Church's Thesis and the Conceptual Analysis of Computability

Michael Rescorla

Abstract: Church's thesis asserts that a number-theoretic function is intuitively computable if and only if it is recursive. A related thesis asserts that Turing's work yields a *conceptual analysis* of the intuitive notion of numerical computability. I endorse Church's thesis, but I argue against the related thesis. I argue that purported conceptual analyses based upon Turing's work involve a subtle but persistent circularity. Turing machines manipulate syntactic entities. To specify which number-theoretic function a Turing machine computes, we must correlate these syntactic entities with numbers. I argue that, in providing this correlation, we must demand that the correlation itself be computable. Otherwise, the Turing machine will compute uncomputable functions. But if we presuppose the intuitive notion of a computable relation between syntactic entities and numbers, then our analysis of computability is circular.[1]

§1. Turing machines and number-theoretic functions

A Turing machine manipulates syntactic entities: strings consisting of strokes and blanks. I restrict attention to Turing machines that possess two key properties. First, the machine eventually halts when supplied with an input of finitely many adjacent strokes. Second, when the

[1] I am greatly indebted to helpful feedback from two anonymous referees from this journal, as well as from: C. Anthony Anderson, Adam Elga, Kevin Falvey, Warren Goldfarb, Richard Heck, Peter Koellner, Oystein Linnebo, Charles Parsons, Gualtiero Piccinini, and Stewart Shapiro. I received extremely helpful comments when I presented earlier versions of this paper at the UCLA Philosophy of Mathematics Workshop, especially from Joseph Almog, D. A. Martin, and Yiannis Moschovakis, and at the ASL Spring Meeting 2004, especially from Shaughan Lavine, Rohit Parikh, and Richard Zach. I am also grateful to participants in a UC Santa Barbara reading group where the paper was discussed, especially Nathan Salmon and Anthony Brueckner.

machine halts, the machine tape is inscribed with a string of adjacent strokes. Any machine that possesses these two properties computes a *string-theoretic* function: a function from strings of strokes to strings of strokes. If we denote a string of n strokes by "$\underline{n}$", then a Turing machine that doubles the number of strokes computes the function $\phi(\underline{n}) = \underline{2n}$.

Our main interest is not string-theoretic functions but number-theoretic functions. We want to investigate computable functions from the natural numbers to the natural numbers. To do so, we must correlate strings of strokes with numbers. Only then can we talk about a Turing machine computing a function defined over numbers. As Boolos and Jeffrey put it, "[b]efore we can speak of Turing machines as computing numerical functions, we must specify the notation in which the numerical arguments and values are to be represented on the machine's tape" ([1], p. 43). Strings of strokes are not numbers. They are syntactic entities. Someone who conflates numbers with strings commits a use-mention error. He confuses a symbolic item with what that item symbolizes.

Different textbooks employ different correlations between Turing machine syntax and the natural numbers. The following three correlations are among the most popular:

$d_1(\underline{n}) = n$.

$d_2(\underline{n+1}) = n$.

$d_3(\underline{n+1}) = n$, as an input

$d_3(\underline{n}) = n$, as an output.

A machine that doubles the number of strokes computes $f(n) = 2n$ under d_1, $g(n) = 2n+1$ under d_2, and $h(n) = 2n+2$ under d_3. Thus, the same Turing machine computes different numerical functions relative to different correlations between symbols and numbers.

More formally, let us define a *semantics* for a set of symbols as a bijective mapping *d* from the symbols to the natural numbers. We say that *Turing machine M computes number-theoretic function f relative to semantics d* just in case the Turing machine computes a string-theoretic function φ such that:

$$\varphi(\underline{n}) = \underline{m} \text{ iff } f(d(\underline{n})) = d(\underline{m}).$$

We say that a number-theoretic function is *Turing-computable relative to semantics d* just in case some Turing machine computes it relative to *d*.

These definitions reflect a relativity inherent to Turing-computability. The superficially two-place relation "Turing machine *M* computes number-theoretic function *f*" disguises a suppressed parameter. It results from holding fixed one element in a *three*-place relation: "Turing machine *M* computes number-theoretic function *f* relative to semantics *d*." When we hold parameter *d* constant, we obtain a two-place relation between Turing machines and number-theoretic functions. But the two-place relation instantiates a more general three-place relation.

There exist uncountably many correlations between numbers and syntactic strings. As we will see in §2, many of these correlations seem highly anomalous. In §§3-4, I investigate how the anomalous correlations bear upon Church's thesis. In §5, I argue that, by distinguishing between acceptable and unacceptable correlations, we inject a persistent circularity into our analysis of computability.

§2. Semantics and Turing-computability

In the previous section, we considered three possible correlations between numbers and strings of strokes: d_1, d_2, and d_3. The same number-theoretic functions are Turing-computable relative to each of these correlations. In this sense, d_1, d_2, and d_3 are *computationally equivalent*.

They demarcate the same privileged class of number-theoretic functions. Following standard usage, I call these privileged functions "recursive." (Note that here I follow the common practice of using the term "recursive" in connection with Turing's formalism for analyzing computability. Soare [25] criticizes this practice. As Soare argues, the term "recursive" was initially introduced in connection with the equation calculus. Only later did "recursive" come to mean something like "Turing-computable relative to d_1." The two concepts are extensionally equivalent, but they are intensionally distinct. Soare argues that we should return to the original usage, which was strongly favored by both Turing and Gödel. While I share Soare's concerns, I will follow current standard usage. In general, these intensional distinctions will not affect my discussion, although they become fleetingly relevant near the end of §3.)

If we employ a semantics sufficiently different from d_1-d_3, then various non-recursive functions become Turing-computable. Suppose X is an infinite, co-infinite subset of the natural numbers. Enumerate the elements of X and $N \setminus X$ in ascending order as follows: $X = \{x_0, x_1, x_2, \ldots\}$ and $N \setminus X = \{y_0, y_1, y_2, \ldots\}$. Define semantics d_X by:

$$d_X(\underline{n}) = x_{n/2}, \text{ if } n \text{ is even}$$

$$d_X(\underline{n}) = y_{(n-1)/2}, \text{ if } n \text{ is odd.}$$

We can then program a Turing machine that computes the characteristic function of X, relative to d_X. The Turing machine proceeds as follows: when supplied with a string of n strokes, the machine examines whether n is even or odd; if n is even then the machine outputs $d_X^{-1}(0)$; if n is odd, then the machine outputs $d_X^{-1}(1)$. This technique applies to any infinite, co-infinite set X, whether or not X possesses a recursive characteristic function.

Say that a semantics is *uniform* if it assigns the same number to each string $\underline{n}$ when $\underline{n}$ appears as an input and when $\underline{n}$ appears as an output. Semantics d_3 is not uniform, but d_1, d_2, and

d_X are uniform. The construction from the previous paragraph generalizes, yielding the following result: any number-theoretic function with finite range is Turing-computable relative to some uniform semantics. Similar techniques establish that there exist uncountably many number-theoretic functions with *infinite* range that are Turing-computable relative to some uniform semantics. There exist only countably many recursive functions. So the number-theoretic functions Turing-computable relative to some uniform semantics outstrip the recursive functions.

It is easy to show that every number-theoretic function is Turing-computable relative to some non-uniform semantics. However, there exist number-theoretic functions that are not Turing-computable relative to any uniform semantics. For an example of such a function, see Shapiro [20].[1] Shapiro's example generalizes, yielding the following result: there exist uncountably many number-theoretic functions that are not Turing-computable relative to any uniform semantics. Hence, the functions Turing-computable relative to some uniform semantics comprise a substantial but highly non-exhaustive subset of the number-theoretic functions.

To simplify matters, I henceforth restrict myself to uniform semantic relations. In what follows, "semantics" means "uniform semantics."

§3. A difficulty surrounding Church's thesis

Intuitively speaking, a function is "computable" just in case there exists a mechanical procedure for determining what value the function attains on a given input. According to Church's thesis, a number-theoretic function is intuitively computable if and only if it is recursive.[2] I will not discuss the right-to-left direction of Church's thesis, except to note that, like most commentators, I find it relatively evident. Traditionally, the left-to-right direction has been

viewed as much more problematic. I now want to explore an apparent difficulty surrounding it. In §4, I will try to resolve the difficulty.[3]

Let X consist of the Gödel numbers of sentences provable in Peano Arithmetic, under some fixed Gödel-numbering. Let f_X be the characteristic function of X. In the previous section, we constructed a semantics d_X such that f_X is Turing-computable relative to d_X. Imagine, then, a philosopher who reasons as follows:

> If Peano Arithmetic is consistent, then f_X is not recursive. But that does not show that f_X is uncomputable. On the contrary, it shows that Church's thesis enshrines an overly restrictive conception of computability. We must expand our conception, supplementing meager semantic relations like d_1 with more useful relations like d_X. Once we adopt these additional semantic relations, many additional functions become computable. There are numerous computable functions beyond the recursive functions.

This line of reasoning seems dubious. But why? Why should we believe that every intuitively computable function is Turing-computable relative to a simple semantics like d_1, as opposed to a more intricate semantics like d_X? In other words, why should we accept the strong thesis that every intuitively computable function is recursive, rather than the weak thesis that every intuitively computable function is Turing-computable relative to some semantics?

Faced with such questions, one naturally consults Turing's classic [27]. Many logicians and philosophers regard this paper as providing the most convincing defense of Church's thesis. Famously, Gödel did not accept Church's thesis until encountering Turing's article.

I follow closely the interpretation of Turing developed by Robin Gandy [6] and Wilfried Sieg [21], [22]. On this interpretation, Turing's argument contains two parts. First, Turing adduces constraints upon the mechanical activity of idealized human agents. Second, Turing argues that any function computable within these constraints is recursive.

Turing begins by imagining an idealized computing agent who performs calculations on a piece of paper. Following Gandy and Sieg, let us call Turing's idealized computing agent a "computor." The computor's paper is divided into squares, and each square is either blank or else contains a symbol. To calculate, the computor manipulates the symbols inscribed upon the paper. Turing isolates five constraints governing these symbolic manipulations, summarized as follows by Sieg ([22], p. 93):

> (a) The behavior of a computor is determined uniquely at any moment by two factors: (1) the symbols or symbolic configurations he observes, and (2) his "state of mind" or "internal state."
>
> (b) There is a fixed finite number of symbolic configurations a computor can immediately recognize.
>
> (c) There is a fixed finite number of states of mind that need to be taken into account.
>
> (d) Only elements of observed symbolic configurations can be changed.
>
> (e) The distribution of observed squares can be changed, but each of the new observed squares must be within a bounded distance *L* of an immediately previously observed square.[4]

According to Turing, constraints (a)-(e) govern the mechanical activity of any human computing agent. Turing motivates these constraints by citing various limits upon our perceptual and cognitive apparatus.[5]

Turing then presents a key result, which Sieg labels *Turing's theorem* and formulates as follows: "Any number-theoretic function *f* that can be computed by a computor, satisfying… [conditions (a)-(e)], can be computed by a Turing machine" ([22], p. 94). The basic idea is this: given a computor who satisfies (a)-(e), we construct a Turing machine that mimics the computor's computational activity. If we accept that constraints (a)-(e) demarcate the intuitively computable functions, then Turing's theorem establishes that every intuitively computable number-theoretic function is computable by some Turing machine.

Two caveats regarding this argument deserve emphasis. First, Turing's analysis concerns only *human* mechanical activity, not *general* mechanical activity. For instance, constraints (b) and (d) imply that there exists some fixed upper bound on the number of symbols that the computor can manipulate simultaneously. This consequence seems plausible for humans but not for all possible computing devices. As Gandy notes, "we can conceive of a machine which prints an arbitrary number of symbols simultaneously" ([6], p. 125). Such a machine would not impugn Turing's analysis, because constraints (a)-(e) do not purport to encompass all possible computations. They describe only human computations. For Turing, "computable" means "computable by a human," not "computable by some possible machine."[6]

Second, Turing's analysis concerns human *mechanical* activity, not human cognition in general. As Gödel puts it, "the question of whether there exist finite *non-mechanical* procedures, not equivalent with any algorithm, has nothing whatsoever to do with the adequacy of the definition of… 'mechanical procedure'" ([10], p. 370). Constraints (a)-(e) do not purport to govern all possible processes for determining some function's value. Imagine a computor who possesses a mysterious cognitive faculty, which enables him to determine some uncomputable function's value upon any input. When presented with an input, the computor "just knows" the correct answer. Constraints (a)-(e) do not deny the existence of this mysterious cognitive faculty. They merely deny that someone who deploys the faculty thereby computes a number-theoretic function. By employing the faculty, the computor introduces an essentially *non-mechanical* element into his mathematical activity. He implements a non-algorithmic cognitive strategy. We cannot deny *a priori* that such cognitive strategies exist. But we can expunge them from our account of computation. That is precisely what (a)-(e) seek to accomplish.

Given these caveats, how convincing is Turing's argument? I find certain aspects of it puzzling. For instance, constraints (a) and (c) include the rather ill-defined phrase "state of mind." What are these states of mind, and why should we assume that only finitely many of them are relevant to a given computation?[7] I also harbor some worries about "Turing's theorem," which asserts that any function computable within constraints (a)-(e) is computable by a Turing machine. Turing's defense of this "theorem" is so elliptical that I find it difficult to understand.[8]

I set such worries aside. Even if they prove surmountable, a more serious worry remains. As I will now argue, Turing's argument exhibits a crucial lacuna. At best, Turing establishes that every intuitively computable function is Turing-computable *relative to some semantics*, not that every intuitively computable function is Turing-computable *relative to semantics* d_1.

Turing adduces constraints upon mechanical manipulation of syntactic items. Constraint (a) demands that computational processes be somehow deterministic. Constraints (b) and (c) mandate that the perceptual and cognitive capacities deployed during computation be finitely limited. Constraints (d) and (e) impose spatial bounds upon the computor's ability to observe and adjust symbols. All five constraints concern how the computor manipulates symbolic representations for numbers. They do not address how symbolic representations and numbers relate to one another. They therefore provide no basis for commending certain semantic relations over others. In particular, they do not favor d_1 over d_X.[9]

Consider the following scenario: a computor manipulates symbols that possess semantics d_X, conforming his manipulations to (a)-(e). Should we classify this scenario as computation? Should we say that the computor computes non-recursive functions, such as f_X? The scenario satisfies constraints (a)-(e), which are the only constraints Turing advances. Thus, based on Turing's analysis, the proposed scenario counts as computational. Yet to classify the scenario as

computational would be to classify certain non-recursive functions as computable, contradicting Church's thesis.

Of course, if we restrict the computor to a semantics like d_1, then the computor can compute only recursive functions. Unfortunately, Turing's discussion provides no reason for imposing this restriction. The proposed scenario is anomalous because the computor manipulates items that possess deviant meanings, not because he manipulates syntactic items in a deviant way. The computor employs a deviant representational relation between symbols and numbers. Turing cannot criticize the representational relation as deviant, for he recognizes no constraints governing how symbols represent numbers. He focuses exclusively upon syntactic manipulation, at the expense of meaning. So he lacks the resources needed for dismissing d_X as unacceptable. He provides no basis for classifying the proposed scenario as non-computational.

Turing therefore fails to establish Church's thesis. At best, Turing shows that Turing machines can replicate all syntactic operations implemented mechanically by humans. He shows that all intuitively computable string-theoretic functions are Turing-computable. But that implies only that each intuitively computable number-theoretic function is Turing-computable *relative to some semantics*, not that each intuitively computable number-theoretic function is Turing-computable *relative to* d_1. Turing does not exclude the possibility of a computor who computes non-recursive functions.

To buttress my argument, I now consider the following objection.[10]

I just conceded that every intuitively computable string-theoretic function is Turing-computable. Thus, given any human string-theoretic computation that proceeds relative to some semantics, there is a Turing-machine that computes the same number-theoretic function relative to that same semantics. It follows that every intuitively computable number-theoretic function is

Turing-computable. At best, then, my argument shows that Turing fails to establish the following: every Turing-computable number-theoretic function is recursive. But it is a *mathematical theorem* that every Turing-computable function is general recursive, where "general recursive" is understood in terms of Kleene's equation calculus. Once supplemented by this mathematical theorem, Turing's argument establishes the desired conclusion.

The flaw in this objection lies in a crucial ambiguity surrounding the phrase "Turing-computable." If that phrase means something like *Turing-computable relative to* d_1 (or to some other fixed semantics), then the third sentence of the preceding paragraph does not follow from the second sentence. We cannot conclude from the fact that a number-theoretic function is intuitively computable that it is computed relative to d_1 either by some intuitively computable string-theoretic function or by some Turing-computable string-theoretic function. To draw this inference would beg the question, since we are investigating whether computability relative d_1 exhausts the intuitive notion of numerical computability. So far, we have seen no reason to think that every intuitively computable number-theoretic functions is computable relative to d_1.

On the other hand, if we understand the phrase "Turing-computable" to mean "Turing-computable relative to *some* semantics," then the third sentence follows from the second sentence, given the plausible assumption that any intuitively computable numerical function is computed relative to *some* semantics by some intuitively computable string-theoretic function. But, under this alternative interpretation of "Turing-computable," we can no longer say that every Turing-computable number-theoretic function is general recursive. On the contrary, function f_X from the previous section is a counter-example. In the standard mathematical theorem that every Turing-computable function is general recursive, "Turing-computable" means something like *Turing-computable relative to* d_1 (or to some other fixed semantics). Under this

interpretation, the theorem is unimpeachable. But the theorem becomes false if we interpret "Turing-computable" to mean *Turing-computable relative to some semantics*. Of course, we might *also* alter the definition of "general recursive" to mean something like *computable in the equation calculus relative to some semantics*, where we allow deviant semantic relations like d_X. Then it would once again become correct to say that every Turing-computable numerical function is general recursive. But this hardly establishes that every intuitively computable numerical function is general recursive *in the normal sense of "general recursive."*

My objections apply to many discussions besides Turing's. Consider Boolos and Jeffrey [1]. Boolos and Jeffrey initially characterize Church's thesis as the statement that "any mechanical routine for symbolic manipulation can be carried out in effect by some Turing machine or other" (p. 52). This characterization is not quite accurate, since it admits non-recursive functions that are Turing-computable relative to deviant notations like d_X. To ensure a more accurate characterization, Boolos and Jeffrey supplement their initial statement with the stipulation that Turing machine syntax possesses semantics d_1. They call this semantics "monadic notation." Boolos and Jeffrey acknowledge the possibility of semantic relations besides monadic notation. They provide no argument for privileging monadic notation over its rivals. Instead, they write:

> No end of notations might be invented, and there is no hope of proving that everything computable in any of them is computable in monadic notation. It is for this reason that we adopt the monadic notation at the outset: *define* Turing-computability as computability… [relative to monadic notation]; and interpret [Church's thesis] in light of that definition.[11]

Clearly, we may offer whatever definitions we like. But if we want to *defend* Church's thesis, mere stipulation does not suffice. If we want to show that every intuitively computable function

is Turing-computable relative to monadic notation, we cannot simply assume that all computations proceed relative to monadic notation.

Establishing Church's thesis requires us to move beyond constraints governing syntactic manipulation. We must address the semantic relation between symbols and numbers. In addressing this relation, we cannot merely stigmatize correlations like d_X as "deviant," "artificial," or "unreasonable." Nor can we confine ourselves to praising correlations like d_1 as "privileged," "natural," or "canonical." The question is not whether certain semantic relations are privileged over others. The question is *what* privileges certain semantic relations over others. This question deserves a principled answer, one which yields an invidious distinction between the anodyne d_1 and the deviant d_X.

In the next section, I address the semantic relation between symbols and numbers, and I present an emended argument for Church's thesis.[12]

§4. A proposed solution to the difficulty

Just as we possess the intuitive notion *computable number-theoretic function*, we possess the intuitive notion *computable semantics*. A semantics for some set of symbols is computable just in case there exists a mechanical procedure for computing what number a given symbol denotes. For instance, the correlations d_1-d_3 are clearly computable. The concept *computable semantics* will strike some philosophers as mysterious or obscure. But why should the idea of a computable function from symbols to numbers seem any more obscure than that of a computable function from numbers to numbers? If we accept the latter as legitimate, we should likewise accept the former.

The notion *computable semantics* is the key ingredient missing from Turing's account. By embracing it, we can construct an improved argument for Church's thesis. In presenting this argument, I presuppose "Turing's theorem": I assume that Turing machines can replicate all mechanical symbolic manipulation implemented by humans. In other words, I assume that all intuitively computable *string-theoretic* functions are Turing-computable. Due to this assumption, my argument faces many obstacles that face Turing's. For instance, any worries regarding Turing's locution "state of mind" also infect my argument. I continue to set such worries aside.

My argument for Church's thesis invokes two intuitive principles:

The composition of two computable functions is computable.

The inverse of a bijective computable function is computable.

The first principle, which says that the computable functions are closed under composition, applies to all computable functions. (*Informal proof*: given an algorithm for computing f and an algorithm for computing g, compute $f(g(x))$ by applying the second algorithm to x and then applying the first algorithm to the result.) The second principle applies to those computable functions defined over domains whose elements one can mechanically enumerate. (*Informal proof*: given a computable function f, and given y, here is an algorithm for computing $f^{-1}(y)$: enumerate the elements of the domain, computing the value that f attains on each element in the enumeration, until encountering an element x such that $f(x) = y$; take x to be $f^{-1}(y)$.)

One might worry about the assumption, which underlies our second intuitive principle, that one can mechanically enumerate the elements of the relevant domains. Don't we need Church's thesis in order to say which domains conform to the assumption? And, if so, won't an argument for Church's thesis based upon that assumption beg the question?[13]

I respond that, although we may require Church's thesis to offer a general, precise characterization of the domains whose elements we can mechanically enumerate, we do not require Church's thesis to recognize certain specific examples of mechanically enumerable domains. Similarly, although we may require Church's thesis to characterize which number-theoretic functions are intuitively computable, we do not require Church's thesis to recognize that certain particular functions are intuitively computable. We intuitively recognize that the natural numbers are mechanically enumerable, via the successor function, and that the stroke language is mechanically enumerable, via adjunction by an additional stroke. This recognition does not depend upon Church's thesis. In general, all the domains relevant to this paper are intuitively recognizable as mechanically enumerable, without reliance upon Church's thesis. (I will henceforth ignore the assumption of mechanical enumerability, somewhat sloppily describing the computable functions as being closed under inverses.)

Given our two intuitive principles, we can easily establish Church's thesis. Suppose f is an intuitively computable numerical function. Define $\varphi = d_1^{-1}\, f\, d_1$. Note that

$$\varphi(\underline{n}) = \underline{m} \text{ iff } f(d_1(\underline{n})) = d_1(\underline{m}),$$

and hence that φ is a string-theoretic function that computes f relative to d_1. Since d_1 is intuitively computable, and since the computable functions are closed under inverses and composition, φ is computable. By Turing's theorem, φ is Turing-computable. Thus, f is Turing-computable relative to d_1. In other words, f is recursive.[14]

Although the proof is mathematically trivial, it introduces a crucial conceptual ingredient missing from Turing's original account: the notion of a computable semantics. Our proof deploys this notion to establish that d_1 yields a notation for the natural numbers sufficiently general for computing *any* intuitively computable numerical function. Crucially, our proof invokes no

features of d_1 beyond its intuitive computability. So it would work equally well for any intuitively computable semantics. Indeed, we may instructively view the proof as exploiting the following generalization:

(*) If a number-theoretic function *f* is intuitively computable, then, for any intuitively computable semantics *d*, there exists an intuitively computable string-theoretic function that computes *f* relative to *d*.

This generalization follows from the fact that the computable functions are closed under inverses and composition.

The notion of a computable semantics does not merely help us prove Church's thesis. It also helps pinpoint what Turing's constraints (a)-(e) omit.

Consider again the scenario described by Turing: an idealized human computor manipulates symbols inscribed on paper. The computor manipulates these symbols because he wants to calculate the value some number-theoretic function assumes on some input. The computor starts with a symbolic representation for the input, performs a series of syntactic operations, and arrives at a symbolic representation for the output. This procedure succeeds only when the computor can *understand* the symbolic representations he manipulates. The computor need not know in advance which number a given symbol represents, but he must be capable, in principle, of determining which number the symbol represents. Only then does his syntactic activity constitute a computation of the relevant number-theoretic function. If the computor lacks any potential understanding of the relevant syntactic items, then his activity counts as mere manipulation of syntax, rather than calculation of one number from another.

These reflections suggest an important new constraint upon Turing's computor, to supplement Turing's original constraints (a)-(e):

***The Computability Constraint*:** The symbols that the computor manipulates bear a computable semantic relation to the numbers they denote.

If the computor manipulates syntactic items that possess a non-computable semantics, then he cannot mechanically determine which number a given symbol denotes. He cannot understand the symbols through purely algorithmic means. But then he cannot calculate which numerical value the desired function assumes on a given input.

I must emphasize that, like Turing's constraints (a)-(e), the Computability Constraint concerns human *mechanical* activity. Suppose that our computor possesses a mysterious cognitive faculty that enables him to understand uncomputable notations. When confronted with symbolic representations for the natural numbers, the computor "just knows" which number a given symbolic representation denotes. The Computability Constraint does not deny that such a faculty exists. It merely denies that someone who exploits the faculty thereby *computes* a number-theoretic function. Someone who employs the mysterious faculty introduces an essentially non-mechanical element into his mathematical activity. He understands the symbols he manipulates, but he understands them in an irreducibly non-mechanical way. He implements a non-algorithmic cognitive strategy. We cannot deny *a priori* that such cognitive strategies exist. But we can expunge them from our account of computation.

The Computability Constraint provides a straightforward diagnosis for why semantics d_X seems deviant. Suppose that X is a non-recursive set. If d_X were computable, it would follow that f_X, the characteristic function of X, was computable. By Church's thesis, it would follow that f_X was recursive. But f_X is not recursive. Hence, d_X is not computable. There exists no mechanical procedure for determining which number a given symbol denotes under d_X. This renders d_X useless for computation.

Thus, the notion of a computable semantics considerably illuminates the difficulties raised in §4. It helps us elaborate Turing's original treatment into a rigorous argument for Church's thesis. And it allows us, through the Computability Constraint, to pinpoint why semantics d_X seems so deviant. I urge that we supplement Turing's account with the notion of a computable semantics.

§5. The conceptual analysis of computability

In the previous section, I presented an emended argument for Church's thesis. I now want to discuss a related thesis, according to which Turing provides a conceptual analysis of the intuitive notion *computable number-theoretic function*. From this perspective, Turing's account transcends mere extensional equivalence with our intuitive notion of computability: it somehow *explicates* or *captures* the intuitive notion. Thus, Gandy declares that Turing "provid[es] the definitive meaning of 'computable function'" and that "Turing's work is a paradigm of philosophical analysis: it shows that what appears to be a vague intuitive notion has in fact a unique meaning which can be stated with complete precision" ([7], p. 84, 86). Similarly, Gödel writes that "Turing's work gives an analysis of the concept of 'mechanical procedure' (alias 'algorithm' or 'computation procedure' or 'finite combinatory procedure'). This concept is shown to be equivalent with that of a 'Turing machine'" ([10], p. 369-70). Gödel contrasts Turing favorably with logicians like Church, Kleene, and Gödel himself. According to Gödel, these other logicians offered extensionally adequate characterizations of computability, but they did not *analyze* it. The omission left us with little reason to believe that their definitions were extensionally adequate. Turing offered a genuine analysis, thereby establishing Church's thesis.

Sieg, who develops the position more fully than Gödel or Gandy, writes [21, p. 391]:

> Church's or Turing's thesis [asserts] that an informal notion of effective calculability is captured fully by a particular precise mathematical concept. Church's thesis, for example, claims in its original form that the effectively calculable number-theoretic functions are exactly those functions whose values are computable in Gödel's equation calculus. My strategy, when arguing for the adequacy of a notion, is to bypass theses altogether and avoid the fruitless discussion of their (un)-provability. This can be achieved by *conceptual analysis*… There is general agreement that Turing, in 1936, gave the most convincing analysis of *effective calculability*… It can be argued that he gave the only convincing analysis… The detailed conceptual analysis of effective calculability yields rigorous characterizations that dispense with theses, reveal human and machine calculability as axiomatically given mathematical concepts, and allow their systematic reduction to Turing computability.

Sieg centers his account around Turing's constraints (a)-(e). According to Sieg, (a)-(e) analyze the intuitive concept of computability. More precisely, we can analyze *mechanical procedure carried out by a computor* as *computation of a computor satisfying constraints (a)-(e).* This analysis, coupled with Turing's theorem, underwrites our confidence in Church's thesis.[15]

Obviously, in evaluating whether Turing analyzes computability, much depends upon what we mean by "conceptual analysis." Gandy, Gödel, and Sieg are not very explicit on this point. However, most philosophers would probably acknowledge the following three desiderata for a good analysis:

> The analysis is extensionally adequate.
>
> The analysis is non-circular, i.e. it does not employ the concept being analyzed.
>
> The analysis "captures the meaning" of the original concept.

The third desideratum is the most problematic, since what it is to "capture the meaning" of a concept remains quite unclear. Nevertheless, something along these lines seems integral to

analyzing a concept, as opposed to offering necessary and sufficient conditions. I will argue that Turing's work provides no analysis of *computable number-theoretic function* satisfying all three desiderata.[16]

I begin with a concession: Turing may well successfully analyze *computable string-theoretic function*. I find it plausible that constraints (a)-(e), or constraints much like them, explicate the concept *mechanical procedure for manipulating syntactic items*. Thus, I concede that Turing analyzes what it is to compute a string-theoretic function.[17]

However, I deny that Turing analyzes what it is to compute a *numerical* function. Number-theoretic computability essentially involves extra-syntactic entities: the numbers. As I argued in §3, constraints (a)-(e) ignore the semantic relation between numbers and the syntactic items that represent them. No account based solely upon constraints (a)-(e) disbars deviant semantic relations. So no such account decrees that the recursive functions exhaust the computable functions. So no such account satisfies the first desideratum for any conceptual analysis: extensional adequacy. The concept *number-theoretic function computable in accord with constraints (a)-(e)* is not extensionally equivalent to *computable number-theoretic function*.

Given Church's thesis, we can remedy this defect. For instance, we might adopt either of the following characterizations:

computable in accord with constraints (a)-(e) relative to d_1

computable in accord with constraints (a)-(e) relative to d_2,

or, more generally, any characterization of the form:

computable in accord with constraint (a)-(e) relative to d,

where *d* is some specific intuitively computable semantics. By conditional (*) from §4, each of these infinitely many characterizations is extensionally adequate.

But *which* should we choose as our conceptual analysis? I see little basis for choosing one characterization over the others. Any computable semantics might subserve some mathematician's computational activity. Speaking purely historically, humans have employed a wide variety of numerical notations: Roman numerals, Arabic decimals, etc. The Babylonians even used a base 60 notation. Why regard one of these notations, rather than another, as constitutively tied to the concept *number-theoretic computability*?

My point here is *not* simply this: we possess infinitely many extensionally adequate characterizations, each of which seems as good a candidate for conceptual analysis as any other; thus, *none* can provide a conceptual analysis. That is *one* important point, but it is not, I think, the most fundamental point.

The most fundamental point is that none of these infinitely many putative analyses attains a sufficiently general description of numerical computation. Our essential concept of number-theoretic computability amounts to this: a number-theoretic function is computable just in case there exists a mechanical procedure for computing it. Thus, if we want to analyze the concept *computable number-theoretic function*, we must analyze the concept *mechanical procedure for computing a number-theoretic function*. We must isolate the salient features shared by all possible number-theoretic computations. Only then do we achieve synonymy with the target concept *number-theoretic computability*. None of our putative analyses attains anything like the requisite generality. Each proposed analysis captures only a limited class of computations, namely, those computations that occur relative to some fixed semantics.

Note that an utterly harmless analogue to this phenomenon arises when we define the mathematical locution "recursive number-theoretic function." Here, too, we face a choice of infinitely many computable semantic relations to mention in our characterization. Following

standard mathematic practice, I chose d_1-d_3 when I defined "recursive" in §2. In the context of defining new mathematical locutions, such a choice is perfectly legitimate. One can define one's terminology however one likes. But an arbitrary choice along these lines becomes fatal if we seek to analyze a *pre-existing* concept like numerical computability. In that case, we must attain a sufficiently general description of what all numerical computations have in common. We undercut this goal if we center our putative analysis around a particular notation, since not all computations proceed relative to that notation.[18]

Compare string-theoretic computability. Constraints (a)-(e), or constraints much like them, quite plausibly govern any possible human mechanical procedure for manipulating syntactic items. They therefore quite plausibly provide the basis for analyzing the concept *computable string-theoretic function*. To analyze the concept *computable number-theoretic function*, we must attain an analogous level of generality regarding the semantic relation between symbols and numbers. Just as Turing adduces general constraints upon the manipulation of syntactic strings, we must adduce general constraints upon any acceptable notation for the natural numbers.

My criticisms echo an enigmatic passage from Emil Post's posthumously published [17]: "Finite operations illuminated as generated by three principles (1) Symbolic 'manipulation' (2) Symbolization (3) Iteration" (p. 426).[19] Turing focuses exclusively upon factors (1) and (3): *symbolic manipulation* and *iteration*. He offers a general theory of iterated human symbol manipulation. He provides no correspondingly general theory of Post's second factor: *symbolization*. In Post's words, Turing does not supplement his treatment of iterated symbol manipulation with "an equally persuasive analysis… [of] all humanly possible modes of

symbolization" (p. 344). Only once we provide such an analysis can we claim to have captured the concept *number-theoretic computability*.

To furnish the requisite analysis, we might invoke the Computability Constraint. In other words, we might offer the following characterization of number-theoretic computability: *computable in accord with (a)-(e) relative to some computable semantics*. This characterization is extensionally adequate. Moreover, unlike our earlier efforts, it reflects general constraints upon any possible number-theoretic computation. The problem is that it also seems blatantly circular, because it presupposes the intuitive notion of computability. Admittedly, it presupposes the concept *computable function from symbols to numbers*, not the slightly different concept *computable function from numbers to number*. Such a minor discrepancy hardly dispels the circularity. By adopting the proposed account, we replace *mechanical procedure for computing one number from another* with *mechanical procedure for computing a number from a symbol*. We leave unanalyzed what it is to calculate a number from an input. We thereby abandon all pretensions towards reductive analysis.

We face a dilemma. If we characterize number-theoretic computability by invoking some fixed computable semantics, our account does not analyze *mechanical procedure for computing a number-theoretic function* and hence does not achieve synonymy with the target concept. Yet when we fix this problem by invoking the notion *computable semantics*, we inject a blatant circularity into our account. How can we isolate an extensionally adequate characterization that is both non-circular *and* synonymous with the original concept? Lacking a satisfactory answer to this question, we must conclude that syntactic accounts like Turing's fail to analyze numerical computation in more primitive terms. Undoubtedly, Turing's discussion profoundly illuminates computability. But illuminating a concept is not the same as analyzing it.

My argument draws an invidious distinction between string-theoretic and number-theoretic computability. But it might seem that the distinction is spurious. Don't my worries about number-theoretic computation arise just as readily at the level of string-theoretic computation? To characterize a given physical activity as computing a string-theoretic function, we must "encode" the strings as physical states. And won't the problem of deviant encodings arise here as well, forcing us to invoke intuitively computable correlations between physical states and strings?

No. The reason is that strings are fundamentally different entities than numbers.

In the terminology introduced by Charles Parsons [15], syntactic entities are *quasi-concrete*. They are abstract, in that they are not located in space or time, but they bear intrinsic relations to privileged concrete embodiments. For example, a string of *n* strokes, viewed as a type, is an abstract entity, but its tokens are concrete physical inscriptions. The relation between the string and its tokens is constitutive of the string's identity. In contrast, numbers are *pure abstract* entities. They do not bear intrinsic relations to concrete embodiments. In particular, they do not bear intrinsic relations to either syntactic-tokens or syntactic-types. The basic insight here goes back to Frege [5], who observed that "[o]ne could imagine the introduction some day of quite new numerals, just as, e.g., the Arabic numerals superseded the Roman. Nobody is seriously going to suppose that in this way we should get quite new numbers, quite new arithmetical objects, with properties still to be investigated." Frege concluded from this observation that "we must distinguish between numerals and their *Bedeutungen*" (p. 132). Frege's observation actually supports a somewhat more general conclusion: the individuation of the numbers is not tied to particular symbolic representations. If it were, then a change in symbolic representations would entail a change in the numbers, which it does not.[20]

This asymmetry between strings and numbers explains why my argument applies to numerical computability but not to string-theoretic computability. A canonical correlation between string-types and concrete inscriptions is built into the identity of the string-types. For instance, the type "string of *n* strokes" is canonically associated with concrete inscriptions featuring *n* adjacent strokes, because it is partially individuated by the fact that such inscriptions are its tokens. We need not worry about disbarring deviant encodings of strings as physical states, for the individuation of the strings enshrines a single privileged encoding. No canonical correlation between numbers and syntactic-types or syntactic-tokens is built into the individuation of numbers. This naturally raises the question of which correlations are admissible for computation. We have seen no way to answer this question without engendering circularity.

Perhaps a non-circular analysis of numerical computability exists. But I want to advertise my inability to locate one. In this vein, I will survey various obvious but unsuccessful maneuvers through which one might attempt to eliminate the circularity.

The most obvious maneuver would be to adopt the following analysis:

computable in accord with constraints (a)-(e) relative to every *semantics*.

However, as Shapiro [20] proves, the only functions that satisfy this condition are those differing from constant functions or the identity function on at most a finite number of arguments. Thus, the proposed maneuver is clearly inadequate.

Another obvious maneuver would be to "go meta-linguistic." We can introduce symbols that denote other symbols; the expression "$\underline{n}$", denoting a string of *n* strokes, is an example. We can then formulate string-theoretic mechanical procedures that compute symbols from meta-linguistic symbols, thereby calculating what number a given symbol denotes. Thus, we might

analyze *computable semantics* by adducing (a)-(e) as constraints upon transforming a meta-linguistic symbol into non-meta-linguistic symbol.

Unfortunately, this meta-linguistic maneuver achieves no advance whatsoever. A meta-linguistic computational procedure computes a numerical output from a syntactic input only relative to a semantics that correlates meta-linguistic symbols with symbols and non-meta-linguistic symbols with numbers. Lacking such a correlation, the computational procedure counts as mere manipulation of syntax, not as computation of a number-valued function. But now our worries about deviant correlations rematerialize, this time in connection with both meta-linguistic and non-meta-linguistic symbols.

A third obvious maneuver would be to invoke the isomorphism between the Dedekind structure of the notation system and the Dedekind structure of the natural numbers. To take a specific example, consider again the stroke language. We can view this language as an ω-sequence, with adjunction by an additional stroke serving as the successor operation. Then there exists a unique isomorphism between the stroke language and the natural numbers. Surely we can select that isomorphism as providing the "canonical" interpretation of the stroke language.

This maneuver fails. The problem is that the stroke language instantiates infinitely many different ω-sequences. Given semantics d for the stroke language, define the string-theoretic function $S(t) = d^{-1}(d(t)+1)$. We can again view the stroke language as an ω-sequence, with S rather than adjunction serving as the successor operation. This new ω-sequence is once again isomorphic to the natural numbers. Thus, the mere appeal to Dedekind structure achieves nothing. It does not favor one semantics over another.

A fourth maneuver would be to treat d_1-d_3 as privileged over all other semantic relations, on the grounds that they assign the "correct meaning" to the syntactic operation of adjunction.

Unlike most other semantic relations, d_1-d_3 interpret adjunction as corresponding to the successor operation on the natural numbers. This interpretation may appear somehow "canonical."

A basic difficulty with the proposed maneuver is that it does not generalize beyond the simple stroke language and thus cannot provide a general criterion of "acceptable notation." A subtler but equally serious difficulty is that the proposal does not seem correct even for Turing machine syntax. Contrary to the proposal, I do not think that adjunction possesses a "canonical" meaning. Inherently speaking, adjunction possesses no semantic interpretation whatsoever. It is a meaningless syntactic operation. For instance, binary notation seems no less valid than d_1-d_3 as an interpretation of Turing machine syntax. Since adjunction does not possess a canonical meaning, the difficulties that beset d_X cannot involve any failure to preserve adjunction's canonical meaning.

Although the maneuvers just canvassed do not seem very promising, other responses require more extended treatment. I now discuss four such responses.

§5.1 The Translation Constraint

Given that the computable functions are closed under composition and inverses, semantics *d* for some numerical language is intuitively computable just in case:

> ***The Translation Constraint:*** The translation between *d* and d_1 is intuitively computable.
>
> More precisely, the string-theoretic function $d_1^{-1}\ d$ is intuitively computable.

So we might offer the following characterization of number-theoretic computability:

> *computable in accord with constraints (a)-(e) relative to some semantics d that satisfies the Translation Constraint*.

This proposal replaces the Computability Constraint with an *extensionally* equivalent but *intensionally* distinct demarcation of the acceptable semantic relations. Since the Translation Constraint mentions only *string-theoretic* computability, we can explicate it through Turing's constraints (a)-(e), thereby eliminating any hint of circularity.

While the new proposal improves upon our earlier efforts, I believe that it falls short of conceptual analysis. Briefly: although the Translation Constraint provides an extensionally adequate demarcation of the acceptable notations, we determine its extensionally adequacy only by determining that it is extensionally equivalent to the Computability Constraint. What we really want is that our notations be intuitively computable. Effective intertranslatability with monadic notation is a superficial correlate of this more fundamental desideratum. Hence, it is the Computability Constraint, rather than the Translation Constraint, that constitutively attaches to our original concept of number-theoretic computability. Notations are not acceptable because they are intertranslatable with d_1; they are intertranslatable with d_1 because they are acceptable.

Clearly, the sheer fact that certain notations are mechanically intertranslatable with some other notation supplies no reason to deem them suitable for numerical computation. For instance, it would be absurd to define an "acceptable notation" as a notation mechanically intertranslatable with d_X. The Translation Constraint strikes us as adequate only because we recognize that d_1 possesses some inherently desirable property, a property which is not shared by d_X but which is preserved under computable translation. That desirable property, I submit, is intuitive computability.

To replace the Computability Constraint with the Translation Constraint is to mistakenly prioritize *translation* over *interpretation*. The fundamental desideratum upon any notation for numerical computation is that we can mechanically determine what number a given symbol

denotes. Mechanically determining what number a symbol denotes is not the same as mechanically translating that symbol into monadic notation. Indeed, upon encountering a long string of strokes, we would surely determine its reference under d_1 by translating it into a more readily intelligible symbolism, like Arabic decimal notation. Thus, the mere fact that one can mechanically translate some symbol into monadic notation is quite tangential. It seems relevant only because one can *also* mechanically interpret monadic notation, thereby mechanically interpreting the original symbol. This crucial transition from symbols to numbers goes unmentioned by the Translation Constraint. By shifting attention from the semantic relation between symbols and numbers to the syntactic relation between symbols and other symbols, the Translation Constraint obscures the essential features of numerical computation.

But what is it to interpret a numeral? More generally, how do we achieve reference to the natural numbers? Many philosophers claim that such reference is mediated by a canonical notation, such as d_1, Arabic notation, or some other favored candidate. An extreme view along these lines holds that the natural numbers just *are* elements in a canonical notation. More sophisticated views, explored by philosophers like Saul Kripke, Per Martin-Löf, and Charles Parsons, avoid the nominalism while retaining the emphasis upon symbolic mediation.[21] A particularly attractive version of this view would regard the relevant symbols as numerals in the language of thought.[22] Don't such views suggest that the Translation Constraint, or something like it, constitutively attaches to our concept of number-theoretic computability? For don't they suggest that interpreting any notation requires, perhaps in a subtle or circuitous way, translation into some canonical notation?

Let us grant that some fixed canonical notation, perhaps in the language of thought, mediates our thinking about the natural numbers. Still, it is hard to deny that *other thinkers* might

refer to the natural numbers through a *different* canonical notation. We must therefore ask what all possible canonical notations have in common. For instance, why couldn't d_X serve as the basis for some mathematician's canonical notation? In addressing such questions, the notion of an intuitively computable semantics will once more prove indispensable. We need it to explain why certain notations, but not others, could serve as the canonical basis for our thought about the natural numbers.[23]

Characterizations based upon the Translation Constraint replace an *explanatorily fundamental* concept (*computable function from symbols and numbers*) with an *explanatorily derivative* concept (*computable intertranslatability with some privileged notation*). The former concept is explanatorily more fundamental because it helps explain why certain notations rather than others are suitable for numerical computation. I conclude that characterizations based upon the Translation Constraint fail to attain synonymy with the concept *number-theoretic computability*. Such characterizations emphasize a superficial symptom shared by all acceptable notations, not the more fundamental trait that explains why they are acceptable.

§5.2 Shapiro on the computability of the successor function

It is not difficult to show that semantics *d* for some numerical language is computable just in case it satisfies the following condition:

> ***The Successor Constraint:*** The successor function is intuitively computable relative to *d*. More precisely, there exists an intuitively computable string-theoretic function ϕ such that $d(\phi(s)) = d(s) + 1$.

A natural proposal is that we replace the Computability Constraint with the Successor Constraint. We can then explicate the Successor Constraint through Turing's constraints (a)-(e), thereby evading circularity.

This proposal receives powerful support from the crucial role the natural numbers play in *counting*. Typically, we measure cardinalities by enumerating elements of some numerical notation in ascending order. This procedure only works if the successor operation is computable relative to the notation. Thus, the Successor Constraint, unlike the Translation Constraint, reflects an inherently desirable property of notations.

It would be churlish to deny that the Successor Constraint carries us much closer than our earlier efforts towards something resembling a satisfying conceptual analysis. Note, however, that the Successor Constraint is *not* extensionally equivalent to the Computability Constraint if we momentarily allow non-injective semantic relations. Given a non-recursive infinite set $Y = \{y_0, y_1, y_2, \ldots\}$ with $y_0 = 0$, consider the following repetitious enumeration of the natural numbers: $y_0, y_1, y_0+1, y_2, y_1+1, y_0+2, y_3, y_2+1, y_1+2, y_0+3$, etc. Let c be the semantics that maps $\underline{n}$ to the nth element of this enumeration. Then c is not computable, since otherwise the characteristic function of Y would be computable. Yet the successor operation is intuitively computable relative to c.

In case there was any doubt, this example demonstrates that the Successor and Computability Constraints are intensionally distinct, since they diverge extensionally over non-injective notations. The example also suggests that the Computability Constraint, rather than the Successor Constraint, supplies the correct criterion for an "acceptable notation." Semantics c conforms to the Successor Constraint, yet it is useless for computation, since there is no uniform mechanical procedure for interpreting numerals relative to c. Thus, a notation's suitability for

performing computations stems from conformity not to the Successor Constraint but to the Computability Constraint, which entails the Successor Constraint but which is entailed by it only in the special case of injective semantic correlations. Even for that special case, the Computability Constraint, not the Successor Constraint, is explanatorily fundamental.[24]

In this connection, Shapiro [20] offers a revealing discussion. Shapiro, who works only with injective notations, initially contemplates something much like Computability Constraint: "The [computor] should be able to *read* the notation. If he is given a token for a numeral, he should (in principle) be able to determine what number it denotes" (p. 18). Shapiro observes that this informal constraint entails the Successor Constraint. He adopts the Successor Constraint as his official definition of an "acceptable notation." Essentially, then, Shapiro motivates his definition by noting that it follows from something resembling the Computability Constraint. I contend that Shapiro's initial, informal characterization is preferable to his final, official definition. The former, rather than the latter, captures what renders a given notation suitable for numerical computation.

Why does Shapiro favor the Successor Constraint over the Computability Constraint? He criticizes talk about "determining what number a numeral denotes" as "vague and perhaps obscure," observing that it "seems to involve the possibility of *de re* knowledge of particular natural numbers independent of notation" (p. 18). This complaint suggests that Shapiro finds congenial the view, discussed in §5.1, that some canonical notation mediates our thought about the natural numbers. As we saw, however, that view is quite consistent with the Computability Constraint. Moreover, even if we accept such a view, it is difficult to see why the Computability Constraint should seem any more obscure than the intuitive concept *computable number-theoretic function*.

Notably, Shapiro seems to regard even this concept rather suspiciously. He writes that "strictly speaking, computability applies only to string-theoretic functions and not to number-theoretic functions" (p. 14). Ultimately, he does introduce a notion of numerical computability: a number-theoretic function *f* is "computable" just in case some computable string-theoretic function computes *f* relative to some acceptable notation, in Shapiro's sense of "acceptable." He then proves a result that he calls "Church's thesis": a number-theoretic function *f* is "computable" just in case it is recursive (p. 20). The proof is easy, once we assume that all semantic correlations are injective, for in that case the Successor Constraint entails the Translation Constraint.

But does Shapiro really prove Church's thesis? It seems to me that he does not even *formulate* it, let alone *prove* it. Church's thesis concerns the *pre-theoretic* concept *computable number-theoretic function*. Shapiro does not employ this pre-theoretic concept. Instead, he employs a syntactic proxy: *computable relative to a notation that satisfies the Successor Constraint*. What Shapiro calls "Church's Thesis" entails Church's thesis, as it is normally understood, only when combined with the further claim that every intuitively computable number-theoretic function is computable relative a notation that satisfies the Successor Constraint. Shapiro does not attempt to establish this further claim.

We may summarize this section as follows. If you think we possess an intuitive concept of computing a number from an input, then you should reject a putative analysis of that concept based upon the Successor Constraint. If you do not think we possess an intuitive concept of computing a number from an input, or if you deny that our formal theorizing answers to any such intuitive concept, then you are of course perfectly entitled to follow Shapiro in adopting a formal ersatz based upon the Successor Constraint. In that case, you should not claim that the formal

ersatz analyzes any pre-theoretic concept. Nor should you claim that your position vindicates Church's thesis, since that thesis, as it is typically understood, concerns a pre-theoretic concept.

§5.3 The purely syntactic conception of computation

The conclusion of the previous section naturally leads us to inquire whether we truly possess a pre-theoretic concept of numerical computability, or at least any such concept worth preserving. We can *develop* a concept of numerical computation within our formal theorizing. But, one might urge, we should not ask whether the formal concept corresponds, either extensionally or intensionally, to some intuitive notion. The only intuitive notion of computability to which our theorizing answers is string-theoretic computability. We should therefore reformulate Church's thesis so that it concerns string-theoretic, rather than numerical, computability. Let us call this approach *the purely syntactic conception of computation*.[25]

The most obvious obstacle facing the purely syntactic conception is that current mathematical practice just *does* seem to enshrine a notion of numerical computability. Virtually every textbook on recursion theory takes as its subject matter the computability of number-theoretic functions. As a representative sample, see Boolos and Jeffrey [1], Rogers [19], and Soare [26]. Nor can we dismiss this emphasis upon number-theoretical computability as reflecting an unfortunate conflation between numerical and string-theoretic computability. For most of these same textbooks take great pains to distinguish numbers from symbols, in the context of emphasizing use-mention distinctions. There is little doubt that virtually all contemporary logicians take themselves to possess a bona fide concept *computable number-theoretic function*, which they take to be co-extensive with the concept *recursive function*.

These observations demonstrate that the purely syntactic conception is *revisionist* regarding current mathematical practice and pedagogy. Its revisionism might seem relatively plausible if we also adopted a sufficiently extreme nominalist, fictionalist, or formalist conception of arithmetic, since presumably such a conception would already bar us from taking ordinary mathematical discourse at face value. However, if one is *not* antecedently committed to more thoroughgoing revisionism regarding arithmetic, then revisionism regarding numerical computability should seem quite unpalatable. Once we accept at face value ordinary mathematical talk about the existence of numerical functions distinct from string-theoretic functions, why should we not also accept at face value ordinary mathematical talk about our ability to compute the values of those functions? Typically, we would not hesitate to say that the multiplication algorithm taught in elementary school is a mechanical procedure for computing the product of two numbers, or that the Euclidean algorithm is a mechanical procedure for computing the greatest common divisor of two numbers. Once we accept that there exist a multiplication function and a greatest common divisor function about which we can think and reason, it seems bizarre not to say that we can, by employing the appropriate algorithms, compute the values those functions assume on given inputs. Yet to say so is to deploy an intuitive notion of numerical computability.

Another obstacle faces the purely syntactic conception. When faced with a definition of "numerical computability" as "computable relative to *some* notation," we naturally find it repugnant, since it allows deviant semantic relations like d_X. A good philosophical account must explain this intuitive verdict. The most natural explanation is that the intuitive verdict reflects our grasp of an *intuitive, pre-theoretic* notion of numerical computability. The proposed definition seems "too broad," in that it classifies various functions as computable even though, by Church's

thesis, they are not intuitively computable. Clearly, this explanation invokes the intuitive notion of numerical computability. Moreover, it is difficult to see how the purely syntactic conception of computation can provide a similarly satisfactory explanation. If that conception were correct, then a formal definition of "numerical computability" based upon the Computability, Translation, or Successor Constraints would apparently deserve no greater approbation than the definition "computable relative to *some* notation." For the formal definitions would answer to no pre-theoretic concept against which we could measure them for extensional adequacy.

Despite these obstacles, the purely syntactic conception exerts a powerful appeal. I now want to examine three arguments one might offer in its favor.

One argument runs as follows: humans and computers directly manipulate symbols, not numbers; thus, what humans and computers *really* compute are string-theoretic functions, not number-theoretic functions. Shapiro seems to endorse something like this argument: "[m]echanical devices engaged in computation and humans following algorithms do not encounter numbers themselves, but rather physical objects such as ink marks on paper… Furthermore, mathematical automata, such as Turing machines… have only appropriately constituted strings for inputs and outputs. It follows that, strictly speaking, computability applies only to string-theoretic functions and not to number-theoretic functions" ([20], p. 18).

The argument is fallacious. Its premise (humans and computers directly manipulate symbols, not numbers) does not support its conclusion (strictly speaking, humans and computers compute only string-theoretic functions). At best, the premise establishes that our computations of number-theoretic functions are *mediated* by our computations of string-theoretic functions. It does not follow that all we *really* or *strictly speaking* compute are string-theoretic functions. To conclude this would be analogous to the inference sometimes drawn by the British empiricists

that, since our ideas mediate our perception of the external world, all we really perceive are our ideas. To modernize the analogy, consider Jerry Fodor's version of *the representational theory of mind*, according to which propositional attitudes are relations to propositionally contentful items in the language of thought. As Fodor himself would emphasize, it simply does not follow from this view that what we really think about are items in the language of thought. All that follows is that the language of thought mediates our thinking about extra-mental items. The point persists when the extra-mental items are numbers or number-theoretic functions. More generally, as I conceded in §5.1, some canonical notation, either mental or non-mental, may well mediate our thought about the numbers. If so, the mediating notation is not an obstacle that prevents us from computing numerical functions. On the contrary, it is precisely what enables us to compute those functions. Mediated computation is still computation.

Another argument for the purely syntactic conception observes that an appropriately chosen system of notation, viewed as an ω-sequence, is isomorphic to the natural numbers. One can therefore develop recursion theory over notations rather than numbers. Machtey and Young [12] do so, and the resulting theory is, in all mathematical essentials, equivalent to recursion theory as standardly developed over the natural numbers. What is lost, one might demand, by retreating in this way from numerical to syntactic computability?

The basic problem with this argument is that it provides no reason to doubt that we possess an intuitive notion of numerical computability. It merely insists that, for most or perhaps all mathematical purposes, we can settle for a syntactic surrogate. Yet, if we possess an intuitive concept of numerical computability, surely we should try to clarify its extension and intension. We should elucidate what it is to compute a number-theoretic function, and we should specify which such functions are computable. One can dismiss these questions as uninteresting. But a

brusque dismissal provides no reason for thinking the questions ill-formed or misguided, so it will hardly persuade the many philosophers and logicians who find them intrinsically interesting.

A final argument for the purely syntactic conception cites the *historical motivations* underlying research on computability in the 1930s. Much of that research was prompted mainly by syntactic concerns. For instance, one might argue that Turing's interest in computability stemmed most fundamentally from the decision problem for first-order logic, while Gödel sought primarily to delimit the class of formal systems. Thus, the argument goes, Church's thesis was introduced to address computations defined over symbols, not over numbers. Even if there *is* a legitimate notion of numerical computation, we need not concern ourselves with trying to explicate it. We can settle for syntactic computability, the only notion relevant to those problems that sparked our initial interest in Church's thesis.[26]

Even if this historical analysis correctly describes the motivations of Turing and Gödel, it strikes me as rather slanted. Kleene's historical retrospective [11] offers a very different interpretation, highlighting the following question as central: "What number-theoretic functions… are computable?" (p. 21). It also seems clear that Post, who placed great emphasis upon the representational relation between symbols and what they symbolize, would have rejected the purely syntactic conception of computation. More importantly, though, observations about the historical motivations of various logicians, no matter how eminent and brilliant, cannot support the purely syntactic conception. Historical observations cannot show that we lack a legitimate pre-theoretic concept of numerical computation or that this concept merits no mathematical or philosophical clarification. Whether or not Turing and Gödel were exclusively concerned with syntactic computability, *we* are not constrained to examine only those topics that interested Turing and Gödel. Even if Turing did not intend to explicate numerical computability,

we can ask whether his work yields a satisfactory explication. Most contemporary recursion theory textbooks do in fact ask this question, usually formulated in extensional rather than intensional terms. As I have urged, the question becomes almost inescapable if we reject radically revisionary versions of nominalism, fictionalism, and formalism.

On balance, then, the purely syntactic of computation strikes me as rather unattractive. I do claim to have refuted it. But I would urge that, unless we have already adopted a more thoroughgoing revisionary conception of the natural numbers, its costs far outweigh its benefits.

Once we reject the purely syntactic conception, we must choose among the following three positions: Turing's account captures neither the extension nor the intension of the intuitive concept *computable number-theoretic function*; or Turing's account captures the intuitive concept's extension *and* its intension; or Turing's account captures the intuitive concept's extension but not its intension. Virtually no one espouses the first position. Gandy and Sieg espouse the second. I espouse the third.

§5.4 Who cares about conceptual analysis?

My argument lacks any significance for research within recursion theory and computer science. A typical application of Church's thesis within these fields runs as follows: one shows that a function is not recursive; by Church's thesis, one concludes that the function is not computable. This argument requires only the truth of Church's thesis. It does not presuppose any conceptual-analytic claims. Since I endorse Church's thesis, my discussion generates no implications for the mathematical study of computability. Accordingly, one might question what interest attaches to my position. Given that Turing's account is extensionally adequate, and given that it provides a foundation for the mathematical study of computation, what more could we reasonably require? Why not simply relish it as an instance of Quinean regimentation or

Carnapian rational reconstruction? Who cares if Turing's account fails to provide a genuine conceptual analysis? Anyway, doesn't the "paradox of analysis" show that virtually all putative analyses fall similarly short?

I have formulated my principal thesis in terms of conceptual analysis, because many other commentators frame the issue in these terms. But we can restate my position without invoking notions like synonymy, conceptual analysis, etc.

Turing centers his discussion around the following question: "What are the possible processes which can be carried out in computing a number?" ([27], p. 135). A conceptual analysis of number-theoretic computability must answer this question. But Turing's question is intrinsically interesting. Even if we abandon any aspirations towards conceptual analysis, we should still attempt to characterize all numerical computational processes. We should try to isolate what, in general, is involved in computing a number from an input. As Sieg puts it, "Turing asked in the historical context in which he found himself the pertinent question, namely, what are the possible processes a human being can carry out (when computing a number or, equivalently, determining algorithmically the value of a number-theoretic function)? The general problematic required an analysis of the idealized capacities of calculators" ([21], p. 395). I agree. My point is that, if we restrict ourselves to constraints upon mechanical manipulation of syntax, then we fail to provide an adequate account of those "idealized capacities," and we thereby fail to answer Turing's question. Indeed, as I argued in §3, a treatment that focuses solely upon constraints governing mechanical manipulation of syntax is not even extensionally adequate, since it does not disbar deviant semantic relations like d_X.

We can restore extensional adequacy by stipulating a particular computable semantics, such as d_1. But we thereby abandon any pretense of offering a general characterization of "the

idealized capacities of calculators." There are infinitely many notations in which calculators can compute number-theoretic functions, and a characterization based upon a single privileged notation will not encompass computations conducted relative to alternative notations.

According to Sieg, a primary "lesson we owe to Turing" is that a characterization of numerical computability should emphasize *symbols*: "[t]o investigate calculations is to analyze symbolic processes carried out by calculators" ([21], p. 290). Maybe so. Intuitively speaking, though, what all symbolic numerical computations have in common is the mechanical manipulation of symbols that are themselves mechanically interpretable. Turing may provide a general theory of mechanical symbol manipulation, but he says virtually nothing about mechanical symbol interpretation. Only when we provide an account of the latter will we possess a general theory of what all possible computational symbolic processes have in common.

In his posthumously published diary, Post pursues a general theory of all possible computations: "a complete analysis… of all the possible ways in which the human mind could set up finite processes" ([17], p. 408). He urges that such an account requires "psychological analysis of the mental processes involved in combinatory mathematical processes" (p. 418). Much of what Post says in the diary is extremely gnomic. However, his emphasis on *the mind* and *mental processes* strikes me as a salutary corrective to the excessive focus upon syntactic manipulation that characterizes not only Turing's exposition but most other modern discussions of computation. The syntactic approach has proved enormously fruitful. Without it, recursion theory and computer science would not exist, at least in anything resembling their current form. Unfortunately, its amazing success has encouraged the conclusion that we can give an *entirely* syntactic account of computation. This conclusion strikes me as fundamentally mistaken. A general theory of numerical computability must eventually breach the circle of syntactic notions,

addressing with suitable generality the cognitive and representational relations we bear to numbers. Any account that shirks this obligation leaves behind an unexplained residue of computational mental activity.

---

[1] Shapiro's example crucially assumes that semantic relations are injective. If we abandon this constraint, then every number-theoretic function *f* is computable relative to some uniform semantics, as illustrated by the following technique. Consider a repetitious enumeration of the natural numbers: 0, 1, *f*(0), 2, *f*(1), *f*(*f*(0)), 3, *f*(2), *f*(*f*(1)), *f*(*f*(*f*(0))), etc. Let *c* be the (non-injective) semantics that carries $\underline{n}$ to the *n*th element in this enumeration. Then *f* is Turing-computable relative to *c*.

[2] As noted in Soare [25], we should distinguish between what Soare calls "Church's thesis" and what he calls "Turing's thesis" (p. 296). The former, which follows Church's original published formulation, employs the equation calculus. The latter employs the concept of Turing-computability. The two theses are extensionally equivalent. But they are intensionally distinct, since they employ different concepts. In Soare's terminology, I am discussing "Turing's thesis," not "Church's thesis." However, what I say would apply to either thesis.

[3] For discussion of some difficulties surrounding the right-to-left direction, see Parsons [16]. Parsons ultimately endorses this direction.

[4] I have relabeled the five constraints. As Sieg notes, we may eliminate the determinacy constraint (a), since deterministic machines can simulate non-deterministic machines.

[5] Turing assumes that the paper is divided one-dimensionally into squares. He claims that this assumption induces no loss of generality, even though people normally calculate on two-dimensional paper. Sieg and Byrnes [23] attempt to vindicate Turing's claim, providing a general treatment that includes two-dimensional computations. Presumably, one could imagine even more general treatments.

[6] For discussion of this point, and related issues, see Copeland [2].

[7] Post raises a related worry in Post [17] (p. 344). For defense of Turing on this point, see the various papers cited in the References written or co-written by Sieg.

[8] Sieg clarifies many of the obscurities in Turing's argument. See especially Sieg and Byrnes [23].

[9] Turing [27] concerns computable real numbers, not computable functions from the natural numbers to the natural numbers. Thus, Turing does not interpret strings of strokes as denoting natural numbers. This aspect of Turing's discussion reinforces my point, which is that Turing provides no general criterion for an "acceptable" mapping from Turing machine syntax to the natural numbers. Turing's focus on computable real numbers also highlights a more general theme: d_1-d_3 do not constitute "default" or "canonical" interpretations for Turing machine syntax.

[10] I am grateful to an anonymous referee who suggested this objection.

[11] For ease of exposition, I have somewhat altered the quote. What Boolos and Jeffrey actually say is that they will define Turing-computability as computability within various constraints, only one of which concerns monadic notation. The alteration does not affect my argument in this paragraph.

[12] The literature contains various arguments for Church's thesis besides Turing's argument based upon constraints (a)-(e). For present purposes, I merely note that, like many other commentators, I generally find these arguments much less convincing than the argument based upon constraints (a)-(e).

[13] I am grateful to an anonymous referee for pressing this point.

[14] I am greatly indebted in this paragraph to Yiannis Moschovakis, who pointed out the streamlined argument in the main text. Previously, I was employing a much more complicated argument. Needless to say, any remaining errors are my own.

[15] See especially Sieg and Byrnes [23] (p. 61). And see Sieg [21], pp. 397-8, which asserts that the extensional equivalence between "calculability of number-theoretic functions" and "calculability by computor satisfying boundedness and locality conditions" is "given by conceptual analysis."

[16] W. V. Quine famously challenges the legitimacy of notions like "synonymy." Accordingly, he rejects the idea "that analysis must consist somehow in the uncovering of hidden meanings" ([18], p. 259). Quine would dismiss our third constraint upon conceptual analyses as hopelessly unclear. I am somewhat sympathetic to this Quinean criticism. However, Quine's position does not undermine my discussion. My point here is this: *if* we demand that a good conceptual analysis "capture the meaning" of the original concept, *then*, whatever exactly that means, Turing's discussion does not plausibly provide a satisfactory conceptual analysis of number-theoretic computability.

[17] We can define the notion of *formal system* by invoking mechanical manipulation of syntax. Thus, Turing's work yields a conceptual analysis of *formal system*. As Gödel puts it, "due to A. M. Turing's work a precise and unquestionably adequate definition of the general notion of formal system can now be given" ([9], p. 195).
[18] I am grateful to comments from an anonymous referee that prompted the addition of this paragraph.
[19] Perhaps one should not read too much into this passage. Post's diary is notable for its highly fragmentary and elusive character, sometimes bordering on the mystical. However, it seems undeniable that Post was preoccupied by what we would now call the semantic relation between symbols and numbers, as in the following passage: "… Notion of meaning bothers me. Put it as subconscious perception of things associated with symbols" (p. 428).
[20] Although this is in some ways a rather crude argument for the thesis that numbers are pure abstract entities, I think that it has considerable force. Parsons provides far more subtle and sophisticated arguments for the thesis in [15] and in his unpublished manuscript *Mathematical Thought and its Objects*. In the unpublished Whitehead lectures "Logicism, Wittgenstein, and *de re* Beliefs about the Numbers," Saul Kripke flirts with the view that the numbers are individuated by their relations to syntactic-types, and he sometimes seems to accept the radical consequence that a change in notation entails a change in the numbers.
[21] Kripke explores these topics in his unpublished Whitehead Lectures. See also Martin-Löf [13] and Parsons [14].
[22] For general discussion of the language of thought, see Fodor [4]. As far as I know, no one has developed the view, specifically with respect to mathematics, in very great detail.
[23] In his unpublished Whitehead lectures, Kripke suggests that computability is not a sufficient condition for being an adequate canonical notation. He urges that an adequate canonical notation must satisfy some stricter constraint. Kripke mentions finite-automata-decidability. Other ideas would include primitive recursiveness or polynomial-time computability. Might such a proposal help alleviate the circularity engendered by the Computability Constraint? I doubt it. I suspect that any plausible strengthening of the Computability Constraint would still invoke the intuitive notion of computability. After all, if we demand that some notation be computable in a certain manner, or within certain limits, then we do not *dispense* with the intuitive concept of computability; we merely employ it in conjunction with certain additional restrictions. (Compare: when analyzing "*X* knows that *p*," it would be circular to include the clause "*X* believes that *p* based upon propositions that he knows through some especially reliable mechanism.") Still, I must admit that this seems like a somewhat promising line of response to my argument and that an adequate assessment would require much more extensive discussion.
[24] What if we supplement the Successor Constraint with the demand that co-reference between numerals be mechanically decidable? This supplemented criterion is extensionally equivalent to the Computability Constraint. But is it a plausible candidate for conceptual analysis? I think we appreciate its extensional adequacy only by noting that it entails the Computability Constraint. We observe that, if a notation satisfies the Successor Constraint, and if co-reference between numerals is decidable, then there exists a mechanical procedure for interpreting the notation; we conclude that the notation is suitable for computation.
[25] I am grateful to an anonymous referee for suggesting this objection.
[26] I am grateful to an anonymous referee for pressing this point.

References

[1] Boolos, G. and R. Jeffrey, *Computability and Logic*, 2nd ed., Cambridge University Press, Cambridge, 1980.

[2] Copeland, J., "The Church-Turing Thesis." *Stanford Encyclopedia of Philosophy, (Fall 2002 Edition)*, ed. E. N. Zalta. URL = <http://plato.stanford.edu/archives/fall2002/entries/church-turing/>.

[3] Davis, M., ed. *The Undecidable*, Raven Press, New York, 1965.

[4] Fodor, J., *The Language of Thought*, Thomas Y. Crowell, New York, 1975.

[5] Frege, G., "Function and Concept," trans. P. Geach, in *The Frege Reader*, ed. M. Beaney, Malden, Blackwell, 1997.

[6] Gandy, R., "Church's Thesis and Principles for Mechanism," *The Kleene Symposium*, ed. J. Barwise, et al., North Holland Publishing Company, Amsterdam,

1980, pp. 123-148.
[7] Gandy, R., "The Confluence of Ideas in 1936," *The Universal Turing Machine: A Half-Century Survey*, ed. R. Herken, Kammerer and Unverzagt, Hamburg, 1988, pp. 55-111.
[8] Gödel, K., *Collected Works, Volume I*, ed. S. Feferman, et al., Oxford University Press, Oxford, 1986.
[9] Gödel, K., "On Formally Undecidable Propositions of *Principia Mathematica* and Related Systems I," rpt. in *Collected Works, Volume I*, trans. Jean van Heijenoort, pp. 145-195.
[10] Gödel, K., "On Undecidable Propositions of Formal Mathematical Systems," rpt. in *Collected Works, Volume I*, pp. 346-371.
[11] Kleene, S. "Turing's Analysis of Computability, and Major Applications of It," *The Universal Turing Machine: A Half-Century Survey*, ed. R. Herken, Kammerer and Unverzagt, Hamburg, 1988, pp. 17-54.
[12] Machtey, M., and Young, P., *An Introduction to the General Theory of Algorithms*, Elsevier North-Holland, New York, 1978.
[13] Martin-Löf, P., *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.
[14] Parsons, C., "Intuition and Number." *Mathematics and Mind*, ed. A. George, Oxford University Press, New York, 1994.
[15] Parsons, C., "Mathematical Intuition," *Proceedings of the Aristotelian Society*, vol. 80 (1980): pp. 145-168.
[16] Parsons, C., "What Can We Do "in Principle'?", in *Logic and Scientific Methods*, ed. M. L. Dalla Chaira, et al., Kluwer, Dordrecht, 1997.
[17] Post, E., "Absolutely Unsolvable Problems and Relatively Undecidable Propositions Account of an Anticipation," in *The Undecidable*, ed. Martin Davis. pp. 338-433.
[18] Quine, W. V., *Word and Object*, MIT Press, Cambridge, 1960.
[19] Rogers, H., *Theory of Recursive Functions and Effective Computability*, MIT Press, Cambridge, 1987.
[20] Shapiro, S., "Acceptable Notation," *Notre Dame Journal of Formal Logic*, vol. 23 (1982), pp. 14-20.
[21] Sieg, W., "Calculations by Man and Machine: Conceptual Analysis," in *Reflections on the Foundations of Mathematics*, ed. W. Sieg, et. al., Association for Symbolic Logic, Natick, 2002, pp. 390-409.
[22] Sieg, W, "Mechanical Procedures and Mathematical Experience." *Mathematics and Mind*, ed. A. George, Oxford University Press, Oxford, 1994.
[23] Sieg, W. and J. Byrnes, "Gödel, Turing, and K-Graph Machines." *Logic and Foundations of Mathematics*, ed. A. Cantini, et al., Kluwer, Dordrecht, 1999.
[24] Sieg., W. and J. Byrnes, "K-Graph Machines: Generalizing Turing's Machines and Arguments," *Gödel '96*, ed. Petr Hájek, Association for Symbolic Logic, Natick, 1996.
[25] Soare, R., "Computability and Recursion," *Bulletin of Symbolic Logic*, vol. 2 (1996): pp. 284-321.
[26] Soare, R., *Recursively Enumerable Sets and Degrees*, Springer-Verlag, New York, 1980.

[27] Turing, A., “On the Computable Numbers, with an Application to the Entscheidungsproblem,” *The Undecidable*, ed. M. Davis, pp. 116-154.