Japan Bilingual Publishing Co.

Philosophy and Realistic Reflection

https://ojs.bilpub.com/index.php/prr

ARTICLE

A Case against Stephen Kleene's Incomputability

Edgar Graham Daylight

Department of Computer Science, KU Leuven, Box 2402, 3001 Leuven, Belgium

ABSTRACT

In *Mathematical Logic* (1967), Stephen Kleene addresses a central activity of software engineering through his introduction of tally-based Turing machines. Yet in doing so, he inadvertently builds a subtle form of impredicativity into his theory: he implicitly defines the semantics of Turing computability by appealing to the very construct—the Turing machine—whose legitimacy is simultaneously under examination. Stewart Shapiro, in *Acceptable Notation*, draws attention to this tension by uncovering the non-mathematical assumptions that tend to enter Kleene-style computability proofs, ultimately grounding his critique in what he terms "an extended version of Church's thesis." In this paper, we revisit Shapiro's concerns but shift the focus to incomputability. We argue that Kleene's framework relies on a latent assumption that complicates the contemporary presentation of the Halting Problem. Specifically, Kleene begins with a notation-dependent foundation: his tally-based Turing machines provide genuine evidence of a limitation internal to that particular representational system. However, he then moves rapidly from this narrow result to a broad, notation-independent claim of incomputability. Our critique is that Kleene's account lacks the necessary conceptual transition from notation-dependent reasoning to notation-independent conclusions. Once this gap is made explicit, what remains is not a universal incomputability theorem but an incomputability claim relative to a specific representational choice. This reframing invites a more careful reconsideration of how the Halting Problem—and, more generally, the notion of incomputability—should be understood within the contemporary sciences.

Keywords: Kleene; Halting Problem; Notation

*CORRESPONDING AUTHOR:

Edgar Graham Daylight, Department of Computer Science, KU Leuven, Box 2402, 3001 Leuven, Belgium; Email: egdaylight@dijkstrascry.com

ARTICLE INFO

Received: 4 August 2025 | Revised: 26 December 2025 | Accepted: 6 January 2026 | Published Online: 29 January 2026

DOI: https://doi.org/10.55121/prr.v3i1.658

CITATION

Daylight, E.G., 2026. A Case against Stephen Kleene's Incomputability. Philosophy and Realistic Reflection. 3(1): 103–125.

DOI: https://doi.org/10.55121/prr.v3i1.658

1. Introduction

Around 1967, and still today in specialized fields, skilled and imaginative individuals in software engineering essentially translate number-theoretic functions (specifications) into Turing machines (program texts). This form of engineering[1,2], a specialized subset of broader human-in-the-loop industrial practices, should not be conflated with functional programming[3,4]. The latter, rooted in Kleene's computability theory, considers only such translations that are automatable within a fixed symbolic framework.

The countably many automatable translations employed by functional programmers—or, similarly, by Kleene-style computability theorists—barely scratch the surface compared to the uncountably many translations available to the creative software engineer in industry. Fundamentally, engineering is not to be equated with automation, nor are engineers mere automatons—let alone tally-based Turing machines—in disguise.

Yet in automata theory courses and Stephen Kleene's 1967 *Mathematical Logic*[5], not only the rote technician—who operates strictly within a fixed notation—but also the creative engineer—who has the potential to engage with an extensive range of notations—are *implicitly* replaced by the tally-based Turing machine *itself*. This paper contends that Kleene's 1967 work embodies this substitution, arguing that, even from a strictly academic standpoint, such a replacement is unwarranted. It results in a form of impredicativity, wherein the semantics of Turing computability are defined in terms of the very construct (the Turing machine) whose legitimacy is under scrutiny.

Our overarching concern has already been acknowledged—both in practice and in theory. In the realm of computing practice, Peter Naur offered the following assessment of software engineering as of 1985:

> More generally, much current discussion of programming seems to assume that programming is similar to industrial production, the programmer being regarded as a component of that production, a component that has to be controlled by rules of procedure and which can be replaced easily. Another related view is that human beings perform best if they act like machines, by following rules, with a consequent stress on formal modes of expression, which makes it possible to formulate certain arguments in terms of rules of formal manipulation. Such views agree well with the notion, seemingly common among persons working with computers, that *the human mind works like a computer*. At the level of industrial management, these views support treating programmers as workers of fairly low responsibility, and only brief education[6] (pp. 237–238, our emphasis).

From a theoretical standpoint, Stewart Shapiro's *Acceptable Notation*[7] underscores the presence of non-mathematical assumptions embedded in several steps of Kleene-style computability proofs, culminating in what he describes as "an extended version of Church's thesis." In our interpretation, Shapiro—perhaps inadvertently—adds weight to Naur's broader critique of logico-mathematical methods, a critique made explicit in a 2011 account[8].

Besides the work of Naur and Shapiro, other contributions relevant to our project have recently come to our attention, including those of Dale Jacquette[9,10], Jean Paul Van Bendegem[11], Luca San Mauro et al.[12,13], Paula Quinon[14,15], and Henri Stephanou[16]. In this paper, however, our focus is limited to Shapiro's technical concerns, albeit with a shift in emphasis from computability to incomputability. We argue that Kleene's 1967 framework harbors an assumption, reminiscent of the Church-Turing Thesis, that subtly undermines the contemporary relevance of the Halting Problem in modern computer science textbooks.

Named after Alonzo Church and Alan Turing, the *Church–Turing Thesis* originates from Stephen Kleene's deliberate conflation of (A) Church's Thesis and (B) Turing's Thesis. Kleene elaborated on this in his *Mathematical Logic*, as follows:

> (A) Church proposed the thesis (published in 1936) that all functions which intuitively we can regard as computable, or in his words "effectively calculable", are λ-definable, or equivalently general recursive.
> (B) A little later but independently, Turing's paper 1936–1937 appeared in which another exactly defined class of intuitively computable functions, which we shall call the "Turing com-

> putable functions", was introduced, and the same claim was made for this class; this claim we call *Turing's thesis*.
> Turing's and Church's theses **are equivalent**. We shall usually refer to them both as *Church's thesis*, or in connection with that one of its three versions ... deals with "Turing machines" as *the Church–Turing thesis*.

Remark 1. *The previous snippet is quoted from Kleene[5] (p. 232), original italics, our boldface. We conjecture that Church, acting as the sole reviewer of Turing's seminal paper, requested Turing to establish a particular connection between the latter's automatic machines and his own functions. In this regard, see Remark 1 in Lucas' historical overview [17]. For two distinct correspondences between raw syntactic machines and number-theoretic functions, see Burgin[18] or Daylight[19].*

In 1982, Stewart Shapiro revised Kleene's conflation of (A) and (B) through his *extended version of Church's thesis*, which we paraphrase[7] (p. 17) thus:

> A number-theoretic function is recursive iff it is Turing machine computable relative to stroke notation.

Shapiro's emphasis on *stroke notation* involves using a string of n strokes on the tape of a Turing machine (TM) to denote the natural number n. Championing Kleene's conflation of (A) and (B) implies acceptance of Shapiro's nuanced, extended version of Church's thesis.

Remark 2. *A minor distinction between Shapiro and Kleene is that Shapiro uses n strokes on a tape of a TM to denote the natural number n, while Kleene uses $n + 1$ tallies to represent n.*

We argue that, while Shapiro's corrective stipulation regarding notation is a commendable start, it remains incomplete. To address this, we will delve deeper into Kleene's 1967 exposition, particularly his incomputability theorem concerning the Halting Problem. We contend that this theorem requires a more substantial correction.

Our aim is to first scrutinize the mathematics of Shapiro (1982) and then criticize that of Kleene (1967). We will demonstrate that, at times, Kleene deliberately intertwines his semantics with his syntax, such as the number 2 with its notation on the tape of a TM, in accordance with Shapiro's explicit reference to an extended version of Church's thesis. However, we will also show that there are instances where Kleene completely overlooks this conflation, necessitating a technical amendment on our part.

We will conclude that Shapiro's computability is sound, whereas Kleene's proof by contradiction regarding incomputability relies on an unstated assumption stemming from his oversight. Upon reflection, we will clarify that Kleene's hidden assumption falls outside mathematics according to his own criteria for what constitutes a proof.

Outline

Our purpose is not to prove a new theorem, but to reassess modern incomputability theory through the lens of Kleene's 1967 framework. The remainder of this paper is organized into five sections and an appendix. First, we begin with an exemplum illustrating the diverse interpretations of computability theory's foundations (Section 2). Certain strong statements in the exemplum do not represent our personal views. Second, we introduce four tenets and a synopsis to orient the reader (Section 3). Third, we scrutinize Shapiro's seminal 1982 account of acceptable notation (Section 4). Fourth, we criticize Kleene's incomputability theorem from 1967 (Section 5). Fifth, we present our conclusions (Section 6). Additionally, **Appendix A** clarifies the distinction between 'descriptive' and 'prescriptive' as they are used in this paper in relation to symbolic logic and Chomskian automata.

Specialists in mathematical logic may begin with the synopsis in Section 3.3, and then immediately proceed to Section 5, where we isolate the internal error in Kleene's framework.

2. Exemplum

There we sit—my wife, our daughter, and I—in the office of the toe surgeon. Meticulous measurements are being taken of her 14-year-old toe, every angle noted with precision, while the surgical plan unfolds in detail. It feels reassuring, almost comforting: science, at its most elegant, in action.

We discuss a range of possible outcomes—both short-term and long-term. Each scenario is laid out with clarity, its advantages and drawbacks transparently explained. Even the risks, including the rare but real chance of surgical error, are addressed with refreshing honesty. Photographs of her current toe are placed beside digital projections: simulations of what it might look like right after the surgery, five years down the road, and even two decades into the future.

My wife, who has some background in the field, occasionally steers the conversation into technical territory. Everything feels under control. Everything makes sense.

Until, that is, the head surgeon walks in.

With a calm but curious tone, he introduces a surprising limitation. While he can adjust our daughter's toe by 2.5 or 2.6 millimeters, an exact shift of 2.55 mm, he explains, is unattainable. "It is because of an impossibility result in logic," he says. "The halting problem. That level of precision would require an incomputable computation."

He says this without a hint of irony.

And just like that, I stand up. Quietly, but with resolve, I gather my wife and daughter, and we leave.

Remark 3. *It is hard to imagine even a single reader taking the side of the toe surgeon at this early point in the narrative.*

> "Wait, wait!" the toe surgeon calls after us. "You do admit that there are computable and incomputable functions, right? You studied logic yourself, after all."
> "Yes," I reply. "In theory, there are both. But even computable functions are physically incomputable."
> He squints. "Huh?"
> "Take the identity function," I say. "Totally computable in theory. But try executing it on an input that's too large."
> "Huh?" he repeats. "Are you seriously suggesting that a finite machine implies a fundamental limitation? That's absurd. If needed, I can simply build a bigger machine."

Remark 4. *At this point, it is easy to imagine multiple readers siding with the surgeon.*

> "Not in the real world," I say. "Physics won't let you."
> "Excuse me?"

At that moment, the surgeon's teenage son strolls into the room. He throws his father a sideways look, then, with a calm but pointed tone, jumps in on my side:

> "Any automaton from the Chomsky hierarchy is (or, rather, resembles) a monolithic system, tied to a global clock. When its state space gets too large, modern physics pushes back.
> General relativity, quantum mechanics—they do not allow for arbitrarily large, synchronized computation. As the size of the practical realization of the automaton increases, signal propagation would need to approach infinite speed to maintain a coherent global state at each clock tick. But physics caps that with the speed of light.
> So, any real-world implementation of a sufficiently large logic engine has to use asynchronous, distributed components—each with its own local clock and state. There is no true 'global' state anymore. Scholars like Carl Petri recognized this decades ago and proposed new models of computation to deal with it[20,21].
> But most people—yourself included, dad—are so enamored with the Universal Turing Machine, as if it were sacred. Turing himself was more pragmatic."
> The surgeon glares. "Why must you always be so difficult, son? I am not talking about physics—I am talking about pure logic."
> "There is no problem, then," the teenager and I reply in unison. "But if you are referring to real-world equipment—and toes—you cannot pretend this is just a matter of logic."
> The surgeon stares in disbelief. "Are you seriously claiming that incomputable functions can be computed?"
> "We are saying that, in the physical world, both computable and incomputable functions are only ever approximated. Sometimes it is useful to model your equipment's behavior as a Turing computable function. Other times—especially in distributed systems—you

will want to work with Turing incomputability, too."

The surgeon throws up his hands. "So what do you want me to do? Operate on your daughter's foot using a Turing machine or with the incomputable gadget you are talking about?"

I smile calmly. "Neither is possible. You keep equivocating physical computability and Turing computability. I believe we will be seeking footcare elsewhere."

"You fail to appreciate logic!" he shouts after me.

Outside, as the door clicks shut behind us, my daughter glances up and whispers, "Did that argument with his dad really have to happen? The son was kinda cute."

I grin. "He is not just cute, sweetheart. He also understands that his father confuses Aristotelian and Platonic reasoning."

Remark 5. *A gentle introduction to the interplay—and tension—between Aristotelian and Platonic modes of reasoning in the context of computability is provided in Daylight[22]. The author acknowledges having confused both perspectives and holds the view that many figures within computer science do the same.*

That night, my daughter dreamed of the words the surgeon's son and I had spoken in unison:

> "We argue that, in the real world, both computable and incomputable functions can only be approximated. In certain contexts, it is useful to model your equipment's operation as a computable function; in others—particularly with distributed, asynchronous components—it is common to consider incomputable ones as well."

Earlier that month, she had studied Tony Hoare's work on formal verification, including his formalization of "fairness" in distributed systems—a concept that, intriguingly, ensures (mathematical) behavior beyond the reach of Turing computability. She had begun to suspect that software engineers sometimes rely on frameworks that quietly transcend the limits of classical computability.

Remark 6. *A technical synonym for "fairness" is finite delay, discussed by Cardone[23].*

And yet, something tugged at her mind. Through countless conversations with me, she had indeed come to regard the Church-Turing Thesis (CTT) less as a law of nature and more as a hypothesis about how humans conceptualize algorithms. Even so, she found herself repeatedly drawn back to a more physical interpretation:

> "I cannot think of a single example of a non-Turing-computable function f that can be reliably computed by a physical process—at least over a sufficiently broad domain. If someone could provide such an example, I would be genuinely intrigued. But if no one can, that in itself is striking—perhaps the CTT does, in the end, hold true after all."

Is this reasoning sound? In her sleep, she murmured:

> "On the one hand, I am more convinced than ever that the CTT says nothing definitive about physics. On the other hand, I find myself continually searching for a physical process—and in its absence, I am tempted to accept the CTT after all. Is that not a contradiction? Help!"

Just then, the surgeon's son appeared in her dream and whispered:

> "Even the identity function cannot be reliably translated into a physical process. Once we accept this, the implication becomes clear: the CTT says far less about physics than computer scientists often suggest—and whether or not it is mathematically disproven has no direct impact on physics or engineering."
>
> Thrilled, she exclaimed: "So, the CTT could have been quietly disproven many times—yet mainstream computer scientists refuse to acknowledge it?"
>
> "Exactly," he replied. "There is a long tradition of independent thinkers publishing on this. See, e.g., the three papers [24–26] for a small sample. I find Doukas Kapantaïs' recent work especially intriguing[27,28]. He outlines a model of computation that is irreducible to the Turing

machine model, yet still satisfies Hilbert's criteria. But do not expect a revolution in physics—because the CTT simply does not belong there. However, engineers and physicists might still find such conceptual advances relevant in the long term."

The next morning, my daughter asked me: "The surgeon mentioned the 'halting problem.' But what does it actually mean according to you?"

I smiled. "In the context of your toe? It is pure intellectual self-gratification."

I explained that the halting problem originates in mathematical logic—often repurposed as intellectual window dressing. It gained prominence when logicians, long marginalized, found relevance with the rise of computers in the 1940s. They leveraged Turing's work on symbolic machines and computational limits to frame the emerging field of computing as a theoretical achievement—much more a triumph of logic than of engineering. I continued:

"In subsequent decades, the first generation of 'computer scientists' rewrote the history of their field. (Compare, for instance, Davis history building[29] with that of Priestley[30] or Daylight[31].) Today, computers are often presented as logic engines, much like dice are regarded as purely mathematical objects, or planetary motion as mere differential equations."

"But isn't that way of thinking useful? You said so yourself last week," my daughter pointed out.

"True," I conceded, "but only if you recognize that it is not always applicable. When discussing computability, the surgeon frequently conflates the model with the thing-in-itself. Now, if he explicitly insists on blurring the categorical realms of mathematics and physics, then I propose he advocates for an Aristotelian realist philosophy of mathematics. The best guidance I can offer is to direct him to James Franklin's work[32]. However, as Franklin clarified to me in correspondence, this would require abandoning core tenets of Platonist set theory. And without that framework, the diagonal argument underpinning the incomputability of the Halting Problem collapses."

Remark 7. *Linnebo and Shapiro's argument[33] may challenge this line of reasoning, yet our forthcoming work aims to expose its limitations—even in light of further developments akin to those presented by Cook[34]. These ongoing dialogues underscore the contested and continually evolving status of computer science's so-called "fundamentals."*

I continued: "So, either you embrace a neo-Aristotelian guise of philosophy without preaching impossibility results (as we teach them today), or you follow Hilbert and emancipate mathematics from physics. These thoughts are, on my reading, crisply formulated by Paul Henry:"

The machines we have considered here, the combination of the Euclidean straightedge and compasses, Descartes' machine, and the Turing machines, have all of them had something to do with the foundations of mathematics. I have insisted upon the fact that those machines have had a theoretical function and that there was no need to materially construct them for them to be operational from that point of view. Furthermore, I said that in reality, they can not be constructed. Now, I can add that it is precisely as "machines impossible to materially construct" that they give rise to impossible problems[35] (p. 121, our italics).

The next day, on their first date: "Still, you should be careful with that," Marcus—the surgeon's son—cautioned my daughter when, instead of watching a movie on their first date, they ended up debating logic and computer science.

"Careful with what?" Helena asked.

"If you view the CTT more as a hypothesis about how humans conceptualize algorithms—rather than a claim about physical computation—you are still circling back to physics," he cautioned.

"Oh, I see, because a conceptualizing person quickly brings us to the person's brain, and thus back to physics," my daughter exclaimed.

"Perhaps that is why computer scientists equate not just computers, but also human brains, to Turing machines. How convenient. So much for open-mindedness[36]."

"The irony," Marcus continued, "is that neither Turing nor Gödel, upon closer inspection, valued such extreme algorithmic thinking[37,38]—let alone today's cognitive scientists. The human brain (including a software engineer's) is not necessarily a Chomsky automaton, though contemplating such an equivalence can be useful in well-defined contexts. Stanislas Dehaene's books are worth reading on this."

Remark 8. *In this regard, Peter Kugel advocates for Putnam-Gold machines (and even more powerful machines) rather than standard TMs in the study of the mind's machinery. On the one hand, he aligns himself with the intellectual positions of Turing and Gödel and cites Stanislas Dehaene as a potential contemporary supporter. For two other cases in point, see Bringsjord and Arkoudas[39] and Longo[40]. On the other hand, Kugel also recognizes that many colleagues in computer science do indeed favor the standard TM[41,42].*

3. Four Tenets and a Synopsis

The preceding exemplum is intentionally tangential to the main content of this paper; it underscores the potential for strong disagreement among our readers regarding the foundations of modern logic and theoretical computer science. For instance, consider the very notion of the Turing machine. As Curtis-Trudel highlights in a recent survey, perspectives diverge sharply on this matter[43]. Piccinini, for example, asserts:

> The tape [of the TM] and processing device are explicitly defined as spatiotemporal components[44] (pp. 119–120).

Curtis-Trudel further examines the "Turing-machine realisism" advocated by Copeland and Shagrir, who posit an "extra ontological level … with Turing machines 'having' causal features"[45] (p. 234). In stark contrast, others interpret the TM as a purely abstract model. Rescorla articulates this view clearly:

> To describe a physical system's computational activity, scientists typically offer a computational model, such as a Turing machine or a finite state machine. Computational models are abstract entities. They are not located in space or time, and they do not participate in causal interactions[46] (p. 1277).

In short, there is no consensus on whether a TM is Aristotelian, Platonic, or something else entirely. Given this divergence, it is hardly surprising that scholars might project differing interpretations onto historical figures like Turing, Church, and Kleene.

Moreover, since our audience spans logicians and engineers, we recognize that not all readers will readily accept the four tenets underlying this paper. For transparency, we state these tenets at the outset.

3.1. Two Direct Tenets

To frame our first tenet via the following proposition—where an "infinite abstraction" is best understood as the mathematical idealization of an infinitely long tape—it is helpful to recall that Putnam-Gold machines (whose details we can set aside) compute a wider class of number-theoretic functions than standard Turing machines.

Proposition 1. *No finite experiment in the real world can falsify all the beliefs underlying X, where X refers to a model of computation that relies on an infinite abstraction such as Putnam-Gold machines and standard TMs.*

The reader should note that we are not suggesting that a finite state machine (FSM) is computable while a Turing machine (TM) is not. Our point is rather that in the physical world of engineering, infinite abstractions are only ever approximated with finite precision.

Two detailed remarks can now be made in connection with Proposition 1. First, Peter Kugel correctly cautions that no finite experiment can demonstrate that human minds are equivalent to, say, Putnam-Gold machines rather than standard TMs. In this regard, he paraphrases Marvin Minsky as follows: "There is no evidence for this, for how could you decide whether the mind or a physical device computes an uncomputable predicate?"[47] (p. 175). Second, Kugel then responds to Minsky, thus:

> You can't. But that does not mean that one might not choose uncomputable models anyway, much as one might choose the infinite (Turing Machine) models, to which most of Minsky's (1967) book is devoted, over the more finite (Finite Automaton) models, even though one cannot prove, on the basis of finite evidence, that any given physical system is not one of the latter[47] (p. 175).

Regarding our second tenet, we ask our audience to be skeptical of historical claims promoted by key figures in the field and open-minded about those written by historians. For instance, Kleene, a prominent member of the Princeton school, offers an interpretation of Alan Turing, whom he did not know personally. Kleene writes thus:

> Turing's machine concept arises from a direct effort to analyze computation procedures as we know them intuitively into elementary operations. Turing argued that repetitions of his elementary operations would suffice *for any possible computation*[5] (p. 233, our emphasis).

However, writing as a historian, the present author proposes a markedly different perspective on Alan Turing's broader views on computing during the period 1932–1948:

> While computer science takes Turing's universal machine as the limit of all achievable forms of computability, it was explicitly perceived as banal by Turing in 1948. From his perspective, a creative person is significantly more than that which a fixed symbolic logic or a universal Turing machine can offer[36] (p. 546).

Our evolving personal understanding suggests that Turing may have become open to advocating for what would later be known as Putnam-Gold machines *over* standard TMs in relation to the human mind and/or engineered computers, whereas Kleene would not have.

In this paper, we do not seek to advocate for any particular stance—though our personal biases may occasionally surface—when examining the expositions of Shapiro and Kleene, in Sections 4 and 5, respectively. Instead, our central assertion is simply that if Kleene did equate the human mind (such as that of a software engineer or the director of a computing laboratory) with a tally-based TM, then this should be explicitly reflected in his theoretical framework.

3.2. Two Surprising Tenets

Closely related to our second tenet are our two final tenets. In general terms, we urge our audience to be cautious of universal claims made by theorists about engineering. We posit that engineers who are well versed in logic are likely to disagree with Kleene's assertion that the societal "significance" of an incomputable number-theoretic function $\psi()$ "comes from the Church-Turing thesis," by which:

> "computability in Turing's sense agrees with the intuitive notion of computability. Accepting the thesis, as most workers in [maths'] foundations do, the director of a computing laboratory must fail if he undertakes to design a procedure to be followed, or to build a machine, to compute this function $\psi(a)$."

These words are quoted from Kleene[5] (p. 245). Kleene's perspective on the incomputability of the function $\psi()$ will be scrutinized in Section 5.

Let us zoom in on our third tenet, potentially the most surprising for logicians and theoretical computer scientists.

Software engineering, we assert, operates beyond a countable realm of mathematical discourse. The uncountable realm of creative engineering is central to Francky Catthoor and his collaborators[1,48,49] and is revisited in Section 4.5. For now, it suffices to highlight that there are uncountably many ways for an engineer to represent a number-theoretic function with a machine. Only two such possibilities (arrows) are depicted in the following diagram with regard to function $\varphi_j()$.

$$\begin{tikzcd}
\text{semantics} & & \varphi_j() \arrow[dl] \arrow[dr] & \\
\text{syntax} & M_i & & N_k
\end{tikzcd}$$

In other words, engineers are inclined to examine a broad family of listings, including listing $\{M_i\}_{i>0}$ and listing $\{N_k\}_{k>0}$. While the first listing fits in Kleene's tally-based account of TMs, the second listing fits in, say, a narrative in which numbers are represented with bits 0 and 1. There are uncountably many such listings. Each member in a listing is a machine specification (or a program text). Each arrow in the diagram represents a sequence of consistent design choices.

Crucially, only after the engineer selects specific machines M_i and N_k (and, hence, specific notations) can the theorist *describe* the translation from M_i to N_k using, say, a tally-based Turing machine (TM). However, the theorist cannot *prescribe*—cannot exhaustively predetermine, via one of his countably many tally-based TMs—the choices available to the engineer in future software development. The ensemble of all tally-based TMs fails to fully encompass the engineer's design space, which includes uncountably many arrows—of which only two are shown in the diagram. Readers who consider 'descriptive' and 'prescriptive' synonymous at this point are encouraged to consult **Appendix A** for a clarification of this distinction.

The essence of mathematical engineering lies in prediction, control, and anticipation of the engineering activity at hand. Kleene's immediate focus on a single notation—tally-based notation—stands in stark contrast to the practical demands of engineering practice. More importantly, this paper argues that Kleene's incomputability result requires substantial qualification to maintain logical coherence, much less engineering applicability.

Hence, coming back to the theorist, he must explicitly stipulate whether to permit or exclude an uncountable number of functions mapping semantics onto syntax. Subsequent reasoning heavily relies on this assumption, especially when employing a proof by contradiction. It will turn out that, as theorists, both Shapiro and Kleene adopt a mathematical perspective on industrial practice that is inherently tally-based and algorithmic. In their view, the creative engineer, who bridges number-theoretic functions (semantics) and program texts (syntax), is an automaton, operating methodically within a tally-based realm. Only the director of such a methodical laboratory must indeed fail when attempting to compute Kleene's incomputable function $\psi()$.

Finally, we present our fourth tenet—the elephant in the room: the notion, reminiscent of 21st-century analytic philosophy, that semantics and syntax are 100% separable, as assumed in the previous diagram. Let us suppose that all of software engineering can be captured a priori within a countable realm of symbolism after all. Even then, we take issue with Kleene's interpretation of his incomputable function $\psi()$. Kleene believes that his function $\psi()$ is incomputable regardless of the syntactic choices made by an engineer. However, we will argue that the mathematical significance of Kleene's function $\psi()$ only emerges when his own syntactic conventions, R_K and C_K, are explicitly adhered to, as opposed to those employed by one of his equally algorithmic-minded colleagues.

Remark 9. *Kleene's conventions, R_K and C_K—denoted by "K" for "Kleene"—are detailed in Section 5. For now, it suffices to associate R_K with a mapping that relates the natural number n to $n+1$ tallies on a TM's tape, while the coding method C_K corresponds to Kleene's specific choice of Gödel coding, ensuring compliance with R_K.*

Kleene defines his function $\psi()$ in terms of his famous predicate $T(i, a, x)$. This predicate does not convey Kleene's intended meaning unless the reader consistently applies Kleene's conventions, R_K and C_K, to the natural numbers i, a, and x. The key point is that *Kleene's semantics hinges on his specific choices regarding syntactic TMs.* The director of a computing laboratory is unlikely to endorse Kleene's interpretation of incomputability unless his engineers function merely as rote technicians, adhering strictly to the conventions R_K and C_K.

To summarize, while Shapiro emphasizes that number-theoretic computability is relative to notation (i.e., syntax), we will argue, contrary to Kleene, that number-theoretic incomputability is also merely a notation-dependent concept. The notion of absolute incomputability, as presented in Kleene's *Mathematical Logic*, is an illusion.

3.3. Synopsis

We close this overview with a synopsis aimed at readers thoroughly familiar with Kleene's computability theory. Those taking a first pass through the paper may comfortably skip it.

On a comparison between Cantor's diagonal argument and Kleene's "proof," we posit that Cantor does not rely on any specific ordering of the number-theoretic functions. Cantor proves that there are more such functions than natural numbers. For his reductio, assume an ordering of them—any ordering—say, $\phi_0, \phi_1, \phi_2, \ldots$, and then define the function

$$\psi(n) = \phi_n(n) + 1.$$

The function $\psi()$ cannot be among the ordered functions. Change the ordering, and a function with the same property pops up again. Ergo, no mapping from natural numbers to all number-theoretic functions exists.

Compared to Cantor's line of reasoning, Kleene's 1967 proof introduces a subtle but crucial difference: Kleene's choice of notation for numbers (and consequently, Gödel numbers and codes for TMs too) appears essential to the incomputable function constructed. This stems from Kleene's bijection between partial recursive functions and TMs, which proceeds by first fixing an ordering of TMs and then assigning partial functions to them. The resulting incomputable function $\psi()$—defined via the predicate $T(i, a, x)$—arises *specifically* from Kleene's representational conventions R_K and C_K.

The fact that it is this specific function $\psi()$—a semantically independent Platonic function that, in itself, is not a "function" of any method of representation of it—that comes out incomputable depends on the specific way Kleene represents natural numbers. Adopt one of infinitely many other (in fact, uncountably many other) methods on how to represent natural numbers, and $\psi()$ might turn out to be computable. Of course, another function will come out incomputable relative to the new method, but does this matter? For the lab director, the answer is no. His goal is to obtain an effective method to compute $\psi()$.

To recapitulate, the set of ordered pairs that corresponds to the Platonic function $\psi()$ remains the same no matter how we choose to refer to the function(s) that generate it. The implication is that the lab director is free to endorse another method for representing natural numbers and hopefully effectively compute $\psi()$.

In retrospect however, is there no caveat to the line of reasoning just presented? Recall that Kleene's normal form theorem states that every partial computable function can be expressed using primitive recursive predicates together with one application of the Minimization operator. Might, then, Kleene's normal form theorem not undermine the preceding discussion? After all, the class of computable functions is rigorously defined, is it not? Yet we appear to be suggesting that Minimization could make the very same function computable under one notation while incomputable under another.

To appreciate why there is no caveat, it is essential to recognize that Kleene's normal form theorem—and, more broadly, his entire notion of computability—is formulated *within* his framework of tally-based TMs. Although theorists frequently invoke the normal form theorem to justify sweeping claims about Turing computation, such claims tacitly depend on an elision between Church's and Turing's theses: the natural number n is silently identified with its tally representation of $n + 1$ tallies. In other words, the result concerning Minimization does not apply uniformly across the uncountably many possible notations. Thus, any preference for a specific countable subclass of notations over others requires an explicit declaration. This declaration—often tacitly assumed by logicians—ought to be clearly acknowledged by our critics when pressed, and made explicit from the outset of any *reductio ad absurdum* argument concerning Kleene-style incomputability.

Remark 10. *Our position aligns with, and extends, San Mauro's exposition*[12]*. Although we endorse Cantorian mathematics in this article, it should be noted that Jacquette's critique*[9] *extends even to Cantor. We are currently not in a position to engage with Jacquette's arguments in technical detail.*

4. Shapiro (1982)

Shapiro's research interests overlap with the distinction between theoretical computability and physical computability. His concern is that "mechanical devices engaged in computation and humans following algorithms do not encounter numbers themselves, but rather physical objects such as ink marks on paper"[7] (p. 14). Shapiro continues by asserting that a distinction must be made between natural numbers (semantics) and their string representations (syntax):

> [S]tricly speaking, computability applies only to string-theoretic functions and not to number-theoretic functions. That is, a string-theoretic function is said to be computable iff there is an algorithm that computes it[7] (p. 14).

We discuss Shapiro's definitions in Section 4.1, some of his theorems in Section 4.2 and Section 4.3, and scrutinize his notion of "knowing a notation" in Section 4.4. We pivot from Shapiro to Kleene in Section 4.5. Despite our various critical comments from an engineering perspective, we regard Shapiro's theory-building as an exemplar par excellence.

4.1. Buildup

An example of notation is stroke notation, in which a numeral $\underline{n}$ is a finite sequence of strokes, i.e., a string of strokes. It denotes the natural number, n, of strokes it contains. Shapiro defines a "notation d" with Definition 1. To begin our discussion, we emphasize a seemingly innocuous adjective in bold.

Definition 1. *A notation d consists of a finite alphabet, a* ***solvable*** *class of strings on this alphabet, called the class of numerals, and a convention which assigns to each numeral x a natural number $d\ x$, called the denotation of x. Shapiro[7] (p. 14), our boldface.*

While Shapiro's current buildup aligns with the mathematics used in many engineering disciplines, we contend—merely for the sake of argument—that its applicability to modern internet encryption may be limited.

Example 1. *The internet company Cloudflare believes it generates random strings via a physically indeterminate process, involving lava lamps. A news update from Cloudflare reads as follows:*

> *As one might expect, lava lamps are consistently random. The "lava" in a lava lamp never takes the same shape twice, and as a result, observing a group of lava lamps is a great source of random data. To collect this data, Cloudflare has arranged about 100 lava lamps on one of the walls in the lobby of the Cloudflare headquarters and mounted a camera pointing at the lamps. The camera takes photos of the lamps at regular intervals and sends the images to Cloudflare servers. All digital images are really stored by computers as a series of numbers, with each pixel having its own numerical value, and so each image becomes a string of totally random numbers that the Cloudflare servers can then use as a starting point for creating secure encryption keys[50].*

Problem 1. *Internet cryptographers at Cloudflare do use a class of strings from a finite alphabet that, arguably, is* ***not solvable****—at least within our evolving understanding of what 'solvable' means according to Shapiro. As the previous example illustrates, the cryptographers cannot effectively enumerate any non-zero number of strings a priori; that is, without first carrying out the corresponding enumeration in the real world of lava lamps (i.e., a posteriori).*

The translation—from n strokes to the random string generated by the n-th lava lamp, with $n \leq 100$—can only be provided after the physical process at hand has been carried out on a specific day at Cloudflare. A cryptographer can *describe* this translation, but cannot *prescribe* it, with one of countably many predefined Chomskian automata.

Also, when we allow the parameter n to be replaced with *any* natural number, thereby embracing an infinite abstraction akin to that of TMs, Cloudflare activity on any given day remains indeterminate until the day concludes. A precise symbolic description can only be established once the process in question has actually been carried out.

While at least one of countably many stroke-based TMs can both prescribe and describe the addition of any two (properly encoded) numbers, or compute the shortest path in any given (finite) graph, the same does not hold in general for Cloudflare's activities related to lava lamps. This is why we assert that Cloudflare operates with a class of strings from a finite alphabet that remains unsolvable within our reading of Shapiro's stipulations.

Putting aside Shapiro's innocuous adjective "solvable" and Problem 1 for now, let us continue to follow Shapiro's theory building. His next step is to define a number-theoretic function f_d based on notation d and to establish a bijection between numerals and numbers.

Definition 2. *Let S be the numeral class for a notation d; let $\mathbb{N}$ be the class of natural numbers. Each string-theoretic function $f\ : S \rightarrow S$ has a number-theoretic counterpart $f_d : \mathbb{N} \rightarrow \mathbb{N}$ relative to d, such that $f_d(d\ x) = d\ f(x)$. Shapiro[7] (p. 15).*

"Notice that if the convention of d is not one to one, then f_d may not be well-defined and that if the convention of d is not onto then f_d may not be defined at every number. *It is assumed, therefore, that a notation convention is a bijection* from S to $\mathbb{N}$."—Shapiro[7] (p. 15, our emphasis).

Alas, the professional engineer will, more often than not, want redundancy built into the notation, resulting in the following problem.

Problem 2. *A natural number n should have more than one string representation, rendering the notation d non-bijective. This fault-tolerant property, which prevails in Shannon's information theory and coding theory in particular, counters Shapiro's assumption.*

However, in this paper, we shall fully adhere to Shapiro's assumption that a bijection is operational. Therefore, we set aside Problem 2 as well. Consequently, we can appreciate Shapiro's next move in which he introduces the *inverse* of notation d, as follows:

Definition 3. *Let $\overline{d} : \mathbb{N} \to S$ be the inverse of the convention of d. If $F : \mathbb{N} \to \mathbb{N}$ is a number-theoretic function, let $F^d : S \to S$ be its string-theoretic counterpart relative to d; that is: $F^d(\overline{d}\, X) = \overline{d}\, F(X)$. Shapiro[7] (p. 15).*

All of this effort is directed towards achieving mathematical elegance (in a non-pejorative sense) in the form of two commuting diagrams, presented in the following figure, which is of our making.

$$\begin{tikzcd}
d\,x \arrow[r, "f_d"] & \surd \\
x \arrow[u] \arrow[r, "f"] & f(x) \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[r, "F"] \arrow[d] & F(X) \arrow[d] \\
\overline{d}\,X \arrow[r, "F^d"] & \surd
\end{tikzcd}$$

The top row represents natural numbers, while the bottom row shows their corresponding numerals.

Accompanied by Definition 4, Shapiro's endeavor allows for the derivation of powerful theorems in the next section.

Definition 4. *If F is a number-theoretic function, we say that F is computable relative to d iff the string-theoretic F^d is computable—iff there is a string-theoretic algorithm that computes F^d. Shapiro[7] (p. 15).*

4.2. Theorem Proving

When it comes to his theorems, Shapiro initially makes three insightful observations. First, each *constant* function is computable relative to every notation. Second, the *identity* function is computable relative to every notation. Third, each function that differs from one of these at only a finite number of arguments is computable relative to every notation. Shapiro then proves the completeness of this list, culminating in:

Theorem 1. *The only number-theoretic functions which are computable relative to* ***every*** *notation are almost constant functions and almost identity functions. Shapiro[7] (p. 15), our emphasis.*

Theorem 1 aligns with what we expect will be the intuitive position for many readers. We include it here to ensure comprehensive treatment of the subject matter. More interesting for the purpose of this paper is Shapiro's second theorem (Theorem 2) and the subsequent commentary, which leads to his third theorem (Theorem 3).

Theorem 2. *Let F be a number-theoretic function. There is a notation d such that F^d is computable iff there is a permutation T of the natural numbers such that $T^{-1}\ F\ T$ is computable relative to stroke notation. Shapiro[7] (p. 17).*

"If one accepts *Church's thesis in the form* (similar to that given by Turing) that a number-theoretic function is recursive iff it is computable relative to stroke notation, then Theorem 2 can be more concisely formulated as Theorem 3."—Shapiro[7] (p. 17), our emphasis.

Theorem 3. *There is a notation d such that F^d is computable iff there is a permutation T such that $T^{-1}\ F\ T$ is recursive. Shapiro[7] (p. 17).*

Starting our discussion with Theorem 2, it is important to note that uncountably many permutations T are under consideration, meaning there are correspondingly uncountably many notations d. The acceptance of Church's thesis in the extended form, which brings us to Theorem 3, does not collapse the uncountable setting into a countable arena of mathematical discourse. Instead, it highlights stroke notation as one viable way for us, theorists and mathematical engineers alike, to bridge the semantic realm of number-theoretic functions and the syntactic realm of string-theoretic functions. For further reference, we present Shapiro's extension of Church's thesis as a separate definition:

Definition 5. *The extended version of Church's thesis states that, a number-theoretic function is recursive iff it is TM computable relative to stroke notation.*

4.3. Towards Incomputability

As we now examine Shapiro's brief exposition on incomputability, we convey the following result:

Theorem 4. *There is a number-theoretic function which is not computable relative to any notation. Shapiro*[7] *(p. 18).*

Concerning Shapiro's proof of Theorem 4, we only distil a rigorous part that matters to us as an exemplar for the remainder of this paper. To start, Shapiro provides a simple definition and a trivial lemma, as follows.

Definition 6. *For each number-theoretic function F, let $S(F)$ be the set of natural number $\{n \mid n \neq 0$ and there is a natural number which has exactly n preimages under F$\}$. Shapiro*[7] *(p. 18).*

Lemma 1. *If T is a permutation of the natural numbers, then $S(F) = S(T^{-1}\, F\, T)$. Shapiro*[7] *(p. 18).*

Shapiro's proof of Theorem 4 relies, once again, on the extension of Church's thesis:

> (*) "Note also, by an extended version of Church's thesis, that if F is computable relative to stroke notation then $S(F)$ is recursively enumerable in the halting problem."—Shapiro[7] (p. 18).

The extension of the thesis is crucial, allowing Shapiro to swiftly transition from stroke notation (syntax) to natural numbers (semantics), and back again. Subsequently, Shapiro takes the contraposition of (*) and uses his Theorem 2 to derive the following implication, which we present as a separate lemma for later reference in this paper.

Lemma 2. *If one accepts the extended version of Church's thesis (see Definition 5), then we have: if $S(F)$ is not recursively enumerable in the halting problem, then F is not computable relative to any notation. Shapiro*[7] *(p. 18).*

In contrast to Shapiro-style accounts, which are crafted with nuance, we will argue in Section 5 and beyond that Kleene-style narratives fail to *consistently* mention Church's thesis or an extended version of it, in otherwise similar discourse pertaining to incomputability.

4.4. Knowing a Notation

After having proved multiple theorems in an uncountable setting, Shapiro remarks that several notations considered so far are "clearly … not acceptable." Therefore, "further restrictions must be placed on notations," which, as we shall see shortly, provide for a countable arena of discourse. Shapiro expects "the computist" to be able "to write" and "to read" numbers in the notation at hand. If this is not the case, then we cannot speak of an "acceptable notation"[7] (p. 18).

Considering our example of the lava lamps once more, Shapiro's following stipulation regarding the "use" of the notation does not sit well with us.

> "If a computist does not know a particular notation, then it is hard to see a sense in which she understands the number-theoretic goal of an algorithm that employs the notation. If, for example, an algorithm for addition uses a notation which is not known by a computist and she is given two numerals in the notation, she could not know that the algorithm determines the sum of the denoted numbers. That is, the computist could not *use* the algorithm to add numbers."—Shapiro[7] (p. 19, our emphasis).

A cryptographer might initially adopt stroke notation (or, more realistically, binary notation), which she can indeed "use" in Shapiro's sense. However, her immediate goal is to swiftly transition, leveraging the inherent unpredictability of her lava lamps, from original strings (of strokes) to random strings (of strokes). Subsequently, she applies an algorithm for addition to various strings, including some of her random ones. We submit that the cryptographer does not fully "know" what she is doing based on our analytical reading of Shapiro's postulates on "knowing a notation." However, Shapiro's proponents might argue that the cryptographer retains the ability to manipulate all her strings, understanding their underlying meaning. Instead of attempting to refute this point, we continue reading Shapiro's paper until we find a stipulation that does not align with the practices of the cryptographer, according to both Shapiro's proponents and ourselves. This leads us to:

Lemma 3. *A person knows (or can easily be taught) notation* d *iff she knows (or can easily be taught) either an effective procedure to translate* d *into stroke notation or an effective procedure to translate stroke notation into* d*. Shapiro*[7] *(p. 19).*

Lemma 3 is part of Shapiro's repertoire of stipulations, which, in our understanding, serves to collapse the uncountable setting of "notations" into a countable arena of "acceptable notations." We view Shapiro's appeal to Lemma 3 as a technically apt way of endorsing a strong version of logical determinacy: an individual acquires new notation if and only if she comes to possess an effective—i.e., a logically determinate—procedure that takes her to or from the familiar land of strokes, where n strokes denote the number n. We hypothesize that Shapiro's computational framework of human inventiveness qua notation starkly contrasts Alan Turing's perspective[36].

At Cloudflare, the translation from n strokes to the random string generated by the n-th lava lamp is due to an uncountable-countable collapse: from analogue (lava) to digital (strings). The central problem lies in the absence of a symbolic prescription—i.e., an effective procedure in Shapiro's sense—to anticipate what a physical computation of the n-th lava lamp would entail if it were to be actually carried out. We believe that Shapiro's proponents concur with us on the following point: The cryptographer knows her lava-generated notation—i.e., she knows the translation from n strokes to the random string generated by the n-th lava lamp *after* the process has been carried out—but not in adherence to Lemma 3.

4.5. Pivoting from Shapiro to Kleene

More importantly, even regarding engineering practices totally unrelated to cryptography, such as Catthoor et al.'s work on inventing energy-efficient data structures for multimedia applications running on hand-held devices[1,2,48], we interpret Shapiro's theory as oversimplified. While symbolic logic may encompass the dos and don'ts of routine engineering procedures, it falls short in an uncountable world of professional engineering. The theorist must first observe specific engineering choices within a given spatiotemporal context, made by creative individuals and high-tech equipment, before proving theorems whose prescriptive force applies, without further contemplation, only within that context.

Remark 11. *In our reading, Ethan Brauer puts it, in a slightly adjacent setting, as follows: "What can be done efficiently depends on the technology available, and it is not a failing of the theory to recognize that dependence"*[51] *(pp. 10507–10508).*

To clarify, researchers in the field of energy-efficient software development are inclined to explore parts of a panorama of uncountably many notations. This includes various listings, such as $\{M_i\}_{i>0}$ and $\{N_k\}_{k>0}$. While the first listing fits in Shapiro's stroke-based account of TMs M_1, M_2, …, the second listing fits in, say, a narrative in which numbers are represented with bits.

Each member in a listing is a program text (i.e., a machine specification). Each arrow in the following diagram represents a sequence of consistent design choices; that is, *meta-consistent* design choices to be more precise, since we are now in the business of studying multiple ways of how to transgress the semantic-syntactic divide. Set theoretically, there are uncountably many such arrows.

$$\begin{tikzcd}
\text{specification} & & \varphi_j() \arrow[dl] \arrow[dr] & \\
\text{program text} & M_i & & N_k
\end{tikzcd}$$

Specific machines M_i and N_k are functionally incomparable in syntactic terms alone, yet they can be compared functionally at the semantical level by specialists—i.e., *humans* in the loop—who grasp the conventions of the engineering practice at hand. Both machines compute the same partial function $\varphi_j()$ in syntactically incompatible ways.

Consider, for example, a shortest path problem, $\varphi_j()$. On the one hand, this problem is implemented with a two-dimensional array data structure, cf. M_i. On the other hand, it is also implemented with a linked list data structure, cf. N_k. The crux is that M_i and N_k cannot be compared number-theoretically without human involvement, i.e., without further stipulation from the engineers. For outsiders, this stipulation always comes a posteriori because the design space of functionally equivalent data structures is uncountably large.

Some carefully selected stroke-based TM can *describe*, but cannot *prescribe*, the conversion from program text M_i into program text N_k. A theorist cannot, in advance, specify a stroke-based TM that anticipates the engineering choices Catthoor and his team will make tomorrow. To assume otherwise is misguided, as the countably many stroke-based TMs are inherently inadequate compared to the uncountably many design options available to a creative engineer.

Our broader argument stands: predefined Chomskian models of computation lack prescriptive authority. The theorist must rely on empirical feedback from engineers to bridge the gap between uncountable and countable realms of discourse.

Remark 12. *The assumption that a key activity of Catthoor's engineering—undertaken by a potentially immortal and creative human being, and partially illustrated in the previous diagram—can, in principle, be subsumed under a single stroke-based TM, reflects a neo-Russellian stance commonly encountered in theoretical computer science textbooks, a position we respectfully reject.*

We now invite the critic, if not Shapiro himself, to compare the uncountable-countable collapse observed in creative engineering with Kleene's comparatively facile line of reasoning. Kleene moves from one countable realm to another, isomorphic one; that is, from recursion theory (semantics) to Turing machinery (syntax), exemplified by the shift from a natural number n to its representation of $n+1$tallies. The following excerpt—from Kleene[5] (p. 238), our boldface—makes this clear:

> The business of finding machines can be **systematized by starting from the theory of recursive functions**. … This theory deals with recursive definitions of functions … The functions commonly used in number theory are definable by use of such recursions, and **proceeding from the recursive definition one can in a systematic way find corresponding Turing machines**, after first setting up Turing machines for such simple operations as filling in with tallies all but the rightmost of a sequence of blank squares preceded and followed by a tally, copying a sequence of tallies, etc.

Is it fair to suggest that Shapiro and Kleene share a common intellectual ground, in that they both assume, at least implicitly, that Catthoor's engineering—if not software engineering *tout court*—operates within a countable, even algorithmic, realm (Remark 12)?

Moving beyond these reflections for now, it is Shapiro's carefully crafted non-mathematical stipulations, such as those found, via Definition 5, in the precondition of his Lemma 2, that we wish to carry forward. Shapiro's rigor is precisely what we need to criticize Kleene's theory building. Our discord with Kleene is mathematical, not philosophical, and will become relevant once we enter the realm of incomputability.

5. Kleene (1967)

In his *Mathematical Logic*, Stephen Kleene distinguishes between number-theoretic functions on one side (semantics) and TMs on the other (syntax). Therefore, following Shapiro's commuting diagrams in Section 4.1, we continue to depict the first category (semantics) at the top and the second category (syntax) at the bottom in our discourse.

Instead of total functions, Kleene relies on the more general concept of a *partial* number-theoretic function. He treats each "argument" for a partial function as a natural number, which he "represents" with his "tally marks"[5] (p. 235).

Claim 1. *The terms "denote" and "represent," employed by Shapiro and Kleene, respectively, are synonymous. Likewise for the words "notation" and "representation."*

Another minor distinction between both authors is noteworthy. Recall that Shapiro uses n strokes on a tape of a TM to denote the natural number n. (He uses the null string to denote 0.) Kleene, however, uses $n+1$ tallies on a tape of a TM to represent n. By sequentially counting all $n+1$ tallies on the tape, the concept of n is then derived within Kleene's realm of number theory.

We capture Kleene's representation with a function, R_K: $\mathbb{N} \rightarrow S$, where S denotes the set of strings of tally marks. Schematically, we depict the situation, thus:

$$\begin{tikzcd}
\text{natural number} & 2 \arrow[d, "R_K"'] \\
\text{string of tallies} & |||
\end{tikzcd}$$

Remark 13. *Function R_K is only slightly different from Shapiro's $\overline{d}$: $\mathbb{N} \rightarrow S$, i.e., the inverse of the convention of Shapiro's notation d, where S denotes the set of strings of strokes (cf. Section 4.1).*

Taking a broader view, Kleene's central query is as follows:

> For what number-theoretic functions are there computation procedures (or algorithms)? Briefly: What is the class of "computable" functions?[5] (p. 230).

Although the Platonic undertones in this and other excerpts from Kleene should not go unnoticed, our concern will be with Kleene's *reductio ad absurdum* proof pertaining to incomputability. As an informal introduction, Kleene writes thus:

> This intuitive notion of a computation procedure ... is vague when we try to extract from it a picture of the totality of all possible computable functions. And we must have such a picture, in exact terms, before we can hope to prove that there is no computation procedure at all for a certain function, or briefly to prove that a certain function is uncomputable. Something more is needed for this[5] (p. 231).

What is additionally needed is the notion of a tally-based TM. To build up toward his tally-based TMs, Kleene continues:

> Hereafter, we may say that something can be done "effectively", or that an operation or process is "effective", as a brief way of saying that there is an algorithm for it (i.e. a decision or computation procedure)[5] (p. 231).

Altogether, Kleene adopts a methodical two-stage reasoning process. Initially, he shifts from natural numbers (top) to a representation based on tallies (bottom). Subsequently, he engages in a bottom-up approach, progressing from enumerating his tally-based TMs to deriving a corresponding listing of partial computable functions.

All of this will be necessary for Kleene to make a claim about incomputability at large. On the one hand, we shall peruse Kleene's reasoning with the following question in mind:

Problem 3. *Does a proof of incomputability in Kleene's countable, tally-based setting have any prescriptive implications for software engineers?*

On the other hand, we will disagree with Kleene's line of reasoning on his very own terms.

We are now ready to zoom in on Kleene's informal commentary and formal buildup. We peruse parts of Kleene's prelude (Section 5.1), and highlight his top-down (Section 5.2) and bottom-up (Section 5.3) reasoning. We examine Kleene's famous T predicate (Section 5.4), which he uses in his *reductio ad absurdum* proof (Section 5.5). Finally, we object to Kleene's incomputability proof on purely technical grounds and amend it (Section 5.6). We shall answer Problem 3 in our discussion (also in Section 5.6).

5.1. Prelude

Kleene, in 1967, and many computability theorists to date, intentionally merge the following four categories of theoretical discourse:

1. Computable partial functions
2. Lambda-definable functions
3. General recursive functions
4. Turing computable functions

In order to streamline our forthcoming critique, we will adopt Kleene's approach of merging Categories 1–3. However, we will contest Kleene's amalgamation of Categories 1 and 4.

Category 1 contains number theory (semantics), while Category 4 contains string-theoretic functions (syntax). In this regard, recall Shapiro's explicit reference—e.g., in Lemma 2—to an extended version of Church's thesis, i.e., Definition 5.

Based on Kleene's commentary on the status of Church's thesis—presented next, from[5] (p. 232)—we conclude with Claim 2.

> … Church proposed the thesis (published in 1936) that all functions which intuitively we can regard as computable, or in his words "effectively calculable", are λ-definable, or equivalently general recursive. This is a thesis rather than a theorem, in as much as it proposes to identify a somewhat vague intuitive concept with a concept phrased in exact mathematical terms, and thus is not susceptible of proof. But very strong evidence was adduced by Church, and subsequently by others, in support of the thesis.

Claim 2. *On Kleene's own terms, any reliance on Church's thesis renders the argument at hand non-mathematical.*

5.2. Top-Down

Coming to Kleene's top-down transition, from natural numbers to tally marks on the tape of a TM, we emphasize specific words in bold to accentuate the human element of the mathematical engineer within his theory building:

> **[W]e must agree** how the argument(s) … are to be **represented** on the tape, and how the machine **is to give us** the resulting value of the function. We shall **make the supposition** that all machines to be considered have among their symbols the tally mark "|"; say it is s_1. **We shall represent** natural numbers **by** sequences of tallies, "|" for 0, "||" for 1, "|||" for 2, … To set up the machine and tape to compute for a given argument a, **we shall arrange** that: at the moment 0 the system consisting of a machine and tape is started off so that the leftmost square of the tape is blank, a is **represented** by tallies on the next $a + 1$ squares, …

Continuing from the previous excerpt[5] (pp. 234–235, our boldface), Kleene writes[5] (p. 235, original emphasis) thus:

> In this situation, we say the machine is *applied to a as argument*. We say the machine *computes a value c for a as argument*, if, starting from this situation at Moment 0, the machine at some later moment assumes the passive state 0 ("stops") with a blank and $c + 1$ tallies printed on the tape after the $a + 1$ tallies representing the argument a, the tape being otherwise blank, …
>
> A given machine may compute a value for each natural number a as argument, or for some a's but for others, or for no a's. If, for each a, it computes a value c where $c = f(a)$, we say that the machine *computes* the function $f(a)$, and that $f(a)$ is *Turing computable*.

The crux is that Kleene blends natural numbers (semantics) and tallies (syntax), which explains why Church's thesis and Turing's thesis are the same for him.

To accommodate Kleene's 1967 exposition and to compare with Shapiro's extended version of Church's thesis (Definition 5), which refers to stroke notation, we introduce an alternative formulation using tallies:

Definition 7. *The extended version of Church's thesis states that, a number-theoretic function is recursive iff it is TM computable relative to Kleene's convention, R_K, which is based on tally marks. (Recall Remark 13.)*

Additionally, Kleene posits the "fact" that "we get no larger class of computable functions" when contemplating TMs that utilize more symbols than merely the tally mark[5] (p. 238). Kleene refers to Chapter XIII in his 1952 treatise[52] for the mathematical proof. However, here too, Kleene is implicitly yet crucially depending on an assertion (Definition 7), which is non-mathematical (Claim 2). The implication is now clear:

Claim 3. *Shapiro in 1982 explicitly, and Kleene in 1967 implicitly, rely on an extended version of Church's thesis (see Definition 5 and Definition 7, respectively) in their com-*

putability theory. Their outlook on software engineering is countable by design; in fact, it is even algorithmic.

Our claim is unsurprising, given that Shapiro addresses precisely this point, albeit without mentioning Kleene in particular. From Claims 2–3, we also infer:

Claim 4. *In retrospect, Kleene's 1967 computability theory is, on his own terms, not purely mathematical.*

We posit that Claim 4 is well known among philosophers of logic, though perhaps not exactly as we present it.

5.3. Bottom-Up

Kleene reasons from his tally-based TMs back to number theory. In this regard, he introduces another convention, C_K, which allows him to recast each Turing machine (say, M_i) with a unique machine index (say, i).

Claim 5. *With Kleene's convention R_K for tallies fixed from the outset, numerous methods exist to consistently generate a machine index for a function, such as the successor function $f(a) = a + 1$. Kleene's specific design choices, which lead to his particular outcome—index i—constitute only one coding method, C_K, among several possible methods.*

To substantiate Claim 5, we present Kleene's own desiderata, which rely on a Turing machine G that computes the successor function[5] (pp. 236–237). The letter "G" presumably stands for "Gödel," given that the following exposition is reminiscent of Gödel coding[5] (pp. 242–243, original italics, our boldface):

> We have seen that the pattern of behavior of a given Turing machine is determined by the table for it …
> Rewriting the table for our machine G in this manner, it becomes:
>
> | Machine state | Scanned square condition | |
> |---|---|---|
> | | 0 | 1 |
> | 1 | $0C0$ | $1R2$ |
> | 2 | $0R3$ | $1R9$ |
> | 3 | $1L4$ | $1R3$ |
> | … | … | … |
> | 10 | $0C0$ | $0R11$ |
> | 11 | $1C0$ | $1R11$ |
>
> The table for a machine **can** be written in code form. Consider the table for G … **let us** insert semicolons at the end of each row of entries, and commas separating entries within a row, and then **string the entire body of the table along as one sequence** of symbols:
>
> $$0C0, 1R2; 0R3, 1R9; 1L4, 1R3; \\ \ldots; 0C0, 0R11; 1C0, 1R11$$
>
> This sequence of symbols is the *code* for the machine G.
> The code for any machine [**relative** to the chosen **representation**] **can** thus be written on a typewriter with the following 15 symbols:
>
> $$L\ C\ R\ ,\ ;\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$
>
> **Such a code** does not begin with the symbol L. **By reinterpreting** these symbols [**relative** to the chosen **representation**] as the digits of a number in the number system based on 15, **we get** a positive integer **which describes [to us]** the machine table and thence the pattern of behavior of the machine; call this number the *index* of the machine...

Once again, we see Kleene blending Turing machinery (syntax) and number theory (semantics), but this time he does so via his concocted 15-symbol system and his other design choices. He chooses, just like Gödel, one specific coding convention, C_K, instead of entertaining, like the creative engineer, a family of two or more conventions.

Hinging upon his conventions R_K and C_K, Kleene is now finally in a position to define a listing of his TMs, $\{M_i\}_{i>0}$, accompanied by the assertion that each machine M_i computes a partial function $\varphi_i()$. Schematically, we depict the situation thus:

$$\begin{array}{lcccc} & \varphi_1 & \varphi_2 & \varphi_3 & \ldots \\ R_K \text{ and } C_K & | & | & | & \\ & M_1 & M_2 & M_3 & \ldots \end{array}$$

Fact 1. *Relying on conventions R_K and C_K, Kleene defines a listing $\{M_i\}_{i>0}$ of his TMs and, next, a corresponding listing $\{\varphi_i\}_{i>0}$ of his computable partial functions; that is, each machine M_i computes a partial function $\varphi_i()$.*

All of this effort aims to achieve mathematical elegance by creating a mirror image between functions (semantics) and machines (syntax). Unfortunately, this pursuit will ultimately prove mistaken.

5.4. Kleene's T Predicate

With conventions R_K and C_K fixed at the outset, Kleene defines his famous $T(i, a, x)$ predicate in 1. and his specific number-theoretic function $\psi()$ in 2. as follows:

1. i is the index of a Turing machine (call it "Machine M_i") which, when applied to a as an argument, will at Moment x (but not earlier) have completed the computation of a value (call that value "$\varphi_i(a)$").
2. $$\psi(a) = \begin{cases} \varphi_a(a) + 1 & \text{if } (Ex)T(a, a, x) \\ 0 & \text{otherwise} \end{cases}$$

These two items are quoted respectively from pages 243 and 245 in Kleene[5].

Observe that argument a and moment x are natural numbers and that machine indices i and a are Gödel numbers in particular. More importantly, note that Kleene evaluates his $T(i, a, x)$ predicate in 1. above as either *true* or *false,* subject to the proviso that machine index i is interpreted in adherence to R_K and C_K.

For instance, in contrast to a statement such as Pythagoras' theorem, the predicate $T(100, 200, 300)$ may hold *true* in Kleene's work but could be *false* in a treatise in which TMs operate based on bits instead of tallies. The implication is that $\psi(a)$, defined above in 2., strongly depends on R_K and C_K, even though these syntactic conventions are not mentioned.

This brings us to the elephant in the room, namely, the very idea that semantics and syntax are 100% separable:

Claim 6. *Kleene's predicate semantics hinges on specific choices pertaining to his syntax. The $T(i, a, x)$ predicate does not convey any meaningful information unless specific syntactic conventions, such as R_K and C_K, are consistently employed for the natural numbers at hand.*

Using $T(i, a, x)$ in theoretical discourse requires acceptance of an extended version of Church's thesis (Definition 7). In Kleene's 1967 exposition, this dependence specifically amounts to embracing syntactic conventions R_K and C_K.

5.5. Reductio ad Absurdum

Reaching the climax, Kleene then proves, via his listing $\{M_i\}_{i>0}$ of TMs and by *reductio ad absurdum*, that his concocted function $\psi()$ is not computable. He writes:

Theorem 5. *The function $\psi(a)$ defined [in 2. above] is not computable. Kleene[5] (p. 245).*

Kleene's proof—which, in modern terms, demonstrates the incomputability of the Halting Problem—proceeds as follows:

Proof. Suppose $\psi(a)$ were computable; say, machine M_p computes it, so that $\psi(a) = \varphi_p(a)$ for all a. Substituting p for a, we have: $\psi(p) = \varphi_p(p)$. But since M_p computes $\psi(a)$, we have, for all a, $(Ex)T(p, a, x)$, and in particular $(Ex)T(p, p, x)$. Using this in the definition of $\psi(a)$, we obtain: $\psi(p) = \varphi_p(p) + 1$. The two displayed equations contradict each other. □

This proof is quoted almost literally from Kleene[5] (p. 245), but we have added the words "we have" and "we obtain" to improve the readability.

Consequently, Kleene concludes that $\psi()$ is incomputable.

5.6. Anti-Climax

Our disagreement is that Kleene's proof solely establishes the incomputability of $\psi()$ within his practice of R_K and C_K. Recall Claims 4–6. We therefore propose to amend Kleene's Theorem 5 as follows:

Theorem 6. *The function $\psi(a)$ defined in 2. above is not computable relative to Kleene's practice, which is characterized by his conventions R_K and C_K.*

To fully appreciate our amendment (Theorem 6), consider one of the uncountably many antagonists of Kleene. Let us refer to her as Antagonist A. Suppose that Antagonist A favors using bits to represent numbers on the tape of a TM. She therefore chooses her own representation, denoted as R_A, in contrast to Kleene's tally-based representation R_K. Her coding convention, C_A, naturally differs from Kleene's convention C_K.

Consequently, she establishes an alternative listing

$\{N_k\}_{k>0}$ of machines, accompanied by a corresponding listing $\{\rho_k\}_{k>0}$ of partial functions. Each TM N_k, based on the binary system, computes some partial function $\rho_k()$, meaning that each $\rho_k()$ is computable relative to conventions R_A and C_A.

$$\begin{array}{ccccccc} & \rho_1 & \rho_2 & \rho_3 & \cdots & \rho_{1241} \text{ is } \psi & \cdots \\ R_A \text{ and } C_A & | & | & | & & | & \\ & N_1 & N_2 & N_3 & \cdots & N_{1241} & \cdots \end{array}$$

Besides deriving some incomputable function relative to representation R_A and coding C_A, it is conceivable that some machine, say N_{1241}, computes Kleene's own "incomputable" function $\psi()$ relative to R_A and C_A. This particular possibility is not invalidated by Kleene's proof of Theorem 5, as presented earlier. The crux is as follows:

Claim 7. *Kleene only proves incomputability relative to his practice (Theorem 6) instead of practice-independent incomputability (Kleene's alleged Theorem 5).*

Kleene mistakenly believes that his semantics is entirely divorced from his syntactic choices. He firmly embeds his proclaimed equivalence between Church's thesis and Turing's thesis within his meta-mathematics, as exemplified by the mirrored relationship between his listing of all computable partial functions and his listing of all tally-based TMs in his proof of Theorem 5. To be technically correct, he should mention, like Shapiro, his dependence on an extended version of Church's thesis (Definition 7). Likewise, by projecting a similar mirrored image onto the Antagonist's machine listing (a common reaction among computer scientists upon encountering our critique), Kleene or his protagonist inevitably invokes an extended version of Church's thesis (Definition 7) yet again. Alas, Kleene does not mention this assumption anywhere in his proof. To be precise, he fails to acknowledge his reliance on R_K and C_K, thereby rendering his semantics ambiguous.

Let us now revisit Problem 3: Does a proof of incomputability in Kleene's countable, tally-based setting have any prescriptive implications for software engineers? The answer is affirmative if the engineers rigorously follow Kleene's facile reasoning, adhering strictly to his practice (cf. R_K and C_K). However, in even slightly less routine settings, such as combining R_K and C_K with R_A and C_A, the answer is negative.

For Kleene's incomputability theorem to hold practical relevance, the engineer must adhere to the precision and consistency characteristic of a tally-based automaton, rather than relying on the creativity and skill prevalent in industry. Certainly, a theorist can retrospectively *describe* the do's and don'ts of a combined practice, like integrating R_K and C_K with R_A and C_A. However, it is inherently impossible to a priori *prescribe* an arbitrary engineering practice, considering the uncountably many possibilities that exist.

6. Conclusions

The director of a computing laboratory is unlikely to embrace Kleene's perspective on incomputability unless he believes that the mind of an engineer functions like a tally-based Turing machine. While such a belief is admissible, Kleene did not explicitly state it as an assumption in his *reductio ad absurdum* proof. Given that many engineers reject this assumption, even an amendment to Kleene's proof lacks the societal significance that Kleene attributes to it.

For several engineers, notation pertaining to computability in the real world *is* context-dependent. Consequently, they typically disagree with Shapiro's 2017 sentiment:

> It would, however, be unfortunate if 'acceptable notation' itself were context-sensitive and interest-relative. That would make computability, over numbers, a context-sensitive and interest-relative matter[53] (p. 276).

The crux is that engineers are hesitant to accept a unifying thesis without scrutiny from industry; while mathematical aesthetics has a role, it may not come at the expense of modeling the real world. In this regard, our findings relate to Brauer[51]. We have complemented his philosophical critique by delving into the uncountable realm of software engineering and challenging Kleene's mathematics as such.

Mathematically, Kleene aimed to establish a theorem concerning an incomputable function independent of human involvement, such as how to represent numbers on a Turing machine tape. Alas, Kleene's proof applies only within the context of his tally-based representation, R_K, and his coding convention, C_K. To address this limitation, it is necessary to introduce an extended version of Church's thesis as an additional assumption, alongside Kleene's initial assumption that the function in question is computable. Consequently, a corrected version of Kleene's *reductio ad absurdum* proof only demonstrates that at least one of these two assumptions is incorrect, rather than proving that the function is necessarily incomputable.

Although Kleene, like Shapiro, would likely have accepted the additional assumption as valid, it is important to note that he believed he was reasoning solely within the realm of mathematics, with references to Church's thesis arising only in the aftermath. He was unaware that his chain of reasoning inherently relied on Church's thesis from the outset. Since Church's thesis is non-mathematical by Kleene's own terms, it follows that both his proof and our amendment to it are also non-mathematical.

Funding

This work received no external funding.

Institutional Review Board Statement

Not applicable.

Informed Consent Statement

Not applicable.

Data Availability Statement

Not applicable.

Acknowledgments

In addition to the two anonymous reviewers, the author—also known as Karel Van Oudheusden in Flemish circles—thanks Doukas Kapantaïs, Erhard Schüttpelz, and Simone Martini for their discussions of (early) drafts of this paper over the past nine years. He gratefully acknowledges Kapantaïs for preparing a draft of the synopsis in Section 3.3 and he thanks Edward A. Lee for bringing the case study of the lava lamps to his attention.

The technical findings in this article are dedicated to the logician Dick de Jongh and may well have been anticipated, in one form or another, by Ludwig Wittgenstein, Paul Finsler, Dale Jacquette, James R Meyer, Nicholas Swenson, and other critical thinkers. No claim to priority—nor to finality—is made in identifying inconsistencies in the foundations of computability theory. Further discussions and errata appear on: https://www.dijkstrascry.com/Kleene.

Conflicts of Interest

The author declares no conflict of interest.

Appendix A. Descriptive vs. Prescriptive

In discussing the attribution of *law*, *symbolic logic*, or *program text* in prescriptive and descriptive terms, the author offers a three-part classification in his recent Think Piece [54]:

1. **Governing laws**—Prescriptive but not necessarily descriptive; these are authoritative recommendations or rules intended to guide behavior, though they are not necessarily followed in the real world.
2. **Natural laws**—Both prescriptive and descriptive; here symbolic representations not only prescribe what ought to occur but also accurately reflect what does occur in the physical world.
3. **Historical trends**—Primarily descriptive and, at most, secondarily prescriptive; they characterize patterns observed over time, with any prescriptive force arising only retrospectively or incidentally.

Different historical figures have assigned symbolic logic to distinct conceptual categories. The author suggests that Eddington regarded logic as falling under the first category—governing laws. Turing, at least temporarily and particularly in his exchanges with Wittgenstein, aligned with Russell's intellectual position and treated symbolic logic as belonging to the second category—natural laws. Wittgenstein, in contrast, rejected both of these views, arguing that symbolic logic should instead be understood as part of the third category—historical trends.

Some anonymous readers of a draft version of this paper do not clearly distinguish between the prescriptive and the descriptive, frequently treating them as interchangeable. As a result—and consistent with what the author identifies as the mainstream view in computer science—they implicitly adopt the second category: symbolic logic is seen as both prescriptive and descriptive, the Church-Turing Thesis is effectively treated as a natural law (cf. the daughter, Helena, in the *Exemplum*), and no substantial distinction is drawn—at least in principle—between functional programming and superficially similar approaches within software engineering.

The standard position rests on an implicit, unacknowledged assumption—a *neo-Russellian tenet*: the belief in an isomorphism between symbolic logic and the essence of physical reality, between a Chomskian automaton and the essence of what it is to be a creative human engineer (see Remark 12), or—closely related—between notation-independent lambda-definable functions and notation-dependent Turing machines. This latter conflation, between one mathematical realm and another one, amounts to an equivocation between Church's Thesis and Turing's Thesis, an elision notably made by Kleene in 1967 and by several theoretical computer scientists today.

References

[1] Daylight, E.G., Demoen, B., Catthoor, F., 2004. Formally Specifying Dynamic Data Structures for Embedded Software Design: An Initial Approach. Electronic Notes in Theoretical Computer Science. 108, 99–112.

[2] Katsaragakis, M., Baloukas, C., Papadopoulos, L., et al., 2025. Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems. ACM Transactions on Architecture and Code Optimization. 22(2), 80.

[3] Pilquist, M., Bjarnason, R., Chiusano, P., 2023. Functional Programming in Scala, 2nd ed. Manning Publications: New York, NY, USA.

[4] Voeten, J., 2001. On the Fundamental Limitations of Transformational Design. ACM Transactions on Design Automation of Electronic Systems. 6(4), 533–552.

[5] Kleene, S.C., 1967. Mathematical Logic. John Wiley and Sons: New York, NY, USA.

[6] Naur, P., 1985. Programming as Theory Building. Microprocessing and Microprogramming. 15, 253–261.

[7] Shapiro, S., 1982. Acceptable Notation. Notre Dame Journal of Formal Logic. 23(1), 14–20.

[8] Daylight, E.G., 2011. Pluralism in Software Engineering: Turing Award Winner Peter Naur Explains. Lonely Scholar: Heverlee, Belgium.

[9] Jacquette, D., 2004. Diagonalization in Logic and Mathematics. In: Gabbay, D.M., Guenthner, F. (Eds.). Handbook of Philosophical Logic. Springer: Dordrecht, Netherlands. pp. 55–147.

[10] Jacquette, D., 2014. Computable Diagonalizations and Turing's Cardinality Paradox. Journal for General Philosophy of Science. 45(2), 239–262.

[11] Van Bendegem, J.P., 2012. A Defense of Strict Finitism. Constructivist Foundations. 7(2), 141–149.

[12] San Mauro, L., 2018. Church–Turing Thesis, in Practice. In: Pulcini, G., Piazza, M. (Eds.). Truth, Existence and Explanation: FILMAT 2016 Studies in the Philosophy of Mathematics. Springer: Cham, Switzerland. pp. 225–248.

[13] Andrews, U., Belin, D.F., San Mauro, L., 2023. On the Structure of Computable Reducibility on Equivalence Relations of Natural Numbers. Journal of Symbolic Logic. 88(3), 1038–1063.

[14] Quinon, P., 2021. Can Church's Thesis Be Viewed as a Carnapian Explication? Synthese. 198 (Suppl 5), 1047–1074.

[15] Quinon, P., 2025. Intensional Differences between Programming Languages: A Conceptual and Practical Analysis. Philosophies. 10, 129.

[16] Stephanou, H., 2025. Systems, Machines, and Problem-Solving. De Gruyter: Berlin, Germany.

[17] Lucas, S., 2021. The Origins of the Halting Problem. Journal of Logical and Algebraic Methods in Programming. 121, 100687.

[18] Burgin, M., 2005. Super-Recursive Algorithms. Springer: New York, NY, USA.

[19] Daylight, E.G., 2024. Refining Mark Burgin's Case against the Church–Turing Thesis. Philosophies. 9(4), 122.

[20] Petri, C.A., 1966. Communication with Automata. Technical Report RADC-TR-65-377. Rome Air Development Center: New York, NY, USA.

[21] Smith, E., 2015. Carl Adam Petri: Life and Science. Denir, T. (Trans.). Springer: Heidelberg, Germany.

[22] Daylight, E.G., 2025. Injecting Observers into Computational Complexity. Philosophies. 10(4), 76.

[23] Daylight, E.G., Cardone, F., 2019. Unbounded Nondeterminism: An Introduction for the Philosopher of Computing. Available from: https://webtv.univ-lille.fr/video/10392/edgar-daylight-and-felice-cardone-unbounded-nondeterminism-an-introduction-for-the-philosopher-of-computing (cited 1 July 2024).

[24] Berg, J., Chihara, C., 1975. Church's Thesis Misconstrued. Philosophical Studies: An International Journal for Philosophy in the Analytic Tradition. 28(5), 357–362.

[25] Bowie, G.L., 1973. An Argument Against Church's Thesis. The Journal of Philosophy. 70(3), 66–76.

[26] Ross, D., 1974. Church's Thesis: What Its Difficulties Are and Are Not. The Journal of Philosophy. 71(15), 515–525.

[27] Kapantaïs, D., 2016. A Refutation of the Church–Turing Thesis According to Some Interpretation of What the Thesis Says. In: Müller, V.C. (Ed.). Computing and Philosophy: Selected Papers from IACAP 2014. Springer: Cham, Switzerland. pp. 45–62.

[28] Kapantaïs, D., 2018. A Counterexample to the Church–Turing Thesis as Standardly Interpreted. APA Newsletter on Philosophy and Computers. 18(1),

24–27.

[29] Davis, M., 1988. Mathematical Logic and the Origin of Modern Computers. In: Herken, R. (Ed.). The Universal Turing Machine: A Half-Century Survey. Oxford University Press: Oxford, UK. pp. 149–174.

[30] Priestley, M., 2011. A Science of Operations: Machines, Logic and the Invention of Programming. Springer: London, UK.

[31] Daylight, E.G., 2015. Towards a Historical Notion of 'Turing—The Father of Computer Science'. History and Philosophy of Logic. 36(3), 205–228.

[32] Franklin, J., 2014. An Aristotelian Realist Philosophy of Mathematics. Palgrave Macmillan: Basingstoke, UK.

[33] Linnebo, Ø., Shapiro, S., 2019. Actual and Potential Infinity. Noûs. 53(1), 160–191.

[34] Cook, R.T., 2024. The Logic of Potential Infinity. Philosophia Mathematica. nkae022. DOI: https://doi.org/10.1093/philmat/nkae022

[35] Henry, P., 1993. Mathematical Machines. In: Haken, H., Karlqvist, A., Svedin, U. (Eds.). The Machine as Metaphor and Tool. Springer: Berlin, Germany. pp. 101–122.

[36] Daylight, E.G., 2024. True Turing: A Bird's-Eye View. Minds & Machines. 34, 29–49.

[37] Copeland, B.J., 2006. Turing's Thesis. In: Olszewski, A., Wolenski, J., Janusz, R. (Eds.). Church's Thesis after 70 Years. De Gruyter: Berlin, Germany. pp. 147–174.

[38] Shagrir, O., 2006. Gödel on Turing on Computability. In: Olszewski, A., Wolenski, J., Janusz, R. (Eds.). Church's Thesis after 70 Years. De Gruyter: Berlin, Germany; Boston, MA, USA. pp. 393–419.

[39] Bringsjord, S., Arkoudas, K., 2004. The Modal Argument for Hypercomputing Minds. Theoretical Computer Science. 317(1–3), 167–190.

[40] Longo, G., 2006. The Cognitive Foundations of Mathematics: Human Gestures in Proofs and Mathematical Incompleteness of Formalisms. In: Grialou, P., Longo, G., Okada, M. (Eds.). Images and Reasoning. Keio University Press: Tokyo, Japan.

[41] Kugel, P., 2002. Computing Machines Can't Be Intelligent (…and Turing Said So). Minds & Machines. 12, 563–579.

[42] Kugel, P., 2009. You Don't Need a Hypercomputer to Evaluate an Uncomputable Function. International Journal of Unconventional Computing. 5(3–4), 209–222.

[43] Curtis-Trudel, A., 2022. Why Do We Need a Theory of Implementation? The British Journal for the Philosophy of Science. 73(4), 1067–1091.

[44] Piccinini, G., 2015. Physical Computation: A Mechanistic Account. Oxford University Press: Oxford, UK.

[45] Copeland, B.J., Shagrir, O., 2011. Do Accelerating Turing Machines Compute the Uncomputable? Minds & Machines. 21, 221–239.

[46] Rescorla, M., 2014. A Theory of Computational Implementation. Synthese. 191, 1277–1307.

[47] Kugel, P., 1986. Thinking May Be More Than Computing. Cognition. 22(2), 137–198.

[48] Daylight, E.G., Atienza, D., Vandecappelle, A., et al., 2004. Memory-Access-Aware Data Structure Transformations for Embedded Software with Dynamic Data Accesses. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 12(3), 269–280.

[49] Katsaragakis, M., Papadopoulos, L., Baloukas, C., et al., 2022. Memory Management Methodology for Application Data Structure Refinement and Placement on Heterogeneous DRAM/NVM Systems. In Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 14–23 March 2022; pp. 748–753.

[50] CloudFlare. How do lava lamps help with Internet encryption? Available from: https://www.cloudflare.com/learning/ssl/lava-lamp-encryption/ (cited 15 June 2024).

[51] Brauer, E., 2021. The Dependence of Computability on Numerical Notations. Synthese. 198(11), 10485–10511.

[52] Kleene, S.C., 1952. Introduction to Metamathematics. D. van Nostrand Company: New York, NY, USA; Toronto, ON, Canada.

[53] Shapiro, S., 2017. Computing with Numbers and Other Non-Syntactic Things: *De re* Knowledge of Abstract Objects. Philosophia Mathematica. 25(2), 268–281.

[54] Daylight, E.G., 2021. Addressing the Question "What Is a Program Text?" via Turing Scholarship. IEEE Annals of the History of Computing. 43(4), 87–91.