++Z)    :O++){      D ( g      ,    o
   ,     to ) X ;};for( t=h=                        0, y =q+1;h<     H&&t<       O ; D      (   f
   ,     clos, e )(f))  {if(                       t&& r (*++Z)      ){;D(       g , o      ,  to
    )    X ;}if (!t){;  for(                      h=q -p+2;--h      ;P(10       )   );      }for
    (    V  = 65 ;h<H   *-~t                     /O;y=U[h++].      y){;Q        (   U[    h].  y
           - y );Q    (x-                      U [h].x);P(       U[h].         z     ); x    =U
            [ h].x;    v()                    ; }Q(q-y);P        (10)          ;     x  =0;y
             =q+1;t     ++;                     }}}O=0;X         :D(r             ,et
                ,urn)      O                        ;           }/*
                    }#                                          */
Copyright (c) 2018 Don Yang

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
associated documentation files (the "Software"), to deal in the Software without restriction,
including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial
portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

The figure shows an encryption computer program by Don Yang (along with the copyright license notice required to be included with it).[348]

For years, the International Obfuscated C Code Contest has invited programmers to submit their most cryptic programs, ones that perform generally simple tasks that are all but impossible to discern

348 The code featured in the figure won an award at the 2019 International Obfuscated C Code Contest. *See 2019/Yang - Most in Need of Transparency*, INT'L OBFUSCATED C CODE CONTEST, https://www.ioccc.org/2019/yang/index.html [https://perma.cc/DQ5Y-NLQK] (last visited Apr. 22, 2026). It is also viewable at Don Yang, *Violet.C*, GITHUB (Dec. 30, 2020), https://github.com/ioccc-src/winner/blob/master/2019/yang/violet.c [https://perma.cc/7MVX-USUR].

from reading the code.[349] Although the practice of coding a simple function in an absurdly complex way sometimes involves functional choices, the contest has at least in part turned into a matter of artistic skill.[350] By manipulating bound symbols and whitespace, judiciously choosing alternative expressions, arranging variable declarations, and making other nonfunctional choices, a clever programmer can arrange the letters, numbers, and symbols of computer code into desired shapes or figures.[351] For example, the previous figure is a working computer program for performing encryption but also has a plain artistic dimension wholly unrelated to that function.[352]

By contrast, minimization of code is sometimes the goal. This practice is sometimes called "code golf," insofar as a lower length is better.[353] The following one-line program in the Perl language computes the Fibonacci series:

```
$a=1;print$a-=$b+=$a*=-1,$/for 0..31
```

This program is functionally identical[354]:

```
$a = 1;
for (0..31) {
    $a *= -1;
    $b += $a;
    $a -= $b;
    print $a, $/;
}
```

But the former one, by virtue of being on one particularly cryptic line, makes the (unfortunately unknown) author seem especially proficient at programming.

These examples all go to show that the communicative elements of code give programmers a great deal of flexibility to structure programs with an eye to how they will be read by others. Their uses of these nonoperational elements demonstrate the kinds of creativity, originality, and authorial intent that copyright law requires, and can serve to advance copyright law's goal of expanding public access to works.

---

349 *See* David Cassel, *A Tradition Continues: The International Obfuscated C Code Contest*, New Stack (Apr. 15, 2018, at 09:00 ET), https://thenewstack.io/a-tradition-continues-the-international-obfuscated-c-code-contest/ [https://perma.cc/JF84-BRTK].

350 *See id.*

351 *See supra* Figure.

352 *See* Don Hsi-Yun Yang, *Info: Violet*, Uguu (Mar. 10, 2019), https://uguu.org/nfo_violet.html [https://perma.cc/GA2Q-VUCW] (describing the program's function and stating that the code arrangement depicts a character from an anime).

353 *E.g.*, Marco Faella, Seriously Good Software 281 (2020).

354 Again, it is possible that the Perl interpreter optimizes some of the constructions used in the shorter code, rendering the differences not merely expressive. *See supra* notes 251, 304 and accompanying text.

IV. Applying the Theory

If communicative elements are the touchstone of copyrightability in software, how much of existing software copyright law changes? The short answer is simple: not much. Literal copying of source code will still likely be an infringement, insofar as source code generally contains enough communicative elements, all of which would be copied. Even object code will probably remain protected under copyright,[355] because although some communicative elements like comments will be discarded upon compilation, others may survive into the object code.[356] For nonliteral copying, the communicative elements theory simply augments the *Computer Associates* abstraction-filtration-comparison test,[357] offering clarity as to what remains protectable after the filtration step.

Certainly, mere copying of communicative elements is not enough. A computer program author alleging infringement would have to show that the elements copied were sufficient in quantity to demonstrate substantial similarity, that those elements were not *scènes à faire*, that other indicia of functionality (e.g., industry jargon) do not obviate copyrightability, and so on.[358] Importantly, any individual communicative element standing alone is probably too insignificant to merit copyright protection;[359] it is the combination of many of them that makes for a copyrightable work. This, of course, makes software no different from any other copyrightable matter—an individual note on the musical

---

355 *Cf.* Apple Comput., Inc. v. Franklin Comput. Corp., 714 F.2d 1240, 1249 (3d Cir. 1983) (finding object code to be a "literary work" and therefore protected from unauthorized copying).

356 What elements will survive exactly depends on the compiler, but here are a few possibilities to consider. First, the order of subroutines in the object code may mirror, at least in part, their order in the source code. Second, many compilers preserve nonfunctional debugging information in executable files, which may include bound variable names. Finally, literal data such as text or graphics can be embedded in the data regions of the executable file. *See supra* note 67 and accompanying text.

357 *See* Comput. Assocs. Int'l, Inc. v. Altai, Inc., 982 F.2d 693, 706 (2d Cir. 1992).

358 *See, e.g.*, *id.* at 707–11. Indeed, a communicative element of software that is not a method of operation may nevertheless still fail § 102(b) for other reasons. For example, bound variable names used commonly in many places may not be copyrightable, not because they are methods of operating a computer, but because industry standardization gives those names acquired functionality to others in the industry. *Cf.* Mitel, Inc. v. Iqtel, Inc., 124 F.3d 1366, 1375 (10th Cir. 1997) (discussing lack of copyright protection for "elements of a work that necessarily result from external factors inherent in the subject matter of the work"); Justin Hughes, *Created Facts and the Flawed Ontology of Copyright Law*, 83 Notre Dame L. Rev. 43, 89 (2007) (discussing the adoption of standards "because those standards have legitimacy and social reference already"). Such industry-standard names are methods of operating a human.

359 *See, e.g.*, Sega Enters. Ltd. v. Accolade, Inc., 977 F.2d 1510, 1524 n.7 (9th Cir. 1992) (noting unprotectability of short phrases).

scale alone is insignificant, but enough of them in sequence makes a copyrightable melody.[360]

Where the communicative elements theory shines, however, is in its application to more difficult questions of software copyright. Two are considered below.

A. Application Programming Interfaces

The theory of communicative elements elegantly answers the question of copyrightability of subroutine declarations in the Oracle–Google litigation.[361] Here is an example of some declarations in Java:

```
public class Geometry {
    // Two-dimensional functions
    int circleArea   (int radius            ). . .
    int rectArea     (int side1, int side2  ). . .
    int triangleArea (int base,  int height ). . .

    // Three-dimensional functions
    int cylinderVol  (int radius, int height). . .
    int sphereVol    (int radius            ). . .
    int cubeVol      (int sideLength        ). . .
}
```

Multiple communicative elements are found here. Subroutines are sorted so that like items are together, and comments explain the groupings.[362] Whitespace creates pleasing visual alignment and helps to highlight similarities and differences. Bound variable names teach what inputs each subroutine expects. The presentation is formal, businesslike, and tabular.

If Google's Android code copied enough of these communicative elements, Oracle may have had a viable case.[363] Instead, though, Oracle

---

360 *See, e.g.*, Swirsky v. Carey, 376 F.3d 841, 848 (9th Cir. 2004) (premising copyright infringement on "a combination of elements, even if those elements are individually unprotected").

361 *See supra* text accompanying notes 186–93.

362 To be clear, the order of subroutines is *not* what Oracle asserted as "structure, sequence, and organization" in its case. *Oracle II*, 750 F.3d 1339, 1351 (Fed. Cir. 2014). In its brief to the Supreme Court, for example, Oracle offered a list of classes purporting to show the "unique organization the authors chose," but the list is in alphabetical order, indicating that the ordering was not what Oracle found unique. Brief for Respondent at 8–9 & fig. 1, Google LLC v. Oracle Am., Inc., 593 U.S. 1 (2021) (No. 18-956).

363 *See supra* text accompanying notes 355–60. There is reason to doubt at least that Google copied the variable names. Joshua Bloch, one of the designers of the Java programming language, has hypothesized that Google would have used a program called *javap* to generate its subroutine declarations. *See* Joshua Bloch & Pamela Samuelson, *Some Misconceptions About Software in the Copyright Literature*, 2022 Proc. Symposium on Comput. Sci. & L. 131, 134, https://dl.acm.org/doi/pdf/10.1145/3511265.3550449 [https://perma.cc/UJ4A-VHFF]. The *javap* program does not emit bound variable names for method arguments. *See Javap*, Oracle: Java Documentation,

focused its infringement theory on subroutine names and the taxonomic interconnections between modules, both of which had previously been shown to be operational elements.[364]

Where the Federal Circuit went wrong was its assumption that creative expression in code must be copyrightable—even if that expression is also operational and in disregard of § 102(b)—because otherwise "no computer program is protectable."[365] Had the court been aware of all the nonoperational, purely expressive communicative elements otherwise present in declarations, the impetus to disregard § 102(b) would have evaporated.

On the flip side, the communicative elements theory shows that the *Lotus* court erred as well in its holding that a menu hierarchy is wholly an "uncopyrightable 'method of operation.'"[366] As with *Google*, the menu item names were certainly operational because macro programs would have depended on those exact names to produce correct outputs.[367] The menu items, however, must have been shown in some order—"Copy" came before "Paste," or after.[368] That order of menu items would probably have no effect on macro programmers, making it solely communicative and possibly copyrightable.[369]

For both *Lotus* and the Google–Oracle litigation, then, the communicative elements theory offers a sharp delineation between operational and expressive elements, shows that some potentially copyrightable elements are present in both cases, and thus resolves the trilemma.

B. Generative Artificial Intelligence

The communicative elements theory also informs the debate on copyright law and AI. In one lawsuit filed against code-generative AI

---

https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html [https://perma.cc/7E96-NAUC] (last visited Mar. 21, 2026).

364 *See Oracle II*, 750 F.3d 1339, 1349–50 (Fed. Cir. 2014); *cf. supra* text accompanying notes 306–11 (subroutine names); *supra* text accompanying notes 298–305 (subroutine taxonomy). It is the taxonomy of connections between classes and modules that Oracle referred to in its "structure, sequence, and organization" argument. *See Oracle II*, 750 F.3d at 1351, 1367–68. This taxonomy is operational for the same reason that subroutine names are operational: A third-party program, invoking a subroutine, must identify that subroutine by its exact taxonomic location, and changing the location will produce an error output. *See* GOSLING ET AL., *supra* note 56, at 89–98 (explaining how Java compiler determines taxonomic position of a method or other name).

365 *Oracle II*, 750 F.3d at 1367.

366 *See* Lotus Dev. Corp. v. Borland Int'l, Inc., 49 F.3d 807, 815 (1st Cir. 1995).

367 *See id.* at 812–13, 815–16.

368 *See id.* at 810.

369 *See supra* Section III.E. Two important caveats: First, even though menu item order may not be a method of operating a computer, it might be unprotectably functional because of conventional user expectations—most people expect "Paste" to follow "Copy." Second, if menu order were the sole copyrightable matter, Borland might not have infringed, because Borland ultimately removed the visual menus. *See Lotus*, 49 F.3d at 812–13.

systems, the complaint included the following output from Github's Copilot:

```
"function isEven(n) {
    if (n == 0)
      return true;
    else if (n == 1)
      return false;
    else if (n < 0)
      return isEven(-n);
    else
      return isEven(n - 2);
 }."[370]
```

It then quoted a textbook containing the following example code:

```
"function isEven(n) {
    if (n == 0) return true;
    else if (n == 1) return false;
    else if (n < 0) return isEven(-n);
    else return isEven(n - 2);
 }"[371]
```

and alleged: "Aside from different line breaks—which are not semantically meaningful in JavaScript—this code for the function 'isEven' is the same as what Codex produced."[372]

Line breaks may not be meaningful to the operation of the Javascript program, but they are a communicative element of code and *semantically meaningful to the reader*.[373] At a glance, the Copilot output is vertical and indented, while the textbook version is concise, drawing less attention to the return values due to a lack of alignment. Many of the obvious similarities in the code, by contrast, go to the programs' methods of operation—the order of the if-else statements, the return values, and the conditions being tested.[374]

---

370 First Amended Complaint at 14, Doe v. Github, Inc., 672 F. Supp. 3d 837 (N.D. Cal. 2024) (Nos. 4:22-cv-06823, 4:22-vc-07074), Dkt. No. 98.

371 *Id.* at 16 (quoting MARIJN HAVERBEKE, ELOQUENT JAVASCRIPT 35, 53–54 (4th ed. 2024)), https://eloquentjavascript.net/ [https://perma.cc/7Z7F-QVDS]. This code is located in the textbook solutions, which can be found at Marijn Haverbeke, *3.2 Recursion Solution*, ELOQUENT JAVASCRIPT: CODE SANDBOX, https://eloquentjavascript.net/code/#3.2 [https://perma.cc/M4MP-P5G4] (last visited Apr. 22, 2026).

372 *Id.* at 16.

373 *See supra* Section III.B.

374 *See supra* text accompanying notes 95–97 (defining method of operation). In particular, the major similarity between the two is the rather idiosyncratic way of determining number evenness (turning the number positive and then subtracting two repeatedly until either zero or one is reached), which definitely affects the speed and performance of the operation. *See* CARDOSO ET

Caution is required in analyzing copyright issues with code-generative AI. Naïve visual comparison risks assuming copyright protection for operational elements, contrary to § 102(b). By contrast, this Article's framework draws focus to different similarities in the above code: the bound variable *n*, the lack of braces around the return values, and the placement of each if-else statement on a separate line, for example. Whether these elements rise to the level of protectable expression still must be addressed,[375] but at least the analysis would be based on expressive rather than operational elements.

Additionally, the communicative elements theory offers a larger insight into the relationship of copyright law and AI. In the above example, Copilot apparently recognized that alterations to line breaks are operationally irrelevant to JavaScript code and *added new information through indentation that arguably improved the communicative experience*. Although one can only guess as to how Copilot's programming recognized this,[376] it seems that Copilot has inferred the difference between communicative and operational elements of JavaScript code and can combine them in generating new code.

It is interesting enough that a computational system untrained in copyright law might be able to distinguish operational and expressive elements in computer code. But what about other instructional texts or creative texts? Futurecasters have worried that generative AI might be able to separate ideas from expression, delivering free summaries of news to the detriment of journalists and others.[377] But this worry assumes the remarkable possibility of a machine learning the line between idea and expression, a line that has eluded the legal profession for over a century.[378] The possibility that generative AI might have answers to a longstanding legal conundrum at a minimum merits further investigation.

CONCLUSION

To advance the resolution of the dilemma of software copyrightability, this Article has proposed a new theory for identifying

---

AL., *supra* note 302, at 132 ("In many cases, code reordering may significantly impact performance/energy/power.").

375 *See supra* text accompanying notes 355–60.

376 *See* Lee et al., *supra* note 5, at 284–85 & nn.113–16.

377 *See, e.g.*, Benjamin L.W. Sobel, *On Copyright, "Facts," & Generative AI*, CORN. TECH: DIGIT. LIFE INITIATIVE (Sep. 23, 2024), https://www.dli.tech.cornell.edu/post/on-copyright-facts-generative-ai [https://perma.cc/8TK8-ZZXU] ("Courts and/or policymakers might similarly balk at generative AI's non-expressive use defense if the technology threatens to undermine incentives for, say, original journalism or other expressive endeavors.").

378 *See* Peter Pan Fabrics, Inc. v. Martin Weiner Corp., 274 F.2d 487, 489 (2d Cir. 1960); Nichols v. Universal Pictures Corp., 45 F.2d 119, 121 (2d Cir. 1930).

copyrightable subject matter in software. Drawing on the structure of programming languages and theory of computer code style, it has identified "communicative elements" of code that serve a purely expressive purpose: explaining how the code works, without affecting the underlying functionality of the computer. This theory resolves the longstanding doctrinal tension between § 102(b) and § 117 of the Copyright Act, and it provides insights into two of the most difficult questions of software copyright today, namely the *Oracle v. Google* question of application programming interfaces and the training of code-generating AI systems.