

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

The Conceptual Development of Nondeterminism
in Theoretical Computer Science

Walter Warwick

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of History and Philosophy of Science
Indiana University

March 2001

UMI Number: 3005421

UMI[®]

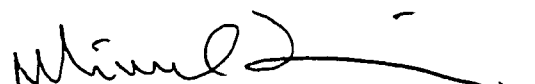
UMI Microform 3005421

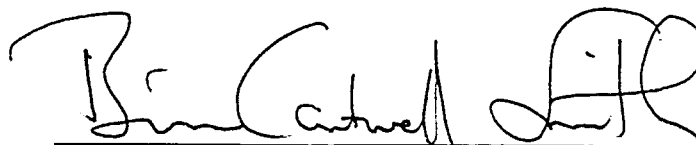
Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

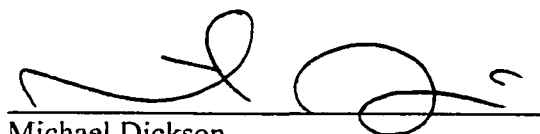
Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

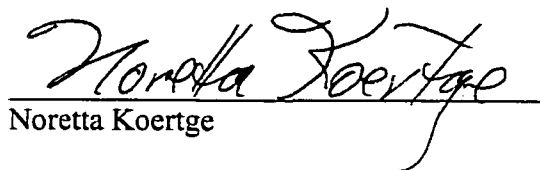

Michael Friedman, Ph.D.


Brian Cantwell Smith

Doctoral
Committee


Michael Dickson

February 9, 2001


Noretta Koertge

This work is dedicated to my Dad for showing me how much fun it can be to think about hard problems, to my Mom for being kind and good-humored (but this ain't about binary numbers, Mom), and especially to my wife, Cathy, for putting up with me while I crawled through this process.

I would like to acknowledge Michael Dickson, Brian Smith, Michael Friedman and Noretta Koertge for all their help from afar as I completed this essay. Out of sight is often out of mind, which makes me all the more grateful for their willingness to comment on the work that would appear from me out of the blue. Also, I'd like to thank Becky Wood for all her help with the last minute details, her willingness to put out all the little fires that started with my neglect and her overall grace in dealing with my panicked, long-distance calls. Finally, I would like to thank Ron Laughery, Sue Archer and Rick Archer for being both flexible and enthusiastic as I tried to balance job and dissertation.

CONTENTS

Chapter 1	Framing the Problem	
§1	Who Wants to be a Millionaire?	1
§2	Four Intuitions about Nondeterminism	4
	2.1 Nondeterminism as an example of extensional equivalence	4
	2.2 Nondeterminism as a natural reflection of the mathematician's behavior	6
	2.3 Nondeterminism as a mathematically interesting construct . .	7
	2.4 Nondeterminism as a physical process	8
§3	Defining the Problem	10
§4	An Outline of the Essay	13
Chapter 2	Some Historical and Technical Background	
§0	Overview	19
§1	The Original Motivation for the Turing Machine	19
§2	The Turing Machine as an Unbounded Model of Computation . .	26
§3	The Turing Machine and Resource-Bounded Computation.	31
§4	Formal Grammars	36
Chapter 3	Philosophical Concerns about Nondeterministic Algorithms	
§0	Overview	38
§1	Introduction	38
§2	Getting to the Center of the Tangle.	41
§3	One Idea from Three Traditions	60
§4	Where Do We Go From here?	66
	4.1 Looking at <i>alternation</i> to understand nondeterminism	68
	4.2 Redefining nondeterminism	69
	4.3 Moving beyond the Turing machine.	72
Chapter 4	A Second Look at the Received History	
§0	Overview	75
§1	Introduction	75
§2	Turing's Nondeterministic Turing Machines	78
§3	Rabin and Scott's Nondeterministic Automata	90
§4	1959-1964 and Kuroda's Nondeterministic Linear-bounded Automaton	97

§5	1965-1972	104
§6	Conclusions	116
Chapter 5	Computer Science and the Philosophy of Science	
§1	Looking Back and Looking Forward	118
	Bibliography	130

I. Framing the Problem

1. Who Wants to be a Millionaire?

There are at least two ways to become a millionaire. The first is to embarrass yourself on any one of a number of nationally televised game shows. The second is to solve the P versus NP problem (see http://www.claymath.org/prize_problems/index.htm for details). For those unwilling to pander to the stupid and greedy, the choice here is obvious, but the problem is not.

On the Claymath homepage, the P versus NP problem is described in this way:

It is Saturday evening and you arrive at a big party. Feeling shy, you wonder whether you already know anyone in the room. Your host proposes that you must certainly know Rose, the lady in the corner next to the dessert tray. In a fraction of a second you are able to cast a glance and verify that your host is correct. However, in the absence of such a suggestion, you are obliged to make a tour of the whole room, checking out each person one by one, to see if there is anyone you recognize. This is an example of the general phenomenon that generating a solution to a problem often takes far longer than verifying that a given solution is correct. Similarly, if someone tells you that the number 13,717,421 can be written as the product of two smaller numbers, you might not know whether to believe him, but if he tells you that it can be factored as 3607 times 3803 then you can easily check that it is true using a hand calculator. One of the outstanding problems in logic and computer science is determining whether questions exist whose answer can be quickly checked (for example by computer), but which require a much longer time to solve from scratch (without knowing the answer). There certainly seem to be many such questions. But so far no one has proved that any of them really does require a long time to solve; it may be that we simply have not yet discovered how to solve them quickly. Stephen Cook formulated the P versus NP problem in 1971.

In addition to the informal description, there is a link to a "technical" description of the problem given by Cook himself:

The P versus NP problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time.

At first blush, it is not clear that the two descriptions refer to the same problem.

The first description talks about quickly checking solutions while the second invokes a recondite notion of nondeterminism (the "N" in "NP"). Obviously, the two descriptions are intended to complement one another; the former primes the intuitions while the latter refers to the formal framework that gives rise to the problem. Still, it is striking that the two descriptions would be so disparate. In fact, there is such a disconnection between the two descriptions that we might wonder how the formal and informal accounts of the P versus NP problem are related. The conspicuous lack of explanation suggests that the relation is obvious. Asking for further explanation here is like shopping for luxuries; if you have to ask, you probably don't understand the problem. But we should not shy away from asking. Indeed, the P versus NP problem is rooted in a tradition that began with a painstaking effort to make the relation between the informal and the formal as clear as possible. In this essay, we will examine the notion of nondeterminism that underlies the P versus NP problem and argue that, far from being obvious, the relation between the formal and informal description of the P versus NP problem betrays an unexpectedly complicated history in the development of theoretical computer science.

Before we launch into a detailed examination of nondeterminism let us first define the deterministic Turing machine and then make a few observations. Intuitively, we can think of a Turing machine as consisting of two parts: a finite *control* and a *tape* with an indefinite number of *cells*. The control consists of a finite collection of internal

states and a read/write head that scans the tape one cell at a time. Each cell contains a single symbol from a finite alphabet. Turing machines compute in a step-wise manner. At each step, the control assumes exactly one of its internal states and scans the contents of a single cell. Depending on the current state and the contents of the cell, the control will overwrite the contents of the cell (perhaps "printing" a blank or even rewriting the old symbol) and then move either one cell to the right or to the left before assuming a new internal state and beginning the next step in the computation. More formally, Turing machines are defined as ordered tuples of some sort.¹ For instance, Papadimitriou (1994) defines a Turing machine, M , as a quadruple, $M = \langle K, \Sigma, \delta, s \rangle$ where K is a finite set of states, Σ is a finite set of symbols, δ is a *transition function* mapping states and symbols to states and behaviors (where a behavior is an atomic action such as printing a new symbol or moving left or right) and s is the designated "start" state in K . The behavior of a Turing machine, and hence the machine itself, is completely determined by its transition function.

Now, let us make a few observations. First, we should note that the intuitive picture of a Turing machine is quite compelling, but as far as the theory of computation is concerned, "the ultimate characterization [of a Turing machine] is entirely mathematical" (Rogers 1967, p. 13). Like the two descriptions of the P versus NP problem, the Turing machine provides us another curious example of the interplay between informal

¹ Presentations differ in terms of the tuple used to define Turing machines; sometimes machines are defined as quadruples (Papadimitriou 1994), other times as quintuples (Bovet and Crescenzi 1994) or even septuples (Hopcroft and Ullman 1979). Such differences arise from idiosyncratic notions of what to count as an atomic action, but they are theoretically inconsequential.

intuitions and formal characterizations. Second, having defined the deterministic Turing machine in terms of a transition *function*, we define a nondeterministic Turing machine in terms of a transition *relation*, where the current state and symbol do not uniquely define the next state and action. Thus the technical distinction that Cook refers to between a deterministic and nondeterministic algorithm in his formal description of the P versus NP problem is clear and unambiguous. By contrast, the informal description of the P versus NP problem turns on our "*understanding of nondeterminism*" (Papadimitriou 1994, p. 45, emphasis added). It is one thing to offer a formal definition, but quite another to understand the notion thus defined. In fact, there are at least four different ways of thinking about nondeterminism.

2. Four Intuitions about Nondeterminism

2.1 Nondeterminism as an example of extensional equivalence

First, we can think about nondeterminism as further evidence that the formal notions of *computable* are sufficiently general. In particular, we can point to the fact that nondeterministic Turing machines are no more powerful than deterministic Turing machines with respect to the class of functions they compute as another example in a long list of extensional equivalencies between seemingly disparate models of computation. (We will outline a proof of the equivalence of deterministic and nondeterministic Turing machines in Chapter 3.)

There is an obvious precedent for such a view of nondeterminism. By the mid 1930s there were several well-known formalisms presented as analogues of the informal notion of an algorithm: there were the general recursive function due to Herbrand and Gödel; the λ -definable functions due to Church; and, finally, the functions computable

by a Turing machine. Although each of these formalisms was put forward to answer the same question (viz., what is it to be algorithmic?) they were strikingly different. Moreover, it seemed that the arguments that any *one* of these formalisms could exhaust the informal notion of algorithm would ultimately be philosophical. Consequently, it was rather surprising that almost as soon as the three formal notions were on the table, rigorous equivalence proofs were given showing that the recursive, effectively calculable and computable functions are coextensive.² These equivalence proofs suggested that the formal accounts adequately captured the informal notion of an algorithm and thus the so-called Church Turing thesis was born.

Likewise, nondeterministic computation seems quite different than deterministic computation. Indeed, we will see below that deterministic Turing machines compute in the plodding, uninspired and exhaustive manner one might expect from a machine. Nondeterministic machines, by contrast, "guess" as they compute and, moreover, they always guess correctly. The fact that nondeterministic machines are no more powerful than deterministic machines is as striking as any of the original equivalencies established in the '30s, and it leads to a view of nondeterminism as further evidence for the Church Turing thesis. Once we begin thinking in such terms, it is hard not to think in the terms that originally motivated the Church Turing thesis, namely, what is it to work algorithmically, to follow a set of instructions or to behave like a machine.

² In a series of strange historical twists, it was Church (not Gödel or Herbrand) who argued for the adequacy of the general recursive functions as a formal account of effectively computable; it was Kleene (and not Church) who demonstrated the equivalence of the λ -definable and general recursive functions; and it was Gödel who publicly celebrated Turing's account.

2.2 Nondeterminism as a natural reflection of the mathematician's behavior

Second, nondeterminism can be seen as a reflection of what humans actually do when working with a formal system (e.g., a system of natural deduction). Unlike the intuitions associated with extensional equivalence, intuitions about nondeterminism as a more natural model of computation speak directly to the question of whether a formal model of computation can do justice to the intuitive notion of an algorithm. Indeed, Turing himself recognized that any argument suggesting that the formal notion of computable exhausts the informal notion of algorithm would be essentially intuitive (Turing 1965a, p. 135). Although it seems counter-intuitive, we will argue below that nondeterminism actually deepens the intuitive appeal of Turing's account. Moreover, we will gain an appreciation for the profound impact such appeals to intuition had. In the meantime, we will simply note that mathematicians since Gödel³ have celebrated the Turing machine as a model that finally made the mathematical notion of an *algorithmic* procedure mathematically precise. There were, of course, ideas about recipes and rules long before Turing, but his analysis was particularly compelling. Even if Church had the claim on priority, the idea that the informal notion of algorithm might be characterized by *any* formal notion (e.g., the general recursive à la Church, or the Turing computable) really gained currency with Turing's work.

³ See, e.g., (Gödel 1965b) or the 1964 Postscriptum to (Gödel 1965a) where Gödel states, "In consequence of later advances, in particular of the fact that, due to A.M. Turing's work, a precise and unquestionably adequate definition of the general system now be given, the existence of undecidable arithmetical propositions and the non-demonstrability of the consistency of a system in the same system can now be proved rigorously for every consistent formal system containing a certain amount of finitary number theory" (emphasis in the original). Details for such a proof can be found in (Kleene 1988).

Although Turing is not credited for the introduction of the nondeterministic machine that bears his name, we will see in Chapter 4 that it was, in fact, Turing who introduced nondeterminism and we will argue that he did so to make his account of computation more familiar and hence more intuitively compelling.

We will also see a related intuition surface in the 1960s with the birth of complexity theory. Complexity theory began because the study of algorithms eventually had to "deal realistically with the quantitative aspects of computing" (Hartmanis and Hopcroft 1971, p. 444). Oddly enough, this requirement initially drove researchers to a model of computation more restricted than the Turing machine and, by extension, more restricted than the nondeterministic Turing machine. Nevertheless, the talk of constraining the Turing machine in the 1960s is reminiscent of Turing's original discussion of nondeterministic computation insofar as each was intended to tie the formal to the actual and familiar.

2.3 Nondeterminism as a mathematically interesting construct

Third, nondeterminism can be viewed as a useful way of classifying problems. The intuition here is rooted in a sense of mathematical utility and has more to do with how hard it is to compute than what it is to compute.⁴ More specifically, if we are given a

⁴ We should note that the Turing machine has always been studied as a purely mathematical model. For instance, in Kleene's (1988) discussion of Turing machines he states that "so far as we [i.e., mathematicians] are interested in it," the behavior of a Turing machine can be completely described by either a table or a transition function. Indeed, the arithmetization of the Turing machine, whereby machines are encoded by natural numbers, is the crucial step in developing interesting theory. The point we are making here, however, is that even if Turing machines themselves have always been studied as mathematical objects, there is an important difference between appealing to the Turing machine as a model of computation and viewing it as an interesting, but

robust model of computation, like the Turing machine, and if we worry about how long it takes to solve problems, then nondeterminism seems to impose structure on the class of computable problems. In this light, nondeterminism is not meant to buttress a formal account of computable nor is it intended to be a reflection of what mathematicians, much less machines, actually do when they compute. Nondeterministic machines are a patently "*unrealistic* model of computation" and they "break our chain of 'reasonable' models of computation" (Papadimitriou 1994, p. 45, emphasis in the original). Moreover, the operation of a nondeterministic machine is either described metaphorically or simply left to the "imagination" of the reader.⁵ The point is not how such nondeterministic machines work, but rather that nondeterminism can be used as a something of a proxy for a class of problems—often described as those with "succinct certificates" or as "easily verified." In Chapter 3 we will see that mathematical intuitions about nondeterminism lead to an unexpected result about the manner in which nondeterministic algorithms compose. In Chapter 4 we will note with even greater surprise that the existence of those problems now characterized by nondeterminism has led some to suggest that it is time to reformulate the Church Turing thesis.

2.4 Nondeterminism as a physical process

Finally, nondeterminism can be understood in a physical sense as a random or irreversible process. For example, we will see in Chapter 3 that Rogers argues that a deterministic algorithm should not depend on the roll of a dice. Likewise, although

perhaps, inessential label for a class of problems.

⁵ See, e.g., (J. D. Smith 1989, especially pp. 296-301) who talks about nondeterministic machines "guessing" what to do next, or (Bovet and Crescenzi 1994, p.19) who struggle

Minsky acknowledges the question whether "noise or other physical realities of a probabilistic nature can be tolerated by the theory," he never addresses the question (1967, p. 299) . Historically, intuitions about nondeterminism as a physical process helped define what computation was *not*. Even if the physical implementation of a computer was subject to vicissitudes like overheating vacuum tubes and faulty wiring, the theory of the '50s and '60s was characterized by an assumption that each step in an algorithm would be completely and uniquely specified by the step that preceded it. The process of computation was thus deterministic in the sense that it was completely reproducible and entirely predictable (at least theoretically). By excluding the physical sense of nondeterminism, researchers made room for an interesting theory that would otherwise be lost in the practical details of electrical engineering. In this respect, the development of theoretical computer science is like the development of any other scientific theory insofar as it depends on a certain degree of simplification and abstraction from the real world. But we will also see in Chapter 3 that some are now advocating the possibility of exploiting *physically nondeterministic* systems for efficient solutions to problems that are thought to be theoretically intractable. In an ironic twist, the sense of nondeterminism that was once excluded from the theory might someday crack complexity theory's most difficult problems. It is an interesting question whether such a development would advance theory or obviate it.

3. Defining the Problem

There are four very different intuitions all lumped together under the rubric of

to imagine what it would look like for a nondeterministic machine to compute a function.

nondeterminism. The multiplicity of these intuitions does little to help us understand the relation between the formal and informal accounts of the P versus NP problem. In fact, when we look closely at these intuitions we see our understanding of nondeterminism beginning to splinter. For instance, how can we reconcile an intuition about extensional equivalence—an intuition that is intended to buttress the Church Turing thesis and plays on our common intuitions about algorithms—with intuitions about mathematical utility that have ultimately led some to suggest that the Church Turing thesis should be reformulated. Likewise, what should we say when our intuitions are motivated by the need to make the theoretical more familiar, while at the same time an assumption of nondeterminism leads to a model of computation that is patently unrealistic. Finally, can we tolerate the possibility that intuitions about nondeterminism as a physical process might lead to a situation where theoretical intuitions are moot?

In this essay, we address such questions as we try to disentangle the various intuitions associated with nondeterminism in the conceptual development of theoretical computer science. To do this we must address two issues. First, insofar as there is a received history to examine, we will find a presumption of continuity but surprisingly little in the history itself to substantiate that presumption. That is to say, references on nondeterminism consistently point to the same names and papers, but when we trace this bibliographic trail backwards we find a host of ellipses and inconsistencies. The classic papers on the subject do not always say exactly what we'd expect them to say given their place in the received history, and, moreover, there are papers that bear directly on our understanding of nondeterminism that have largely gone unnoticed. Obviously, theory

marches on, but making philosophical sense of it is hard to do in the face of its fragmented history. Second, the presumption of historical continuity leads quite naturally to a presumption of conceptual continuity. In particular, it might be said that the four intuitions about nondeterminism we identified above are really not so different, but rather that they are complementary and represent an ever sharper and more complete understanding of a single phenomenon. We will challenge that claim throughout this essay.

In the meantime, however, we can at least make a *prima facie* case to rebut the presumption of conceptual continuity. Even if we ignore the historical ellipses and incongruities, there is an obvious and true story to tell about the development of theoretical computer science. It begins in the 1930s with the study of what could be achieved *algorithmically* or *effectively*. Research in the theory of recursive functions and effective computability was vigorously pursued for some thirty years. Then, for a variety of reasons, the theoretical emphasis shifted away from the study of what could be computed in an absolute sense to a more finely grained study of computational complexity; the interesting question was no longer what could be computed, but rather how hard it would be to compute. Answers to these kinds of questions depended on a measure of computation difficulty that was traditionally and quite naturally given in terms of number of steps taken or the amount of tape used by a Turing machine. For this reason, complexity theory is often presented as a natural extension of recursion theory and, hence, a continuation of the study of what can be achieved algorithmically. It turns out, however, that complexity theory is now driven by three variations to the traditional

Turing machine model: there are nondeterministic Turing machines, oracle machines and alternating machines.

As we indicated above (and will argue throughout this essay), it is not clear that the variety of intuitions associated with nondeterminism can be reconciled. A similar comment can be made about both oracle computation and alternating machines. For instance, oracle computation is performed by an otherwise normal Turing machine with a special additional state called the *query state*. At any point during its computation an oracle machine can enter the *query state*, compute certain characteristic functions in an instant, and then go on with its regular computation using the oracle information as needed. The idea is due to Turing (1965b) and was originally intended to illuminate implications about Gödelian incompleteness. In complexity theory, however, oracle machines underwrite curious methodological theorems about solving the P versus NP problem.⁶ Today's use of oracle machines in complexity theory is nothing like the use to which they were originally put by Turing. Moreover, it is hard enough to reconcile our intuitions about the P versus NP problem, much less meta-theoretic results about the problem, with the more pedestrian intuitions about algorithms that presumably underlie complexity theory. Obviously, there is a story to tell about how we got from the computability theory of the '30s to the complexity theory of today.

In a similar vein, the notion of an alternating machine is not so much about

⁶ These results are quite striking, for they assert that neither simulation nor diagonalization can be used to settle the most general statement of the P versus NP problem. These two proof techniques are the bread and butter of recursion and complexity theory, and without them it is hard to imagine how the P versus NP problem

machines at all, but, rather, about the so-called computation trees we associate with Turing machines. We will discuss computation trees in more detail in Chapter 3, but for now it is enough to note that alternation is proposed as a generalization of nondeterminism whereby a Turing machine alternates between "existential" and "universal" behaviors in its computation. Intuitively, one might think of alternation in the same way that one thinks of a winning strategy in a game like chess: do I have a move, such that for every possible counter move, I have another move etc. What is striking about alternating machines is that they lead to descriptions of complexity classes (classes of problems) that make no reference to machines. The notion of alternation is clearly a long way from intuitions about machines.

The foregoing remarks are not intended to suggest that there is no coherent story to tell about the conceptual development of theoretical computer science, but rather that there is a story worth telling. We now outline the story we will tell.

4. An Outline of the Essay

The work we will do here is philosophical, and as such, we should not expect it to have an immediate impact on the existing or ongoing work in complexity theory. Likewise, the analysis of the following chapters is not intended to be an indictment of theoretical computer science, a theory that has produced real theorems and has led to very tangible results.

But even if we concede that complexity theory has a life of its own, the work we do here will have an impact on how we understand the broader issues addressed by

will be settled.

theoretical computer science. For instance, our focus on the relation between the formal and informal accounts of nondeterminism will be familiar to those who have worried about other dualities in computer science; questions about syntax and semantics, use and mention, and the differences between the concrete and the abstract each play on the fuzzy relation between our intuitions and the formal mechanisms we use to model them. More generally, we will begin to see how big-picture questions about the development of nondeterminism resonate with long-standing issues from the philosophy of mathematics, and how they ultimately shed light of the nature of computer science *qua* science.

Our immediate goals, however, are more modest. We will not presume to set the agenda for a comprehensive philosophy of computation, but we will take a good, albeit small, step in that direction by untangling the intuitions associated with nondeterminism. Our work is part detective story and part philosophy; the first explicit, theoretical reference to the nondeterministic Turing machine is our smoking gun and the work we do to find it will reveal otherwise unnoticed tensions.

Before we can get to our main questions concerning the nondeterministic Turing machine, we will need to understand the deterministic model. Hence, in Chapter 2 we will devote ourselves to the review of some canonical results, beginning with Turing's solution to the *Entscheidungsproblem* and concluding with some of the arguments that initially convinced complexity theorists that the deterministic Turing machine was sufficiently robust model to support new theory. Although the proofs are all familiar, our efforts here are not perfunctory. Rather, by revisiting familiar results we remind ourselves of the theorist's original motivation to view Turing machines as algorithms and

we will bring the problems of Chapter 3 into sharper relief.

Next we turn to one of the main objectives of this essay, that of challenging the presumption of conceptual continuity in the development of nondeterminism. In Chapter 3 we will see just how far removed intuitions about the mathematical utility of nondeterminism are from intuitions about nondeterminism as a evidence for the Church Turing thesis. We will begin with a standard presentation of the nondeterministic Turing machine along with the usual proof of the equivalence between (unbounded) deterministic and nondeterministic machines. The equivalence proof is important because it suggests that, despite first impressions, nondeterministic machines do indeed have a place in a theory of effective procedures. But when we turn our attention to resource-bound computation we will see that we can no longer take this equivalence for granted. In fact, I will present an open problem from complexity theory where the received understanding of nondeterminism has strikingly counter-intuitive consequences.⁷ At first blush, the problem seems trivial—understanding why it is not reveals the tensions between the mathematical intuitions about nondeterminism and our common intuitions about algorithms. At the same time, however, we should not dismiss our non-solution as naïve. Quite to the contrary, the non-solution follows from deeply held intuitions about composing algorithms. Should we embrace the nondeterministic Turing machine as yet another equivalent model of computation or should we emphasize a counter intuitive distinction between deterministic and nondeterministic algorithms in order to support a

⁷ In particular, we will look at the NP versus coNP question which asks whether the class of problems solved by a nondeterministic Turing machine (i.e., those problems whose solutions are easily verified) is coextensive with its complement class (i.e., those

non-trivial theory of complexity? Philosophically speaking, we cannot have it both ways. Moreover, we will see that insofar as there is a received history of computation theory, it does not help us here, for it fails to pin down the explicit introduction of the nondeterministic Turing machine, thus leaving us without a broader context to sort out these conflicting intuitions. Making sense of nondeterministic algorithms turns out to be very difficult.

It will be clear by the end of Chapter 3 that we face a dilemma: We can either ignore resource bounds when we articulate our notion of computable or we can pursue a rich theory of complexity, which happens to be premised on a model of computation that has nothing to do with our pre-theoretic intuitions. In Chapter 4 we will push this dilemma deeper still. We will argue that the predicament we face is not just a philosopher's problem, but rather the result of conflicting theoretical motivations that have yet to be acknowledged. We will argue that received history does not adequately account for the origin of the nondeterministic Turing machine. When we look more carefully, we will find that nondeterminism was first discussed by Turing in his seminal 1936 work. The discovery is surprising given that the subsequent development in recursion theory focused so completely on deterministic computation; so much so that one might conclude that the idea of nondeterministic computation had never even been considered. What is more surprising is that Turing appealed to nondeterminism to argue for the robustness of his notion of *computable*. We will see that the intuitions were altogether different when nondeterminism was re-introduced as a conservative expedient

problems whose solutions are not easily verified).

in automata theory some twenty years later. The intuitions change once again when nondeterminism turns out to be a non-conservative assumption in formal language theory before finally becoming the source of so much headache in complexity theory in the late 1960s and early 1970s. It is no wonder that the nondeterministic Turing machine is the source of so much philosophical confusion given that it has been put to so many disparate uses.

In the end, we will not find a philosophical solution to this dilemma. We will argue that the nondeterministic Turing machine, *qua* algorithm, makes no sense. But at the same time, we cannot simply dismiss the nondeterministic Turing machine out of hand given how much theory rides on the idea. Not only are there the famously (some might say notoriously) open questions from complexity theory, there are also inter-theoretic reductions among logic, model theory and complexity theory. So, for better or for worse, we are stuck with conflicting intuitions about nondeterminism and a philosophically inscrutable model of computation. Still, in Chapter 5 we will see that the work we have done raking the muck will illuminate some interesting philosophical questions. For instance, by the time we are done it should be clear that one of the main tasks for a more thorough-going philosophy of computation will be explaining the shift from a theory motivated by intuitions about machines and what it means to compute, to a theory that is now focused on questions about patently unrealistic models of computation. While some might see a natural evolution that requires no explanation, I hope that Chapters 3 and 4 will have shaken any initial confidence in a presumption of conceptual or historical continuity. A comprehensive philosophy of computation must also include

the intellectual history of the Church Turing thesis. Although Davis (1965; 1982; 1988a; 1988b) and Webb (1980) have started that investigation, our discussion of Turing's nondeterministic machines complements their efforts. Finally, in Chapter 5 we will find ourselves in position to evaluate the claim that computer science is a new science. We will see interesting implications not only for the philosopher of science, but also for the philosopher of mathematics.

II. Some Historical and Technical Background

0. Overview

Before we go into any more detail about the philosophical problems surrounding the nondeterministic Turing machine, we will discuss in this chapter the original motivations for the Turing machine. We will also review some of the standard results that first convinced mathematicians that the Turing machine was an appropriate model of computation and some of those results that later convinced complexity theorists that the Turing machine was an appropriate model of resource-bounded computation. We will also present the definition of a formal grammar (a notion we will discuss in Chapters 3 and 4).

1. The Original Motivations for the Turing machine

It is well known that Hilbert once considered the problem of deciding the validity of an arbitrary formula of the first order predicate calculus to be one of "fundamental importance" to mathematical logic (Hilbert and Ackermann 1950, p. 112). The so-called *Entscheidungsproblem* came to occupy center stage not only as a question about logic *per se* but also because Hilbert had seen how substantive mathematical questions could be reduced to questions about the validity of particular first-order sentences. Moreover, the problem demanded an *algorithmic* solution. There was nothing to gain in reducing a mathematical question to a decision problem that required inspiration to solve; but there was real potential in the possibility that validity might be decidable by the crudest of methods—methods that could be followed like recipes and applied mechanically, without

insight. Of course, at the time the decision problem was posed, the formal sense of algorithm had yet to be worked out mathematically. As Davis (1988b) points out, without a formal notion of algorithm, a negative solution to the *Entscheidungsproblem* would be doubly hard; while a positive solution might exploit existing intuitions about algorithms, it was not enough to establish that the decision problem was unsolvable by this or that mechanical procedure, but rather that the problem was unsolvable by *any possible* mechanical procedure. Anything short of this would leave open the possibility that a richer sense of algorithm might yield a solution to the *Entscheidungsproblem*.¹

In 1936, Turing proved the decision problem to be algorithmically unsolvable.² The proof itself is a short, straightforward *reductio*: Turing observes that a positive solution to the *Entscheidungsproblem* would entail a positive solution to a problem that, on pain of contradiction, has no solution; hence the *Entscheidungsproblem* must be unsolvable. Of course, before Turing can reach this conclusion, he must argue for the existence of an unsolvable problem and to do this Turing must pin down the sense of algorithm. Thus, the Turing machine enters the picture.

Turing asks us to compare the actions of working mathematician to those of a

¹ Hilbert used just such an argument in 1938 to respond to Church's notion of computable and his proof of the insolubility of the *Entscheidungsproblem*, "Church's work proves, however, the non-existence of such a recursive procedure: *at least*, the necessary recursion would not fall under the general type of recursion set up by Church, who has given ... *a certain* precise formalization" (Hilbert and Ackermann 1950, p.124, emphasis added).

² Church actually solved the problem before Turing did, but it is has been well documented that the two worked independently (see Davis 1988b, pp. 159-161). Moreover, as we will see in the coming chapters, Turing's work was more influential than Church's. And so we focus on the Turing machine.

machine. The idea is that just as human computation is a step-by-step process carried out symbolically on paper and pencil, so too we might imagine a machine that can scan and print symbols while working in a regimented, stepwise manner. Moreover, just as the human computer is limited—he can only keep track of so many symbols at a time and he can remember only so much—we imagine that the Turing machine is likewise limited in its capacity. Thus the machine works with a finite number of symbols and it depends on a finite number of internal "states" in order to "remember" what it is doing. With these assumptions in place, Turing presents his model of computation. We have a "machine" (the computer) with a segmented "tape" (the paper) running through it. Each segment of the tape can contain a single symbol and the machine is "directly aware" of only one symbol at a time. Turing defines the *configuration* of a machine at a given time as given by the internal state of the machine together with the symbol it is scanning. Turing also contends that in order to compute, a machine need only be capable of a handful of very rudimentary behaviors. In response to a given configuration a machine might erase the scanned symbol, it might write a new symbol into the square, it might shift the tape one segment to the left or to the right, or it might change its internal state.

The model is austere, but more important, it is finite.³ That is, there is a finite number of internal states, a finite number of symbols and a finite (and small) number of

³ It is interesting to note that Turing imagined his machines working out computable sequences by printing out an infinite number of terms in the sequence. Turing's machines never stopped working. These days we prefer to think of working machines as those that always halt. The difference is not as great as it seems, however, for even if Turing's computations are of an indefinite length, his computing machines are still described by "finite means."

possible behaviors. Thus it is possible to describe the actions of a given machine, and hence the machine itself, in any number of ways. Turing chooses to describe his machines in terms of tables with columns to represent configurations and behaviors while each row represents a particular pair of state and symbol together with the machine's response to the current configuration (i.e., erase, print, move or change state).

Consider Turing's simple example:⁴

<u>configuration</u>	<u>behavior</u>
q ₁ , blank	"1", q ₂ , right
q ₂ , blank	"0", q ₁ , right

This table describes a machine with just two states that, when started in state q₁ reading a blank square, will print a "1," change its internal state to q₂ and move one square to the right. In state q₂ reading a blank the machine prints a "0" returns to state q₁ and moves one square to the right at which point it is again scanning a blank square in state q₁. This machine prints the string "10101010 ..." *ad infinitum*.

Of course, more complicated machines will have more complicated tables, but what is most important is that the tables themselves, no matter how complicated, can be transposed into a standard format. In fact, rather than using a table with rows and columns, we can concatenate the information in each row and present the table as a string of characters. In our example we can write something like:

$$q_1 S_0 S_1 q_2 R : q_2 S_0 S_0 q_1 R :$$

⁴ Our presentation is somewhat anachronistic in that the machine we have described prints its output on consecutive squares. Turing originally imagined his machines

where S_0 , S_1 , R stand for "blank," "1" and "Right," respectively. Finally, if we denote " q_1 " by the numeral "10" and " q_2 " by the numeral "100" (in general q_i will be denoted by a "1" followed by i "0"s), " S_0 " by "20", " S_1 " by "200", " R " by "3" and ":" by 4 we can present our machine as a numeral, namely:

10202001003410020201034

Turing calls such numerals "description numbers." The idea is that every machine will have at least one description number, while each description number will pick out a unique machine. The standardization and arithmetization of machine tables leads to several remarkable facts: the computable sequences turn out to be enumerable; machine descriptions can be presented as input to other machines; and there exists a "universal" machine that can simulate the behavior of any possible Turing machine.⁵

Now Turing can argue for the existence of an unsolvable problem. He begins by introducing the notions of *circular* and *circle-free* machine: A machine is *circular* if it

printing on alternate squares and using the intervening squares for "scratch work."

⁵ The first two facts are immediate consequences of Turing's arithmetization (the cardinality results implicit in the first fact also foreshadows the diagonalization Turing will use to establish the existence of undecidable problems). The third fact, however, is quite surprising. Indeed, it is far from obvious that one machine could do the work of *any* other machine. But as Davis (1988b, p.159) points out, the universal machine is another immediate consequence of Turing's analysis. Machine tables are really just lists of instructions and according to those instructions each machine will execute a different algorithm. The universal machine, on the other hand, executes a very simple algorithm, namely, follow those instructions encoded by this description number. Except for the decoding (a trivial step), Turing describes such a machine in detail. According to Davis, the "apparent implausibility" of such a universal machine together with the fact that Turing was actually able to describe it provides a "significant vindication" of Turing's analysis of computation. Davis also suggests that Turing's universal machine anticipates both the stored program computer and the notion of an *interpretive* program, and that it anticipates the blurred boundaries between hardware and software and program and data.

never prints more than a finite number of symbols (the idea is that a machine that prints only a finite number of symbols must eventually repeat some configuration of state, symbol and scanned-square). Otherwise, the machine is *circle-free*. Next, Turing uses a diagonal argument to show that there cannot be an algorithmic process for determining whether a given machine is circle-free, for if there were such a process we could compute the sequence $\beta' = \phi_n(n)$ (where $\phi_n(n)$ is the n^{th} digit in the n^{th} sequence computed by a circle-free machine) which leads to a subtle contradiction: Suppose there were a machine D to decide circularity. To compute β' for some number n , we construct a composite machine H that feeds successive description numbers to D (starting with 0) until n circle-free machines have been identified. At this point the n^{th} machine is simulated (via the universal machine, another component of H) until it prints its n^{th} figure—which it is guaranteed to do since it is a circle-free machine—and H outputs that figure as $\phi_n(n)$. H is circle-free by construction (since each component machine is circle-free) and it has some description number k . So H must compute $\phi_k(k)$. It is obvious that H can simulate the other $k-1$ machines as it prints the first $k-1$ digits of β' , but how does H simulate itself when it is time to print the k^{th} digit? As Turing notes, there is no explicit instruction for H to compute the k^{th} digit, "but the instructions for calculating the [k^{th} digit] would amount to 'calculate the first [k] figures computed by H and write down the [k^{th}]" (Turing 1965a, p.133). There is, so to speak, a regress in simulation. $\phi_k(k)$ is "never found," and contrary to the assumptions underwriting our construction, H is circular. Thus Turing concludes that there can be no algorithmic process for deciding circularity.

With this undecidable problem in hand, Turing presents another problem and

argues that a solution to it would entail a solution to the problem of deciding circularity. Although Turing correctly identifies the problem of deciding whether an arbitrary machine will print a given symbol as undecidable, his proof is confusing.⁶ Nevertheless, the undecidability of this printing problem allows Turing to prove, finally, that the *Entscheidungsproblem* is unsolvable. As we indicated above, Turing argues that a positive solution to the *Entscheidungsproblem* would entail a positive solution to the printing problem. The details of the proof are tedious, but again straightforward (we will see essentially the same construction again in Chapter 4). The underlying idea is that the operation of a Turing machine on a given input can be described logically. In particular, it is possible to construct (effectively) first-order formulae that codify the machine's behavior and the contents of its tape. Turing begins by introducing a variety of predicates to represent, for example, the individual cell being scanned at a particular point in the computation, the contents of that cell, the configuration of the machine at a particular point in the computation, and a handful of conditional statements to represent the

⁶ Following (Post 1965) Davis notes, "the argument is a bit complicated" (Davis 1988b, p. 136). Although Turing seems to be using a reduction, Davis points out that the problem of deciding circularity is of a higher degree of unsolvability than Turing's problem of deciding whether an arbitrary machine will print a given symbol and hence circularity cannot reduce to printing. A simple diagonalization yields an easier (and correct) proof of the latter problem's undecidability: Following Davis (1988b), suppose there were an algorithm to determine whether a given machine prints, say, a "|". Then we can construct another machine, *M*, which will take description numbers as input and print a "|" iff the described machine does not print a "|". Now what happens when *M* takes its own description number as input? *M* prints a "|" if and only if it does *not* print a "|"—a contradiction. Hence, there can be no algorithmic procedure to decide whether an arbitrary machine will print a given symbol.

machine's instructions.⁷ Turing uses these formulae to construct another existentially quantified formula which asserts that "in some complete configuration of M , S_1 (i.e., 0) appears on the tape" (Turing 1965a, p.146). It then follows by an inductive argument that the formula is provable if and only if the machine M does, in fact, print a zero. So, if we could decide derivability in a formal system (i.e., if we could decide the validity of a conditional statement relating the axioms of a formal system with the theorem in question) then we could decide the printing problem, which we cannot do, so we cannot solve the *Entscheidungsproblem*.

2. The Turing Machine as an Unbounded Model of Computation

The immediate effects of Turing's work were clear: Hilbert could not hope to answer every mathematical question and these undecidability results together with the Gödel incompleteness theorems severely undermined the formalist program in mathematics. But more important, Turing's work was one of the first attempts to give precise mathematical content to notions like *computable*, *algorithmic* and *effective* which before had been only vague, intuitive notions. Consequently, attention was focused on a host of arguments for the adequacy of the Turing machine as a formal model of computation. As we will see in Chapter 4, these arguments were as much philosophical

⁷ For each instruction Turing introduces a conditional statement universally quantified over *complete* configurations (i.e. tape contents + machine configuration) and tape positions. Intuitively, the antecedent represents the state of affairs before the instruction is executed (e.g., scanning symbol s in cell x , in state q) while the consequent reflects the updated tape/machine configuration after the instruction is carried out (e.g., scanning symbol s' in cell $x-1$ in state q'). We should also note that Turing's original formulation of these conditionals was a bit loose—so we will pass over the particulars of Turing's presentation.

as mathematical. In the meantime, however, we will look at some of the mathematical arguments.

As we have noted, Church was the first to solve the *Entscheidungsproblem*. Turing learned of Church's work as he was completing his own and he felt compelled to add an appendix outlining the equivalence between Turing-machine computability and the notion of effective calculability implicit in Church's system of λ -definable functions. Quite apart from the intuitive appeal of Turing machine, we will see in Chapter 4 that it was crucial that Turing's work did not fall short of extant accounts of computation. The fact that Turing-machine computability was provably equivalent to Church's sense of "effective calculability" was, perhaps, the most compelling mathematical support for the Turing machine. Indeed, similar results led Church to comment in 1935: "[The fact] that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability [i.e., the λ -definable and general recursive functions] turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it" (Church 1965, p. 90). Adding another equivalence result not only bolsters our faith that we have identified a general characterization of computable, but also has the reciprocal effect of legitimizing the Turing machine as a model of computation.

There were also more direct mathematical justifications for the model itself. For instance, there is no loss in generality in restricting Turing machines to a linear tape even though human computers use two-dimensional work spaces. Kleene outlines the proof:

We imagine a two-dimensional work space "sufficiently regular in structure so that the [human] computer will not become lost in it during computation" (1952, pp. 380-381). So, we might think of a grid-paper where from each cell in the grid there will be only finitely many directions (e.g., up, down, right, left, diagonally etc.) in which we can move to different cells in the grid. Since the number of different motions is finite, the number of cells reachable from a given cell is countable. Next, Kleene argues that for "any readily imagined symbol space" it is possible to enumerate all the cells in such a way that for each direction of movement there is a computable function that enumerates the cells reachable from a given cell in that direction. This function is then used to index two-dimensional grid-positions into a linear tape.⁸ In a similar vein, well known conversions between numeral systems demonstrate that the restriction to monadic (in the unbounded case), or dyadic notations does not compromise the ability of the Turing machine with respect to a human computer who might use a richer set of symbols. Finally, there were competing versions of the Turing machine (cf. Post 1965) where the notion of an atomic act differed slightly from Turing's; thus it is reasonable to assume that the machine behaves discretely, in a step-by-step manner, even if the "atomic" acts are actually composite.

In addition to the foregoing mathematical considerations, there were also surprising methodological affinities between the approach taken at Princeton and the one developed independently by Turing that would have augured well for the Turing machine

⁸ The same idea is implicit in the programmer's view of a two-dimensional (or n -dimensional) array as a one-dimensional array of arrays (or of an array of arrays of arrays

as an adequate model of computation. For instance, Kleene reports that immediately after Church proposed the identification of the effectively calculable with the λ -definable functions, he "sat down to disprove it [i.e., the identification] by diagonalizing out of the class of λ -definable functions. But quickly realizing that the diagonalization cannot be done effectively, [he] became an overnight supporter" (Kleene 1981, p. 59). Although Turing was probably not aware of Kleene's overnight conversion, Turing was likewise concerned that an obvious diagonalization out of the computable sequences might undercut his work. But Turing concluded, like Kleene, that such a diagonalization could not be performed effectively. Not only does the "correct" application of the diagonal process lead to Turing's proof for the existence of an undecidable problem, but also his concerns must have resonated with those already working out the mathematical sense of effective calculability at Princeton. Turing's work would have seemed familiar even if the approach he actually took was quite different from the one taken on the other side of the Atlantic.⁹ Moreover, when he made it to this side of the Atlantic, the work Turing did at Princeton extended existing work quite naturally. For example, before Turing arrived in the summer of 1936, Church and Kleene had already given equivalence proofs identifying the class of λ -definable functions with the class of general recursive functions

of arrays. . .).

⁹ I do not mean to overstate the case here, but I imagine there was a substantial difference in mathematical temperament between Turing (before he went to Princeton in the summer of '36) and those working with Church. Turing was notoriously sloppy in his work, while people who knew Church have described him to me as exceedingly precise. Nevertheless, it is easy to imagine that Turing's work was favorably received by Church and others because they could recognize that Turing was doing things the right way even if his proofs were riddled with mistakes.

proposed by Herbrand and Gödel. Kleene (1981) recalls how this work on λ -definability had led him to his famous 1936 normal form theorem for general recursive functions: He thought of the computation of general recursive functions in terms of "stages" where each stage could be represented by a Gödel number, with the whole process admitting a primitive-recursive characterization. By applying a least-number operator it is possible to recognize, primitive-recursively, a "terminal" stage in the computation at which point the value of the function can be "extracted" primitive-recursively from the Gödel number. Thus we establish the normal form theorem that each general recursive function is obtainable from primitive recursive functions with (exactly) one application of the Kleene minimization operator (Kleene 1981, p. 60). Kleene's approach must have made an impression on Turing, for he cites Kleene's work and adapts it quite naturally to his 1937 demonstration that every Turing-computable function is general recursive. In fact, Turing proceeds exactly as Kleene had: he describes an arithmetization of complete configurations (the "stages" in Turing-computation) and proves that the operation of the machine can be described primitive-recursively. Then Turing uses Kleene's minimization operator to identify the point (the "terminal" stage) at which the output can be read from the tape (the "extraction").

The early equivalence results relating the general recursive functions, the λ -definable functions and the Turing-computable sequences are, perhaps, the most compelling mathematical evidence for the Church-Turing thesis, and by implication, evidence that the Turing machine itself is an adequate model of computation. But it is likewise compelling that, working independently, Turing confronted the same kinds of

problems as those working in the States and that he was able to adapt extant work so directly once he became aware of it. Even if Turing's work was something of a "rediscovery" it is still remarkable that it could piggyback so easily on the mathematical evidence that was already supporting Church's work.

3. The Turing Machine and Resource-Bounded Computation

As we will see in Chapter 4, by the time the Turing machine had become entrenched as the *de facto* model of computation, it was becoming clear that if the theory of computation was to have any connection to real computing then it would have to take resource-bounds into account. Once again, the Turing machine proved to be a compelling model; the transition from one complete configuration to the next was a natural analogue for a "unit of time" while the machine's tape cells were readily viewed as "units of space." Moreover, the Turing machine was robust in a resource-bounded sense, both in terms of modifications of the model and with respect to other models of computation. For example, in their classic 1965 paper, Hartmanis and Stearns initiate the theory of computational complexity using the *multi-tape* Turing machine as their model of computation because "it closely resembles the operation of a present day computer" (Hartmanis and Stearns 1965, p. 287). Indeed, with respect to the manner in which information is retrieved, a Turing machine that accesses several different tapes simultaneously is much more like a modern computer than a single-tape Turing machine that must access information in a strictly sequential manner. More important, such multi-tape machines are no more powerful than single-tape machines in unbounded contexts, but as Hartmanis and Stearns point out, the extra tapes do make a difference when we

worry about how long the computation takes (multi-tape machines are faster). Still, even in a time-bounded setting, it turns out that it makes no theoretical difference whether we think in terms of a Turing machine working with multiple tapes or whether we stick to the model of a single-tape machine. Hartmanis and Stearns prove that a single-tape machine can simulate a multi-tape machine with at worst a quadratic increase in the time of computation. The idea is to store the n symbols currently scanned on each of the n tapes of a given multi-tape Turing machine, MT , in a contiguous block of n cells (called it the 0 block) on the single tape of a machine ST . The symbols immediately to the right (or left) of each of the n scanned symbols of MT are stored in another contiguous block to the right (or left) of the 0 -block on the ST 's tape, and so on. With MT 's transition table "hard-coded" into the control, ST does the following for each of MT 's moves: First, ST scans the symbols in the 0 block to determine MT 's next state and what MT will do to each tape (e.g., print, erase, shift left or right). Next, ST prints the appropriate symbols in the 0 -block; then starting from the leftmost printed square, ST sweeps across its tape from left to right and for each tape MT shifts to the right, ST shifts the corresponding symbols from each block into the next block to the right. Upon reaching the rightmost square, ST sweeps back across its tape, repeating the process for each tape MT shifts to the left. Finally, ST returns to the 0 block and starts simulating MT 's next move. The number of moves ST makes is linearly proportional to the length of the tape it must traverse which itself grows in linear proportion to the number of moves MT makes. So if MT makes t moves, each of which requires roughly t moves for ST to simulate, then ST makes, at

most, t^2 moves.¹⁰

In a similar vein, Hartmanis and Stearns show that a Turing machine with access to a 2-dimensional tape can be simulated by a Turing machine with a linear tape with only a quadratic loss of efficiency. This result is the time-bounded analogue of Kleene's proof and it is important for similar reasons. Just as Kleene's proof suggests that the Turing machine is a sufficiently general model of computation, Hartmanis and Stearns' so-called square law suggests that the Turing machine is sufficiently robust with respect to time-bounds. As Papadimitriou puts it, these results give us reason to believe that "there is no conceivable 'realistic' improvement on the Turing machine that will increase the domain of languages such machines decide, or will affect their speed more than polynomially" (Papadimitriou 1994, p. 31).

Finally, much like the equivalence results of the '30s, there are time-bounded equivalences relating the Turing machine to different models of computation. For example, in 1963 Shepherdson and Sturgis introduced the *unlimited register machine* as another idealized model of computation. A register machine consists of a *program* and

¹⁰ N.b., complexity theorists are not interested in constant terms and linear factors. The constants are discounted because complexity theorists are worried about *rates* of growth (i.e., the proportional contribution a constant value makes to the time it takes to compute decreases as the length of computation time increases), while the linear factors are ignored on the basis of a "linear speed up theorem" which trades on the fact that computation time can always be improved at the cost of a more complex tape alphabet, or a greater number of internal states.

Also, we should note that as far as the complexity theorist is concerned, a quadratic loss of efficiency is an acceptable cost. In fact, as we will see in Chapter 4, polynomial complexity (of any reasonably low degree) has been equated with tractability. Exponential growth rates, on the other hand, make problems hard. As any undergraduate logic student knows, it is easy to determine whether a propositional formula of 3 atoms is

denumerably many *registers* each of which can store a natural number (only finitely many registered are used during the execution of a program). A program is sequential list of instructions which take register indexes as arguments and perform basic operations on register contents. Shepherdson and Sturgis introduce only six kinds of instruction: there are instructions for incrementing, decrementing and clearing registers, along with an instruction for copying the contents of one register into another, and two jump instructions to permit non-sequential flow of control in the program. (One jump is conditional depending on whether or not a given register is empty, the other jump is unconditional).

It turns out that the functions computable by register machines are exactly the functions that can be computed by a Turing machine. For Shepherdson and Sturgis this equivalence permits a more perspicuous proof of the equivalence between the partial recursive functions and the Turing-computable functions. At the same time, however, they recognize the register machine as a significant step in "the 'rapprochement' between the practical and theoretical aspects of computation" (Shepherdson and Sturgis 1963, p. 218). In fact, subsequent treatments of the register machine often emphasize the affinities to actual computers; most writers are quick to point out the obvious resemblance between the register machine "architecture" and RAM memory, and some even go so far as to talk about "program counters" and designate specific registers as "accumulators." Likewise, the instruction sets have been augmented to the point that they resemble the assembly-level instructions of existing computers; unlike Shepherdson and Sturgis' spartan set,

satisfiable; it is practically impossible to do the same for a formula with 6 atoms.

writers now include higher-level arithmetic instructions (e.g., add, subtract, divide), indirect register addressing and more complex jump instructions.

The net result is another equivalence relating a seemingly more concrete model of computation to the Turing machine. Most see this equivalence as further evidence for the Church Turing thesis. Moreover, it turns out that a Turing machine can simulate a register machine with only a polynomial loss of efficiency. The trick here is to use a contiguous block of tape cells containing $\langle \text{index:content} \rangle$ pairs to simulate the more flexible memory of the register machine (along with routines for shifting and copying as the contents of each register grow and shrink). The register machine program can then be "hard-coded" into the Turing machine control (assigning a group of internal states to implement each basic register machine instruction as a Turing machine subroutine). As before in the single-tape simulation of the multi-tape machine, the Turing machine simulation of each register machine instruction might require several steps, but in general, the time it takes to simulate each step the register machine takes will be linearly proportional to the amount of tape needed to store the register contents (which will never grow more than linearly for each register machine instruction executed). Hence, if a register program executes t instructions (each instruction requiring one unit of time to execute) then a Turing machine will require roughly t^2 for its simulation.

The fact that a single-tape Turing machine can simulate seemingly more powerful machines together with the fact that it can simulate altogether different models of computation is compelling evidence for the Church Turing thesis. Likewise, the fact that these simulations result in only a polynomial loss of efficiency is compelling evidence

that the Turing machine is also the right model of resource-bounded¹¹ computation. The similarity here is not accidental: A robust model of resource-bounded computation makes for a more robust model of computation generally. The nagging exception is nondeterminism. As we will see in the next chapter, although nondeterminism adds no power to the unbounded Turing machine (more evidence for the Church Turing thesis), it seems that deterministic simulation of a nondeterministic machine results in an exponential loss of efficiency (perhaps evidence against the Church Turing thesis?).

4. Formal Grammars

The last thing to do before we try to tackle the more difficult questions about nondeterminism is to introduce the definition of a formal grammar. The notion is due to Chomsky (1959) and, as we will see in Chapter 4, it is at the heart of another important collection of equivalences. Intuitively, a grammar is a set of rules we apply to words, noun-phrases, verbs-phrases, etc. in order to form (or identify) grammatically correct sentences. Likewise, a formal grammar is a 4-tuple (V, T, P, S) where V and T are disjoint finite sets of *variables* (denoting syntactic categories) and *terminal symbols* (words), P is a finite set of *productions* (rules) and S is a specially designated element of V (the "start" symbol). Productions are of the form $A \rightarrow \alpha$ where A is a variable and α is a concatenation of elements from $V \cup T$. For example, we might think of a simple language with only two "words," a and b and two syntactic categories A (for a-phrases) and B (for b-phrases) where the only grammatically correct sentence are those that begin

¹¹ Even if all our discussion has focused on time-bounded computation, we speak more generally of resource-bounded computation since a time bound implies an equal space

with at least one a and end with twice as many b 's. Thus we might define the corresponding grammar G as:

$$\langle \{A, B\}, \{a, b\}, \{S \rightarrow AB, S \rightarrow aSbb, A \rightarrow a, B \rightarrow bb\}, S \rangle.$$

So, if we think of productions as derivation rules, we can derive the sentence "abb" by starting with S , applying the production that yields "AB," which, in turn, yields "aB" by the third production, before we finally derive "abb" by the fourth production. Similarly, to derive "aabbbb" we would start with S and apply the second production to get "aSbb" at which point we would follow the steps of the last example to arrive at "aabbbb." It should be clear, at least informally, that G generates all and only the sentences of our simple language. Taking " \Rightarrow_G " to be the reflexive and transitive closure of " \rightarrow " from our production rules, we say that the languages generated by G is all the sentences w such that $S \Rightarrow_G w$.

Although the notion of a formal grammar was introduced long after Turing machine had been accepted as an adequate model of computation, it still had a profound influence on the development of theoretical computer science. In particular, a handful of machine-grammar equivalences were explored in the late fifties and early sixties. It turned out that by placing certain restrictions on the productions of a grammar (e.g. that each production be of the form $A \rightarrow tB$ or $A \rightarrow B$ where t is a terminal symbol), it was possible to relate the generative capacity of various grammars with the power of various abstract machine models to recognize the sentences thus generated. As we will see, questions about such equivalences led to surprising questions about nondeterminism.

bound (but not conversely).

III. Philosophical Concerns about Nondeterministic Algorithms

0. Overview

In Chapter 1 we noted that there are at least four different ways to think about nondeterminism. We suggested that this variety of intuitions betrays a host of philosophical tensions, but we also recognized that a long tradition in theoretical computer science has implicitly established a presumption of continuity in both the historical and conceptual development of nondeterminism. In this chapter we will examine an open question from complexity theory where the received theoretical understanding runs counter to some very natural intuitions about algorithms. We will point to a specific intuition about the workings of a nondeterministic machine as the source of these tensions and then sketch a brief history of nondeterminism in theoretical computer science. Finally, we will see that more recent theoretical developments are unlikely to resolve these philosophical tensions.

1. Introduction

This chapter is motivated by a single footnote in Rogers' classic text, *Theory of Recursive Functions and Effective Computability*. The footnote comes on the second page where Rogers discusses discrete stepwise computation and deterministic operation as two of five "essential" features of the informal notion of algorithm. He notes, "In a more careful discussion, a philosopher of science might contend that *4 [i.e., discrete stepwise computation] implies *5 [i.e., determinism]. Indeed, he might question whether there is any real difference between *4 and *5" (Rogers 1967, p. 2).

When we engage in that "more careful discussion" we must ask ourselves why Rogers (or anyone for that matter) would regard the implication between discrete stepwise computation and deterministic computation as philosophically obvious when even a cursory glance at a text in complexity theory seems to provide a counter example: the nondeterministic Turing machine, like the deterministic machine it generalizes, is presented as a discrete stepwise model of computation and yet it is clearly not deterministic. Should we conclude that discrete stepwise computation has nothing to do with determinism, or that nondeterminism has nothing to do with Rogers' informal characterization of algorithm?

Neither option is attractive. Rogers motivates his discussion of algorithms with an analogy to digital computers. He claims that discrete stepwise operation corresponds to the "digital nature" of real computers, and that the sense of determinism—that is, not having to "resort to random methods or deices, e.g. dice"—is reflected in the "mechanistic nature" of digital computers (Rogers 1967, pp. 2-3). The analogy is compelling. Indeed, the paradigm of digital computing completely depends on an "edge-driven" synchronous architecture and the ability to guarantee state transitions with absolute certainty. It is hard to deny Rogers' claim that *4 implies *5 when intuitions about discrete stepwise operation, determinism and the notion of an algorithm are tied together in the context of real computing.

At the same time, however, we cannot baldly assert that the study of nondeterminism has nothing to do with the study of algorithms. Quite to the contrary, the notion of a nondeterministic algorithm is ubiquitous in complexity theory. The $P=NP$

question—the most celebrated open question of theoretical computer science—is a question about the difference (or lack thereof) between deterministic and nondeterministic algorithms. Moreover, nondeterministic algorithms really are presented as algorithms. For example, consider how Bovet and Crescenzi (1994, p. 70) describe a nondeterministic algorithm to decide whether a Boolean formula, f , consisting of n atoms is a member of SATISFIABILITY, the language of all satisfiable Boolean formulae:

A nondeterministic algorithm for SATISFIABILITY can be obtained by guessing any of the 2^n assignments of values to the n variables of f and verifying whether it satisfies f :

```
begin {input:  $f$ }  
  guess  $t$  in set of assignments of values to the  $n$  variables of  $f$ ;  
  if  $t$  satisfies  $f$  then accept else reject;  
end.
```

There are no scare-quotes or disclaimers in the foregoing passage. The use of the Pascal-like language is deliberate and is intended "to provide a more succinct description of *algorithms*" (Bovet and Crescenzi 1994, p. 47, emphasis added). Such descriptions of nondeterministic algorithms appear side-by-side with, and are embedded in, deterministic algorithms specified in the same language. Except for the "guess" operator, there is no difference in the presentation of deterministic and nondeterministic algorithms.

There is a dilemma here with no easy solution. Although Rogers excludes those methods that resort to "random devices" from his informal characterization of an algorithm, the "guessing" of a nondeterministic algorithm is not random, hence, the sense of nondeterminism we might infer from Rogers' discussion does not help us. Our dilemma is rooted in a much deeper and more subtle tangle of intuitions. We will see

below that the nondeterministic Turing machine has been presented as further vindication of the Church Turing thesis and that intuitions stemming from the evidence of extensional equivalence naturally resonate with intuitions about what it is to be an algorithm. At the same time, however, many of the open problems in complexity theory trade on intuitions about nondeterminism as mathematically useful description of those problems. We will argue that understanding why one such problem remains open demands that we give up our intuitions of what it means to be algorithmic. We will also see that the tension between these intuitions has been obscured by a third intuition that supports a view of complexity theory as a study of those questions that affect real algorithms and real computers.

2. Getting to the Center of the Tangle

In this section we will do our best to present the nondeterministic Turing machine in the light of the *received theoretical view*. In other words, we will try to present the nondeterministic Turing machine as it might be presented in an introductory text book on the theory of computational complexity. Although the presentation in this section will be deliberately a-historical, it will help us uncover the conflicting theoretical intuitions about nondeterminism.

The study of computational complexity begins with the study of computation and the (deterministic) Turing machine is most often presented as an intuitively appealing model of computation (cf. Papadimitriou 1994, pp.19ff). Various equivalencies between the Turing machine and other models of computation are demonstrated and adduced as evidence for the Church Turing thesis: the claim that our intuitive notion of algorithm

can be identified with the Turing's model of computation (cf. Bovet and Crescenzi 1994, p. 25; Smith 1989, p. 299).

At this point we might ask whether extending our model of computation will affect the class of Turing computable functions. For example, in formal language theory Turing machines are used to decide membership in a language (i.e., used to compute a characteristic function for a language like SATISFIABILITY), and it is often helpful to relax the demand that the next move be uniquely determined by the current state and input. As we indicated in Chapter 1, we define a nondeterministic Turing machine exactly like a deterministic machine except that the current state and input does not uniquely determine what the machine will do next (i.e., what state it will assume, what action it will perform). It is hard to imagine how such a machine might work given that its behavior is determined by a transition relation rather than a transition function. Indeed, text books rarely described how such nondeterministic machines really work. Instead, there is a handful of related heuristics. Some writers invoke metaphors about the machine "guessing" what to do next (cf. Bovet and Crescenzi 1994; Smith 1989, pp.299-302), while others appeal to a sense of parallelism (cf. Bovet and Crescenzi 1994, p. 48; Papadimitriou 1994, p. 172). We will try to motivate nondeterminism in terms of so-called computation trees (cf. Bovet and Crescenzi 1994; Papadimitriou 1994). The common thread in each of these stories is that the machine *always* guesses correctly, *always* computes efficiently, *always* follows the accepting branch in a computation tree if one exists.

Let us consider nondeterminism in terms of a computation tree. A computation

tree is a directed graph where the nodes represent a particular configuration of the state, input and tape contents at a particular step in the computation of a nondeterministic Turing machine. A single node—the root—represents the initial configuration of the machine. Each (directed) edge represents a transition to one of the (finitely many) different configurations the nondeterministic machine might realize on the next step. The leaf nodes represent all the possible final configurations of the machine. On any given input the computation tree might have several distinct branches, some of which lead to acceptance and others of which do not. So, to avoid ambiguous results on a given input, a nondeterministic machine is said to accept an input if there is at least one "accepting branch" in the computation tree, otherwise the input is rejected. In this context we can see how intuitions about guessing and parallelism are related; to say that a nondeterministic machine guesses is to say that it makes exactly the right choices so that it assumes the sequence of configuration reflected on the accepting branch of the computation tree (if such a branch exists). Likewise, the intuitions about parallelism are rooted in the image of a nondeterministic machine somehow surveying all the branches in the computation tree at once, but manifesting only the behavior of the accepting series of configurations (if such a series exists). By contrast, the computation trees we might associate with deterministic machines are more like computation twigs; in a deterministic computation there is a single path from the initial configuration to the final configuration (either accepting or rejecting) without any branching whatsoever. There is no guessing, no implicit parallelism.

To return to the example of a nondeterministic Turing machine deciding the

language SATISFIABILITY, we can think of the each node in the computation tree representing the choice the machine must make between assigning a value of "true" or "false" to a particular atom. Thus, if the input formula has n atoms, the computation tree will have 2^n different branches. Whether the nondeterministic machine surveys all these branches in parallel, or whether it "guesses" a satisfying assignment, the machine accepts if and only if there is branch in the tree that corresponds to a satisfying assignment. By contrast, again, we can think of the deterministic computation twig as corresponding to a sequential enumeration of the 2^n different assignments with each assignment checked in turn to see if it satisfies the input formula. Although it must survey an exponential number of assignments, the deterministic Turing machine will eventually find a satisfying assignment if one exists.

It might seem that allowing the machine to "guess" would yield greater computational power but, as it happens, nothing changes; any language decidable on the extended model turns out to be decidable on the original model. In other words, given a nondeterministic Turing machine it is always possible to construct a deterministic machine that will "simulate" the nondeterministic machine. The proof exploits the fact that the behavior of a nondeterministic machine is determined by a *finite* transition relation. Suppose we have a nondeterministic machine to decide whether a string of input is a member of a given language. Given a description of the transition relation of the nondeterministic machine, a deterministic machine can, on a given input, systematically work its way through the branches of the nondeterministic computation tree using its tape to keep track of the choices the nondeterministic machine can make. By looking at all the

choices the nondeterministic machine can make after 1 step, 2 steps, 3 steps and so on, the deterministic machine will eventually survey all the choices that could lead the nondeterministic machine to accept. The deterministic machine will accept the input if and only if there is an accepting branch in the nondeterministic computation tree. Thus, the deterministic and nondeterministic machine compute the same characteristic function.

Here we see more evidence for the Church Turing thesis (see, e.g., (Bovet and Crescenzi 1994, p. 20) or (Hopcroft and Ullman 1979, pp. 159-166)). The fact that a nondeterministic machine is no more powerful than a deterministic machine with respect to the class of languages it can decide is compelling. Moreover, we find that the heuristics we have used to understand the behavior of a nondeterministic machine are inessential in this context. For instance, the equivalence proof we sketched above demonstrates that even if we think of a nondeterministic machine as guessing during its computation, we can eliminate those guesses in favor of an exhaustive search. We can see immediately that such a search can be conducted deterministically, and hence we can maintain our intuitions of what it is to be an algorithm even though the notion of a "guessing" algorithm seems to have no place in the discussion of what can be accomplished by machines.

But strange things happen when we introduce resource bounds and thereby shift our attention from nondeterminism as a vindication of the Church Turing thesis to nondeterminism as a useful device in the theory of computational complexity. The idea is to keep track of how much time (or space) it takes to compute. For instance, using the Turing machine as our model of computation, we can count the number of steps (or tape

cells) it takes to compute a given function. We are especially interested in those characteristic functions that can be computed in a polynomial amount of time where the polynomial is a function of the length of the input string whose membership in a language is being decided. When a characteristic function can be computed by a Turing machine within a polynomial time bound, we say that the corresponding language can be decided by a polynomial-time Turing machine.

As we indicated in Chapter 1, the P versus NP problem asks whether the P, the class of languages that can be decided by a deterministic polynomial-time Turing machine is the same as the NP, the class of languages that can be decided by a polynomial-time nondeterministic Turing machine. Since we have already established the equivalence between deterministic and nondeterministic Turing machines, it might seem that we could use the same simulation technique to prove that $P=NP$. But we cannot.

To see why we cannot we note that although the simulation demonstrates how all the branches in a nondeterministic computation tree can be surveyed systematically, there will be, in general, exponentially many branches to survey. Even if each individual branch can be surveyed in a polynomial amount of time, it will take an exponential amount of time to survey every branch in the nondeterministic computation tree. Although we can always use a deterministic machine to simulate nondeterministic computation we do so with an exponential loss of efficiency with respect to the time it takes to perform the computation.

Papadimitriou claims that the P versus NP problem is a matter of understanding "[w]hether this exponential loss is inherent or an artifact of our limited understanding of

nondeterminism" (1994, p. 45). At this point, however, there is nothing very mysterious about nondeterminism; we define nondeterministic machines in terms of transition relations rather than transition functions, we point to a well-known simulation technique and then find an exponential explosion in the time it takes to survey all the choices a nondeterministic machine can make. If we look at nondeterminism in this light, it is not surprising that the P versus NP problem would be described informally on the Claymath web-page as the question whether there exists a certain kind of problem that "really does require a long time to solve" or whether "we simply have not discovered how to solve them quickly." The intuitions about guessing, parallelism and bushy computation trees are all colorful ways of describing the fact that some problems seem to engender an exponential search-space. The heuristics we use to understand nondeterministic behavior are inessential in this context. In fact, NP might as well stand for nonpolynomial rather than nondeterministic. We say this even though Smith makes clear that, "'NP' stands for nondeterministic polynomial" not "nonpolynomial"(1989, p. 318). Presumably, Smith is making sure his readers understand what the abbreviations stand for in a theoretical context. But in a philosophical context, we might as well assume that "NP" abbreviates nonpolynomial since it is clear how we can transform a nondeterministic algorithm into a deterministic algorithm with an exponential loss of efficiency. Nothing about such a transformation challenges our intuitions about what it is to be algorithmic; it merely suggests that the algorithms for some problems might take a long time to execute.

Still, our glib reinterpretation of "NP" is reasonable only if we think that there is nothing about nondeterminism that needs to be understood apart from our usual intuitions

about algorithms. Such a view is implicitly supported when the equivalence between (unbounded) deterministic and nondeterministic Turing machines is presented as evidence for the Church Turing thesis. But suppose we take Smith to be reminding us of something more fundamental than just the proper use of abbreviations. Likewise, suppose it is possible that our understanding of nondeterminism really is limited as Papadimitriou suggests. The question then becomes one of making sense of nondeterminism *per se* and it is in this context that we find conflicting intuitions beginning to emerge.

Let us consider another open problem in complexity theory. We define the class coNP as the class of languages whose complement¹ can be decided by a nondeterministic Turing machine in polynomial time. Now when we ask whether $\text{NP}=\text{coNP}$ we run into a problem. In the case of deterministic machines the analogous question ($\text{P}=\text{coP}$?) is trivially settled by observing that a Turing machine which decides a language in polynomial time can be modified to decide the complement language in polynomial time simply by interchanging the "accept" and "reject" states. It is tempting to make the same argument in the case of nondeterministic machines, but we cannot.

To understand why we cannot, let us again consider the language **SATISFIABLE**. As we observed above, **SATISFIABLE** can be decided by a nondeterministic algorithm and, moreover, it can be decided in polynomial time.² Hence, **SATISFIABLE** is in NP

¹ Given a finite alphabet, Σ , the complement here is taken with respect to a decidable set of strings, $S \subseteq \Sigma^*$ so that given a language, $L \subseteq S$, the complement $L^c = S - L$ (and not $\Sigma^* - L$).

² How? Recall that the nondeterministic machine does not survey all of the exponentially many assignments. Rather, it guesses an assignment if such an assignment exists (in a single step) and then verifies that the assignment does indeed satisfy the

and its complement language of all Boolean formulae that are not satisfiable is in coNP. If we take the nondeterministic machine that decides SATISFIABLE and simply switch its "accept" and "reject" states (as we do to show that $P=coP$) the new machine will not decide the complement language of Boolean contradictions but rather the language consisting of all those Boolean formulae which have at least one falsifying assignment (cf. Bovet and Crescenzi 1994, p. 134). Had our naive solution worked, we would have demonstrated that $NP=coNP$.³ Obviously, we cannot infer that $NP \neq coNP$ from the failure of our single attempt, but we can look at our failure as a concrete example where the nondeterministic polynomial-time algorithm we use to decide one language cannot be transformed into another nondeterministic polynomial-time algorithm to decide the complement language.

There is something peculiar about the explanation we just gave but once again we find ourselves in a position where it seems that if we have to ask, we must not understand the question. Even though the our explanation is drawn from a canonical example, text books devote hardly a sentence or two to its explanation. For example, Bovet and Crescenzi point out that it is "immediately" clear that the naive attempt to interchange "accept" and "reject" states fails, but then they observe without further explanation that,

formula in question. The verification can be performed in polynomial time. In fact, if the formula is presented in conjunctive normal form, the machine need only scan the input once to see if the assignment makes a single literal true in each conjunct.

³ For suppose we have a NP-complete language, L (nb., SATISFIABILITY is NP-complete), such that $L^c \in NP$. Let $L' \in NP$. Since L is NP-complete, $L' \leq L$, and moreover, $L'^c \leq L^c$. But $L^c \in NP$, so $L'^c \in NP$ and $L' \in coNP$. Hence, $NP \subseteq coNP$. Conversely, suppose $L' \in coNP$. Since L is NP-complete, L^c is coNP-complete (since for any $L'' \in coNP$, we have $L''^c \in NP$, and so $L''^c \leq L$, hence $L'' \leq L^c$), so $L' \leq L^c$. But since $L^c \in NP$, $L^c \leq L$. Hence,

"All attempts to design a nondeterministic polynomial-time Turing machine deciding SATISFIABILITY^c [i.e., the complement of SATISFIABILITY] have failed up to now" (Bovet and Crescenzi 1994, p. 134). Obviously, there is more to understanding *why* the naive solution fails than simply observing that it *does* fail, as Bovet and Crescenzi do; for if it were a trivial matter to understand why various attempts to solve to the NP=coNP problem fail we wouldn't be here discussing an *open* problem. In fact, when we try to understand what exactly has gone wrong with our naive solution to the NP=coNP problem, we uncover a tension between our intuitions about algorithms and our intuitions about nondeterminism as a useful mathematical device.

For instance, there is a very strong pre-theoretic intuition that algorithms compose (think of function calls and subroutines here). Let us recast our original impulse to interchange "accept" and "reject" states more precisely in this light: Let us take theory on its face and assume that we have a nondeterministic polynomial time *algorithm* that *decides* SATISFIABILITY; that is, we have an nondeterministic algorithm that takes a Boolean formula as input and outputs either "yes, satisfiable" or "no, not satisfiable" according to whether the formula is in fact satisfiable. Of course, we also have a deterministic constant time algorithm that takes either "yes, satisfiable" or "no, not satisfiable" as input and outputs accordingly either "no, not contradictory" or "yes, contradiction." Put these two algorithms together and it would seem we have a nondeterministic polynomial time algorithm to decide the set of Boolean contradictions.

Have we just shown that NP=coNP? The answer is an emphatic "no," but let us be

$L' \leq L$, and $L' \in \text{NP}$. So, $\text{coNP} \subseteq \text{NP}$. (See also Bovet and Crescenzi 1994, pp. 134-140).

clear about the reason. The text book explanation does not attribute our nonsolution to a peculiar constraint on of the output of the nondeterministic machine or, for that matter, how the output should be understood. Indeed, such discussions about the nature of symbol and interpretation are never even addressed. Rather, on the received view, our composite algorithm fails because of a fundamental asymmetry between *yes* and *no* outputs from nondeterministic algorithms (cf. Bovet and Crescenzi 1994, p. 134; Papadimitriou 1994, pp. 45-46).

Again, it is helpful to think about our non-solution in terms of the computation trees we associate with nondeterministic Turing machines. The composite algorithm we propose yields a "yes, contradictory" answer even if there is only a single rejecting branch in the computation tree of the constituent algorithm for SATISFIABILITY. What we need is a composite algorithm to say "yes, contradictory" only if *every* branch in the computation tree of the constituent algorithm for SATISFIABILITY rejects, but we do not get that algorithm by composition.

Speaking more generally, we might say that nondeterministic Turing machines have a very weak input-output relation, or even that the sense of *decide* associated with nondeterminism is very liberal (Papadimitriou 1994, p. 44). In fact, the relation is so weak and the sense of *decides* is so liberal that the output of a nondeterministic machine cannot be used as input to a deterministic machine. It does not follow, however, that we can never compose deterministic and nondeterministic algorithms, for we talk about nondeterministic algorithms with deterministic components all the time (e.g., every time we display an NP-complete problem). The problem is that deterministic and

nondeterministic algorithms have different "implementations," so to speak, and we must take care that composite algorithms produce their answers in the right way. In this case, it turns out that our composite algorithm does not accept in the right manner and it ends up saying "yes" too often.

So, do we abandon our pre-theoretic intuitions about composition? I do not think so, for up till now we have seen that the heuristics we associate with nondeterminism are inessential, but when we understand why our composite algorithm fails, we finally see an instance where it makes a difference how we think about nondeterminism. In particular, we see that our composite algorithm fails because it is so easy for a nondeterministic Turing machine to output "yes" and so hard to output "no." The fact that nondeterministic machines can guess correctly is essential here and it manifests itself in the asymmetry between "yes" and "no" answers; a nondeterministic machine can always make a correct "yes" guess (if one exists), but it cannot make all the necessary "no" guesses. If we do not think about nondeterministic machines as always guessing correctly, we do not have the asymmetry, and moreover, we do not have the putative separation between NP and coNP. Note that the guessing here is not random, nor is it a reflection of an irreversible process. Rather, we are committed to a view in which the nondeterministic machine is somehow inspired (either to make the correct guess all the time or to manifests the correct behavior among all implicitly parallel behaviors). If a nondeterministic machine were to guess randomly, then the very process that might lead it to discover an accepting branch would also ultimately lead it to a survey of all the rejecting branches, and again, we wouldn't have the asymmetry on which the putative separation between NP and coNP rests. Thus,

it is theoretically important that we understand nondeterminism in terms of a very peculiar sense of guessing. More to the point, there is nothing about an inspired guess that can be reconciled with our intuitions about what it is to be an algorithm, what it is for a process to be physically nondeterministic or even what it is that the unlucky mathematician does.

No doubt some will see foregoing arguments as too literally-minded. Perhaps they will view words like *algorithm*, *output* and *decides* more metaphorically and will instead point to another more abstract characterization of the $NP=coNP$ question. For them, the suggestion that $NP \neq coNP$ amounts to a claim that there is something fundamentally different about deciding SATISFIABILITY and deciding the set of Boolean contradictions. They will readily agree that there is nothing realistic about the inspired guessing of a nondeterministic algorithm, but that such intuitions underwrite an interesting mathematical characterization of a class of problems.

To make good on such a claim we need a characterization of the $NP=coNP$ problem that allows us to focus our attention on intrinsic features of the decision problems at hand rather than our intuitions about machines. So let us forget about Turing machines and algorithms and instead consider how we decide SATISFIABILITY and the set of Boolean contradictions using something like a truth table. The first thing to notice is that we can certify that a Boolean formula is satisfiable by looking at a (well-chosen) single row of a truth table where, by contrast, a single row in a truth table can tell us only that a Boolean formula is not a contradiction. We might say that problems in NP have succinct *certificates* while problems in coNP have succinct *disqualifications*. That is, it is

easy to certify that a formula is satisfiable and it is likewise easy to certify that a formula is not a contradiction, but it is hard to certify that a formula is not satisfiable or that it is a contradiction. Once again, we see an asymmetry between certifying a Boolean formula as satisfiable and verifying a formula as a contradiction, only this time the difference is evident in the relationship between negated quantifiers rather than in our definition of acceptance by a nondeterministic Turing machine. Moreover, given this more general logical characterization, it really does seem that we have hit upon an intrinsic feature of the NP=coNP problem as opposed to some artifact of a peculiar machine definition.

But recall that a decision procedure is given by describing algorithms for *both* positive and negative tests, and the negative test for SATISFIABILITY involves inspecting every row of a truth table, which is exactly what we must do to verify a formula as a contradiction. As the name of the NP=coNP question suggests, the problem of deciding SATISFIABILITY and the problem of deciding the set of Boolean contradictions are completely complementary problems. Indeed, when we consider both positive and negative tests, the only asymmetry between the two decision procedures is that wherever the one says "yes" the other says "no" and conversely; a complete specification of one algorithm together with a trivial transposition of answers gives, *ipso facto*, a complete specification for the other algorithm.

Moreover, the observation that we need only inspect a single row in a truth table to certify that a formula is satisfiable, but that we need to survey every row to certify that a formula is contradictory, is not quite the difference in kind that we might expect. Indeed, such an observation is cold comfort to the beginning logic student who knows

full well how to use a truth table to decide satisfiability, but must take an exam where all the Boolean formulae have been deliberately constructed so that a single satisfying assignment appears in the last row in the canonical enumeration of truth-assignments. The point here is not how to make freshmen logic students unhappy (although that point is certainly worth noting), but rather, that it is hard to appreciate two algorithms as fundamentally different just because one *might* terminate before the other.

The trick is to make these points without seeming ignorant or stubborn. We do not mean to suggest that complexity theorists have overlooked a trivial nondeterministic polynomial-time algorithm to decide all Boolean contradictions and we should take seriously the *putative* separation between NP and coNP suggested by the failure so far to find an NP-complete language whose complement is also in NP. We should also recognize the fact that the putative separation between NP and coNP allows a finer grained classification for some problems.⁴ Rather, our point is that understanding simple *nonsolutions* to the NP=coNP either entails a view of Turing machines that weighs against some very natural intuitions about algorithms or shows that we must embrace a more general logical distinction between algorithms that seems to mark no substantive difference outside of a purely theoretical context.

⁴ As Papadimitriou explains, we can use NP and coNP to classify problems that require exact solution (1994, p. 412). For example, consider the *exact* traveling salesman problem where we are given a list of cities, a distance matrix and an integer k . The problem is to decide whether there is tour of all the cities where the total distance traveled is exactly k . There is no obvious nondeterministic algorithm to decide the problem directly, but we can classify it as the intersection between an NP problem and an coNP problem: namely, is there a tour that covers a distance no greater than k , and is there a tour that covers a distance of at least k .

The NP=coNP problem represents a crossroads of sorts. It marks a point at which we turn away from a discussion where intuitions about nondeterminism resonate naturally with our intuitions about machines and what it is to be algorithmic, to a discussion of nondeterminism *per se* in which we seem to abandon our intuitions about algorithms. Consider how Papadimitriou describes the situation,

The nondeterministic Turing machine is not a true model of computation. Unlike the Turing machine [i.e., the deterministic Turing machine] and the random access machine, it was not the result of an urge to define a rigorous mathematical model for the formidable phenomenon of computation that was being either envisioned or practiced. Nondeterminism is a central concept in complexity theory because of its affinity not so much with computation itself, but with *the applications of computation*, most notably logic, combinatorial optimization and artificial intelligence (1994, p. 49, emphasis in the original).

We see now the extent to which our intuitions about nondeterminism and computation have given way to intuitions about nondeterminism as a useful mathematical device. There is, of course, a presumption that these intuitions are reconcilable—that we are standing not so much at a crossroads of two divergent paths but rather at a point of contact between two approaches to the same theory. But we have good reason to doubt such a presumption. For starters, when we reflect on the original motivation for a theory of complexity we find that researchers felt it was time to "deal realistically with the quantitative aspects of computing" (Hartmanis and Hopcroft 1971, p. 444).⁵ At the very least, recognizing the original motivation for a theory of computational complexity suggests that there will be a non-trivial story to tell about how we got from a theory

⁵ We'll also see below that even automata theory, the more general theory of "abstract" machines, was often motivated with an appeal to the better understanding of real, concrete computers.

ostensibly concerned with concrete machines and real algorithms to one where the focus has shifted to, among other things, the finite model theory of second order logic. (We will try to tell that story below and in more detail in Chapter 4.) Our immediate concern, however, is how to make sense of the seemingly contradictory intuitions about nondeterminism.

Let us summarize the conceptual development. Understanding the received theoretical view often begins with a demonstration of the equivalence between deterministic and nondeterministic Turing machines. The proof is presented as further evidence for the Church Turing thesis. When we impose a polynomial-time bound that equivalence comes into doubt, but our intuitions about nondeterminism remain anchored to thoughts about algorithms and machines insofar as we understand nondeterministic algorithms as colorful descriptions of exponential search spaces. But as we delve more deeply into the theory, and as we begin to think about nondeterminism *per se*, we find an example where it is hard to understand the problem without giving up our intuitions about algorithms. Indeed, the anchor to our intuitions about algorithms is lost when we must think about nondeterminism in terms of inspired guessing. The justification for such a view of nondeterminism is that it underwrites a (peculiar) distinction, and hence, yields an interesting mathematical characterization of a class problems. All the while, we talk about *the* nondeterministic Turing machine, as if the intuitions surrounding it were a clear and consistent. But when we look closely we find a tangle of intuitions.

In their classic text, Hopcroft and Ullman suggest that we might avoid all this confusion "As long as our intuitive notion of 'computable' places no bound on the

number of steps or the amount of storage..." (Hopcroft and Ullman 1979, p.166). In other words, we might still identify the Turing computable functions with the intuitively computable functions, think about nondeterminism as further vindication for such a position and simply ignore complexity theory.

Or perhaps we should ignore the Turing machine instead. This is exactly what Stewart suggests when he announces "The Demise of the Turing Machine in Complexity Theory"(1996). At first blush, it is hard to imagine how we could even hope to develop a theory of complexity without some robust, undergirding notion of algorithm (either Turing's or one of the equivalent notions). It turns out that Stewart's motivations are more pragmatic than his sensational title suggests: he wants to strip away the cumbersome detail of Turing machine "code" from the theory and hopes that the vast resources of formal logic might help crack some of the seemingly intractable problems in complexity theory. But on a deeper level Stewart's motivations reveal exactly what is entailed in the shift from a more intuitive theory of complexity to the theory about the applications of computing: without the Turing machine and the concomitant intuitions about algorithms, assumptions about nondeterminism are justified by their theoretical fecundity rather than their intuitive cogency. As we observed above, "NP" becomes just a label for some class of problems. All that matters on this view is that we have some way of characterizing classes of problems and that characterization can be logical and abstract or premised on an "*unrealistic* model of computation" (Papadimitriou 1994, p. 45, emphasis in the original).

Theory will often have a life of its own quite apart from practice. Even if the idea

of a guessing computer is anathema to today's digital engineer,⁶ the idea of a nondeterministic algorithm clearly plays a central role in today's complexity theory. Still, we confront a philosophical dilemma when we take the theory at face value: we have just seen that there is tension among our intuitions about algorithms, determinism and resource bounds; either we ignore resource bounds (and 35 years of theoretical work) when we formalize our notion of algorithm or we embrace a theory that has seemingly divorced itself from our natural intuitions about algorithms. It is possible that the theory itself might someday reconcile these positions, but given the extant work on relativized computation and the more recent attempts at independence proofs for the P=NP question, we must remain circumspect about the theoretical ends justifying the means.⁷ In the meantime, it will be hard to tell whether continued work reflects the articulation of a detailed theory of complexity or the first symptoms of a degenerating research program pursued "purely for aesthetic reasons" (Stewart 1996, p. 222). A wait-and-see attitude will not help us here. Pace Hopcroft and Ullman, we can neither ignore complexity theory nor wait for results that *might* someday reconcile our intuitions about

⁶ And it really is. Speculation about future technology notwithstanding, as we indicated above, one of the central motivations for today's synchronous design paradigm is the *predictable* evolution from state to state in the control of a computer. The idea that the next state might not be completely determined by the present state and input is more likely to come up in discussions of transition or output races and it is hardly a welcome thought. See, e.g., (Prosser and Winkel 1996, pp. 170-175, 191-194).

⁷ We also do well to remember some of the unexpected results about space complexity here. For example, reflecting on (Savitch 1969), Hartmanis recalls that he "never suspected" a sub-exponential deterministic simulation of tape-bounded nondeterministic computations (Hartmanis 1981). And more recently, Immerman's 3-page (1988) result settled a longstanding problem proving that nondeterministic space is closed under complementation—a notoriously difficult problem that was at one time conjectured to

nondeterminism.

3. One Idea from Three Traditions

Given that it is so hard to see where the theory of computational complexity is going, it might make more sense to see where it has been. Indeed, if we could place the idea of a nondeterministic Turing machine in a robust historical context, we might be able to make more sense of the philosophical dilemma described above. But making historical sense of nondeterminism turns out to be a doubly hard task; not only has the historical work on theoretical computer science just begun, historiographic tensions have already become evident. Insofar as there is a received history of nondeterminism, it is characterized somewhat oddly both as a continuous and seamless development of ideas and as a locus for the unexpected convergence of seemingly orthogonal interests. For example, in their seminal paper, Hartmanis and Stearns (1965) cite Davis (1958) who, in turn, refers the reader to a long list of references to Kleene and Post. And, of course, everyone mentions Turing's 1936 paper. Thus there appears to be a natural progression of work starting with Turing and culminating in the theory of computational complexity. At the same time, however, theoretical computer science (broadly speaking) is widely recognized to be the product of three different disciplines. This convergence occurred in the late fifties and early sixties as ideas from recursion theory, formal language theory and complexity theory came together as the central threads in theoretical computer science. These conflicting historical emphases make it all the harder to uncover the origin of the nondeterministic Turing machine. Nevertheless, let us do our best and proceed by

have a *negative* solution.

reconstructing the received history around the framework given by Greibach (1981).

We begin by noting that although Turing mentions "choice machines" in his 1936 paper, they are seemingly excluded from the formal development of Turing's Turing machines.⁸ As Hodges (1983) points out, whatever Turing thought about nondeterminism is probably better understood in either a much wider philosophical context of free will versus determinism or, perhaps owing to his experiences with cryptography during the Second World War, how machines might be constructed "whose behavior *appears* quite random to anyone who does not know the details of their construction" (Hodges 1983, pp. 441-442). Moreover, those who carried on Turing's tradition by developing a rich theory of computation focused exclusively on deterministic machines. In his 1957 address on "The Present Theory of Turing Machine Computability," Rogers describes the theory as an investigation of what can be done on a digital computer with "explicit deterministic programs of instruction" (Rogers 1969, p.130). Likewise, except for a footnote excluding machines with "random" elements from the discussion, there is simply no mention of nondeterministic machines in Davis' influential 1958 text on computability theory. Finally, Minsky (1967, p. 314) comes closest to discussing the what we would call nondeterminism when he describes (in the index) a "non-determinate" machine "whose behavior is not entirely specified in the given description." He goes on to note, however, that such machines are not discussed in his text. Thus we see that idea of a nondeterministic machine, though recognized at some level, was not considered germane

⁸ Actually, there is much more to say here, but we will postpone that discussion until Chapter 4. We give a hint of the discussion in note 11 below.

to the study of algorithms.

By 1957, the theory of effective computability was well developed mathematically with applications ranging from the study of logic and mathematical foundations to recursive analysis. But it was no longer a theory which that with practical questions about real computers (see, e.g., Rabin and Scott 1959; Rogers 1969). The Turing machine was so powerful a model of computation as to be uninteresting in many applications (see, e.g., Chomsky 1959, p.138). Then in 1959 Rabin and Scott published their "Finite automata and their decision problems," pointing to the finite automata as a "better approximation to the idea of a physical machine" (p. 2). More important, that paper contained what is widely believed to be the first explicit, formal discussion of a nondeterministic machine along with a proof of the equivalence between deterministic and nondeterministic automata.⁹

At roughly the same time, the Chomsky hierarchy was introduced in (Chomsky 1959). The hierarchy consists of four types of grammar (corresponding to four kinds of formal languages) each of which is characterized in terms of the restrictions it places on the rules of that grammar. Proper containment among the four corresponding languages

⁹ Once again, the proof introduces an exponential explosion, only this time the jump comes in the number of states and not the amount of time required to perform the simulation. Unfortunately, the simulation of nondeterministic automata does not generalize to Turing machines and so the Rabin and Scott paper, as famous as it is, brings us no closer to understanding when the equivalence between unbounded deterministic and nondeterministic Turing machines was established. The best I can do is to point to three sentences in a footnote in Turing's original paper where he describes how to start with a choice machine to construct an "automatic" (i.e., deterministic) Turing machine. But there is a long story to tell about that note concerning the perceived naturalness of Turing's account and perhaps ultimately the widespread acceptance of his model over

was demonstrated along with some interesting affinities to recursion theory (e.g., type-0 languages = r.e. sets, type-1 languages = recursive sets). More significant, connections between grammars and machines were developed. Greibach points out that by 1958 Chomsky and Miller had essentially defined the type-3 grammars in terms of finite automata and that subsequent work on the finite-state languages most often pointed to the 1959 paper by Rabin and Scott (Greibach 1981, p.18). Here we see an unexpected convergence: the connections between the theory of abstract machines and formal grammars is one of the first overtures between two otherwise disparate research programs.

On the received view it is easy to see how this connection grew stronger with the development of type-2 grammars and their relation to push-down automata. The idea of a last-in-first-out storage had applications to the syntactic analysis of both natural and artificial languages; in particular, it was suggested in 1960 that the push-down store might be useful in the compilation of ALGOL while at the same time it was being used in the mechanical translation of Russian into English (Greibach 1981, pp.19-22). In 1962 Chomsky proved the equivalence between the context-free languages and those accepted by nondeterministic push-down automata, and by 1964 it was clear that the deterministic context-free languages were properly contained within the nondeterministic context-free languages (Greibach 1981, p. 22 & p. 25).

The last machine-grammar connection to be made in the Chomsky hierarchy was between the type-1, or context-sensitive grammars, and the linear-bounded automata.

others models of computation.

Unlike the finite state machine or the push-down automaton, the linear-bounded automaton was not a new kind of machine but simply a Turing machine (originally deterministic) limited by the input length in the amount of tape it could use (Greibach 1981, p. 23). But if the model seemed familiar, the questions it raised were not. Greibach recalls that despite the other well known equivalences between deterministic and nondeterministic machines, proofs with respect to the context-sensitive languages seemed difficult because no one was really accustomed to thinking nondeterministically (Greibach 1981, p. 24). So it seems that thinking about linear-bounded automata must have made thinking about nondeterminism more familiar. It also seems that the focus on the linear-bounded automaton would have made for a very natural dovetailing between automata and formal language theory on the one hand and complexity theory on the other. For by 1964 not only had Kuroda completed the Chomsky hierarchy by relating the type-1 grammars to nondeterministic linear-bounded automata, but Hartmanis and Stearns had introduced a robust theory of computational complexity. The study of linear-bounded automata finally presents us with a candidate problem around which different ideas about machines, determinism and resource bounds might have crystallized.

Unfortunately, the history we have just presented raises more questions than it answers. For instance, Hartmanis recalls that it was an unnatural hitch in Rabin and Scott's (unpublished) definition of two-way automata which led Myhill to define the linear-bounded automaton in 1960. Hartmanis goes on to remark how such "an innocuous and unnatural model can trigger a fruitful investigation" (Hartmanis 1981, pp. 48-49). True enough, but on a view like that we are forced to ask how it happened after 1965 that

nondeterminism, by then an assumption that could no longer be regarded as innocuous, was reconciled with a complexity theory where the time-bounded *deterministic* Turing machine was chosen as a model for its obvious resemblance to the "operation of a present day computer" (Hartmanis and Stearns 1965, p. 287). It is likewise remarkable that the first serious questions about nondeterminism were raised with respect to the linear bounded automaton, which itself was considered to be a "more natural model for computers" (Hartmanis 1981, p. 48). Part of the answer comes from Greibach who recalls that attention had centered on the context-sensitive languages because the connections at the top of the hierarchy (type-0 = r.e. languages) had already been worked out by Chomsky and because the equivalence between unbounded deterministic and nondeterministic machines had been proposed in a 1963 dissertation by Evey (Greibach 1981, p. 24). Greibach's claim is surprising, for the 1959 proof by Rabin and Scott of the equivalence between deterministic and nondeterministic finite state machines is consistently cited, while the Evey dissertation, when it is cited, only appears in the context of push-down automata (see e.g., Hopcroft and Ullman 1969, pp. 45 & 79). It seems odd that an equivalence proof that is so ubiquitous in text books and so troublesome in complexity theory could hang on such a recondite source.

Thus it seems that while the other nondeterministic machines were explicitly introduced as they related to the Chomsky hierarchy, the idea of a nondeterministic Turing machine crept into theoretical computer science rather quietly. In the case of finite automata, nondeterminism was presented as a conservative assumption. The assumption proved to be more interesting with respect to push-down automata when

separation between the deterministic and nondeterministic machines was established. Even in the case of tape-bounded automata, where answers have often been a long time coming, it must have been reassuring when Savitch (1969) demonstrated that a nondeterministic machine working in polynomial space could still be simulated deterministically in polynomial space. But while nondeterminism was proving to be rather well-behaved with respect to space, there was no such comfort to be found with respect to time. Cook (1971) introduced the idea of an NP-complete problem (i.e., a problem which characterizes the difficulty of all the problems solvable by a nondeterministic machine working in polynomial time) and by 1972, Karp had presented a veritable laundry list of NP-complete problems and demonstrated that either all the problems are solvable in deterministic polynomial time or none of them are. Thus began a long and philosophically strange journey into complexity theory.¹⁰

4. Where do we go from here?

Although there are unsettling gaps in the received history, there are also hints of a grand continuity. For instance, like Hilbert long before him, Cook was preoccupied by questions about mechanical theorem-proving and hoped that his work would "bring out fundamental limitations and suggest new goals to pursue" and ultimately "stimulate progress toward finding better complexity measures for theorem provers" (Cook 1971, p.157). Once again we see affinities between logic and complexity theory, only this time the emphasis is historical and it suggests a new way of understanding how the theorist is

¹⁰ Moreover, the bibliographic trail from Karp through Cook points back to familiar sources, (Hopcroft and Ullman 1969) in particular, and again fails to reveal a reference to

(and has been) able to countenance an otherwise counter-intuitive notion of a nondeterministic Turing machine. The story goes something like this: The longstanding historical connections between logic and complexity theory are not accidental; ideas and techniques from one discipline have inevitably made their way into the other. The nondeterministic Turing machine is a case in point: rather than try to understand nondeterminism directly, we should instead think of how we go about proving theorems in an axiomatic system. The first thing we notice is that although the rules of proof are strictly specified in an axiomatic system, the system itself does not determine which rule is applied at any point in a derivation. In exactly the same way, while a nondeterministic Turing machine has only finitely many different configurations it might realize on the next step, there is no telling which configuration it will realize. The nondeterministic Turing machine is thus seen by many in a natural context, both historically and conceptually, as a useful way of characterizing derivations in formal systems.

There is something comforting about this story, but ultimately it does not explain the gaps in the received history, it only conceals them. When we look carefully at the original theoretical justifications for nondeterminism we find a wide variety of motivations, some of which are logical while others are not. Ultimately, the study of these disparate motivations brings us no closer to resolving the philosophical dilemma we described above.

Having failed to find any historical solace, we can only hope that our worries will someday be resolved theoretically. There are three avenues of research in the current

the explicit introduction of the nondeterministic Turing machine.

state of the art. In one direction the theory becomes even more abstract; in the other we see the signs of a theoretical regress; and in the third direction we see the potential for the theory of computational complexity to be rendered moot.

4.1 Looking at *alternation* to understand nondeterminism

Chandra and Stockmeyer (1976) introduced the idea of *alternation* as a generalization of nondeterminism. As we indicated in Chapter 1, the idea is to consider the computation tree we associate with a nondeterministic machine, and imagine that at some nodes in the tree the machine must survey *all* the branches below that node, while at other nodes the machine need only survey a single branch. In this sense, the machine *alternates* between what we might call existential and universal behaviors. Alternating machines generalize nondeterministic machines in the sense that nondeterministic machines exhibit only the existential behavior (i.e., nondeterministic machines choose a single path through the computation tree). Chandra and Stockmeyer established some interesting relations between time and space complexities and subsequent work (e.g. Kannan 1981; Paul et al. 1983) has shown separation between deterministic and nondeterministic time under particular constraints. Many, however, see results like these as symptomatic of trend toward scholasticism in complexity theory. Moreover, results like these focus on the applications of computation and, as we have already suggested, the philosopher's computational interests are more likely to resonate with issues of mechanism rather than logic. While the persistent use of the word "machine" suggests continuity, it is not clear that intuitions about algorithms find any natural place in the discussion of a machine-free complexity theory. This is not to say that such an

explanation cannot be given, but rather that it has not yet been given. Without a more robust philosophical account relating notions of effectiveness and formal logic (i.e., an account that does not presuppose that the connection has already been forged by the work in theoretical computer science), focus on alternating Turing machines will do little to resolve our conflicting intuitions about nondeterminism.

4.2 Redefining nondeterminism

A second direction is followed by Spaan, Torenvliet, and van Emde Boas (1989) who modify the definition of acceptance by a nondeterministic machine in such a way that the recursion theoretic distinction between the recursive and recursively enumerable sets (a distinction that never had anything to do with nondeterminism) can be viewed in analogy with the putative distinction between P and NP (a distinction which has everything to do with nondeterminism). Their idea is to impose a *fairness* condition such that a nondeterministic machine, when given a choice between transitions to two different configurations, is guaranteed to explore both transitions after *some* finite number of steps. In other words, a *fair* nondeterministic machine will eventually explore all its choices, whereas an *unfair* nondeterministic machine is guaranteed to survey only those choices that lead to acceptance, provided such a sequence exists. There is no guarantee that an unfair machine will explore all the choices available to it.

The motivation to introduce the notion of a fair nondeterministic machine follows from

the self evident observation that in the world of unbounded computation nondeterministic devices are more powerful than deterministic ones as exemplified by the inequality $REC \neq RE$... the nondeterministic devices could guess and verify the halting computations which a deterministic device cannot

produce (Spaan, Torenvliet, and van Emde Boas 1989, p.187).

In particular, we can imagine a fair nondeterministic Turing machine deciding the Halting problem¹¹ as follows. Let M be a fair nondeterministic machine M such that M can simulate any other deterministic machine M' on any input. During its computation, M chooses nondeterministically either to simulate M' or M simply prints a "0" and halts. If M' halts on its input, it does so after a finite number of steps, at which point M halts and prints a "1." If, however, M' does not accept its input, it will never halt and the simulation of M' could diverge (i.e., continue indefinitely). But since M is fair, it must after some finite number of steps must choose not to simulate M' and instead will print a "0" and halt. After some finite number of steps, M will either print a "1" if M' halts on its input, or M will print a "0" if M' does not halt on its input. Hence we have a fair, unbounded nondeterministic machine that solves the Halting problem—something no deterministic machine can do, and thus we have a separation between deterministic and nondeterministic devices in the unbounded case that parallels the suspected separation between such devices in the time-bounded case (Spaan, Torenvliet, and van Emde Boas 1989, pp.188-190).

Although the notion of fairness has well-established roots in the theory of concurrent processes, the argument Spaan, Torenvliet, and van Emde Boas give might seem a bit slippery.¹² Indeed, because it contradicts the well-know unbounded

¹¹ See (Rogers 1967, pp. 24-26) for a description of the Halting problem.

¹² In fact, the presentation of the proof given in (Spaan, Torenvliet, and van Emde Boas 1989, pp. 189-190) is a bit hard to follow. At one point they state that a bound is specified in advance indicating how many steps M must simulate M' and if M' "accepts

equivalence between deterministic and nondeterministic Turing machines, we might see the proof here as a *reductio* to establish that the notion of a *fair* nondeterministic machine has no place in the traditional theory of recursive functions. But even if we worry about the cogency of the proof, we should still take note of what has motivated Spaan *et al* to challenge the tradition. In particular, they report that,

All of this work was inspired by the frustration originating from the difficulty of the fundamental problem in computational complexity which has become known as the P=NP? problem (Spaan, Torenvliet, and van Emde Boas 1989, p.187).

They continue,

Given the difficulty of solving the P=NP? problem we have considered to modify the realm of recursive function theory instead. We propose in the following section the adoption of an alternative acceptance convention for the nondeterministic version of the Turing machine for recursive function theory such that the difference we suspect between determinism and nondeterminism in complexity theory can easily be established in the unbounded case (Spaan, Torenvliet, and van Emde Boas 1989, p.188).

Given the difficulty of the P versus NP problem, it is understandable that some will propose novel and indirect approaches to the problem. Still, there is something *ad hoc*,

within this number of steps then M halts and accepts also, else M halts and rejects." Of course, the Halting problem is solvable if we impose time bounds—if we know how long to wait, we let the computation unfold for that period of time and see whether the input has been accepted or not. The Halting problem gets its teeth when we can not say *a-priori* how long it will be before a machine accepts its input; that is, if we do not know how long it will be before an machine accepts, we can never be sure if we have waited long enough before we say that the machine rejects. I think what Spaan, Torenvliet, and van Emde Boas 1989 mean to say is that if M' accepts then it does so after some finite amount of time, otherwise we can rely on the fact that M is a fair nondeterministic machine to wait out a divergent computation "without risking infinite computations" (Spaan, Torenvliet, and van Emde Boas 1989, p.192). But more important than the technical details of the proof, or even whether the proof flies, for that matter, is the fact that Spaan, Torenvliet, and van Emde Boas have been motivated to start re-thinking long

perhaps even desperate, about redefining a notion so that a problem that was once contained in the theory of computational bleeds into the theory of recursive functions. Is this progress or regress? Indeed, nondeterminism is a conservative assumption, and hence, well-understood in the theory of recursive functions. While it might be possible to shed new light on nondeterminism in complexity theory by imposing the sort of distinction that Spaan, Torenvliet, and van Emde Boas propose, it might also be the first step toward a degenerating research program where a recalcitrant problem is explained away by tinkering with a more basic assumption.

4.3 Moving beyond the Turing machine

Finally, there is talk of moving "beyond" the Turing machine. The idea here is to view the $P \neq NP$ conjecture as a theoretically roundabout way of marking the real world distinction between the problems which are tractable in an absolute sense and those which are tractable when CPU time is sold by the second. That is, there are known algorithms for solving all the problems in NP, but they typically involve an exhaustive search of an exponential search space. Hence, they take a prohibitively long time to perform.

There is discussion of exploiting physical analogues (e.g., quantum mechanical systems or chaotic systems) in which the combinatorial explosion inherent in nondeterministic algorithms is conveniently collapsed by the physical system. There is even discussion of a physical version of Church's thesis relating the sense of computable to whatever it is that can *actually* be computed by a physical device. Unfortunately, none

standing definitions and equivalence results.

of this has much to do with the original Church Turing thesis (at least according to some of those discussing the idea, see Pitowsky (1990)) and is thus unlikely to help us reconcile our conflicting issues about nondeterminism. Our concern is not so much whether a problem in NP can be solved quickly, but whether the underlying notion of nondeterminism can be reconciled with our intuitions about algorithms.

Of course, we do well here to recognize the emergence of intuitions about nondeterminism in the physical sense. Indeed, it seems there is something to say about the possibility of using the nondeterminism inherent in a quantum mechanical system to address the sense of nondeterminism inherent in the P versus NP problem. Unfortunately, the physical sense of nondeterminism and the theoretical sense of nondeterminism seem to be at odds. In fact, Pitowsky distinguishes between the physically computable and the theoretically computable and goes on to talk about NP-complete problems which might have *physical* polynomial time solutions even if it happens theoretically that $P \neq NP$ (Pitowsky 1990). The possibility seems likely (provided we figure out how to build a quantum computer) since it is widely believed that $P \neq NP$. What would we say in such a situation? What would it mean to say that a problem like SATISFIABILITY is theoretically intractable if eventually happens that it is decidable in polynomial time by a quantum computer? In such a situation, the physical sense of nondeterminism would not illuminate the theoretical sense, it would instead render it moot.

Of course, it is also possible that we will never build a quantum computer (maybe $P \neq NP$ and there is no way around it) or perhaps we build a quantum computer and prove that $P = NP$. In both these cases the relation between the physical and theoretical sense of

nondeterminism would be more obvious. In one case it would be clear that the theoretical sense of intractability is absolute, in the other case we'd find that quantum computing is overkill. In the meantime, however, it is hard to see how to relate the physical and theoretical intuitions about nondeterminism.

The idea of a nondeterministic Turing machine is rooted in a long tradition in theoretical computer science and during the course of that tradition several ideas about nondeterminism have emerged. In particular, the nondeterministic Turing machine has been used to think about algorithms, resource bounds and the classification of mathematical problems. Although the tradition has implicitly established a presumption of continuity in both the historical and conceptual development of nondeterminism, it should be clear by now that such a presumption is ill-founded. The nondeterministic Turing machine cannot be adduced as evidence for the Church Turing thesis, and at the same time, be presented as a patently "unrealistic" model of computation. We cannot reconcile intuitions about machines with intuitions about inspired guessing.

IV. A Second Look at the Received History

0. Overview

In this chapter we will revisit the history presented in the last chapter. As we have seen, there are a handful of watershed papers in the development of nondeterminism. Unfortunately, none of these papers contain an explicit reference to the introduction of the nondeterministic Turing machine, nor do they reveal a unified context in which we can understand the theoretical motivations for nondeterminism. Our goal below is to place the received history in a more critical light and thereby reinforce the philosophical concerns we raised in the last chapter.

1. Introduction

We have argued that it is difficult to make philosophical sense of the nondeterministic Turing machine; now we will see that it is hard to make historical sense of it as well. We begin where we left off in the last chapter by noting that the received history never really pins down a date for the formal introduction of the nondeterministic Turing machine. There are vague allusions to the interplay between formal language theory, automata theory and computation theory, but explicit references to the first nondeterministic Turing machine are conspicuously absent. We get closer to an actual date with Greibach's citation of Evey's 1964 dissertation for the proof of equivalence between unbounded deterministic and nondeterministic Turing machines. In fact, Evey himself, having taken some care with questions of priority, claims his proof "appears to be new simply because nondeterministic Turing machines have not been discussed"

(Evey 1963, p. 2-71).¹ Unfortunately, the received history is inadequate. For starters, it pins down the wrong date for the introduction of the nondeterministic Turing machine. But the problems run much deeper than just a quibble about dates. Without a firm sense of when the nondeterministic Turing machine was introduced, we cannot understand why it was introduced which makes it all too easy to overlook philosophical tensions.

To sort things out we must start anew. Contrary to the received view, the history of the nondeterministic Turing machine really begins nearly thirty years before the Evey dissertation with Turing's seminal 1936 work. There we find the theoretical discussion of nondeterminism in Turing's so-called *choice machines*. Moreover, that discussion comes at a crucial juncture in the argument for Turing-machine computability as an adequate account of effective computability. Unfortunately, Turing's discussion of choice machines is also rather brief and it seems to have been overlooked or, perhaps, forgotten by his immediate successors who focused exclusively on deterministic computation. The notion of a nondeterministic machine does not surface again until 1959 with Rabin and Scott's (re)introduction of nondeterministic automata. Although Rabin and Scott have the same idea in mind, their motivation for considering nondeterminism was very different from Turing's. Next, there is Kuroda's 1964 proof for the equivalence between type-1 grammars and the languages accepted by nondeterministic linear-bounded automata. Kuroda's proof is important not only because it completes the work Chomsky began relating formal grammars to automata, but also because on the received view, the

¹ In fairness to Evey, we should note that he is not really interested in the Turing machine *per se*; rather, his goal is to use the pushdown store to achieve greater

nondeterministic linear bounded automaton marks a confluence of ideas concerning computation, determinism and resource bounds.² Finally, there is Hartmanis and Stearns (1965) and the subsequent work of Cook and Karp wherein the connections between determinism and resource bounds find their modern expression.

Our history of nondeterminism is thus divided into four episodes. In the next section we will look at Turing's work in the '30s and try to understand what he thought he had achieved by introducing the choice machine. In §3 we will explain away the 20-odd year absence of the nondeterministic Turing machine before looking at Rabin and Scott's work. We will also make note of some independent developments in the Soviet Union. §4 will be devoted to a discussion of the theoretical climate of the early '60s and the concerns that would ultimately lead to a theory of computational complexity. We will note the emergence of conflicting intuitions in the early '60s as researchers were driven by the desire to develop a realistic theory of computing and, at the same time, were "learning to think nondeterministically." Finally, in §5 we will look at Hartmanis and Stearn's seminal paper, which initiated a robust theory of complexity, and at the papers of Cook and Karp, which together defined the theory's central methodological approach and its most famous open problem.

In many respects, the history we describe here will resemble the received history we discussed in the last chapter; we will recognize the same mix of influences and we will examine the same classic papers. There is, however, an important difference. Where

theoretical unification among abstract machines (and grammars).

² Cf. (Hartmanis and Hunt 1973).

the received history emphasizes continuity and a seamless flow of ideas, we will find discontinuity and a variety of conflicting intuitions.

2. Turing's Nondeterministic Turing Machines

In §2 of his 1936 paper, Turing defines *automatic machines* and contrasts them with so-called *choice machines*:

If at each stage the motion of a machine (in the sense of §1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a-machine*).

For some purposes we might use machines (choice machines or *c-machines*) whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator (Turing 1965a, p.118, emphasis in the original).

Unlike an automatic machine, a choice machine needs the input of an operator to keep it running. We might wonder whether this makes choice machines incomplete or defective in some sense, but let us postpone that question and note in the meantime that the formal distinction between *a* and *c*-machines is the same distinction we now mark with the terms deterministic and nondeterministic. The choice here is what to do next given a particular combination of state and input and so, mathematically speaking, we have the familiar distinction between an automatic (i.e., deterministic) machine which is defined by a transition function and a choice machine (i.e., nondeterministic) machine defined by a transition relation. Moreover, the essential mathematical intuition we identified in Chapter 3, that of the nondeterministic machine's inspired guess, is also evident in Turing's discussion of choice machines insofar as we might think of the operator

knowing exactly what the machine needs to do next.

The choice machine does not appear again until §9 when Turing describes in a footnote how an automatic machine can be constructed to do the work of a choice machine. The problem is to construct a Turing machine to enumerate theorems in Hilbert's first order logic. Although Turing never discusses the details, it is easy enough to describe how a choice machine might work. For example, given an axiom as input (or axioms, or perhaps other theorems) a choice machine can scan the input, determine whether any of the rules of inference apply (a purely syntactic determination) and then prompt the operator to choose among the appropriate rules. With the inference rules "hard-coded" as subroutines in the machine's transition table, the machine takes the operator input, jumps to the corresponding subroutine to apply the rule (another purely syntactic task), and finally outputs the resulting theorem. With some careful bookkeeping, the operator can continue to feed in axioms (and previous output) as input and thereby enumerate first order theorems in almost any order he pleases. Clearly, writing out the actual transition table for such a machine would involve some fairly tedious detail, but there is nothing tricky here; the operator does the thinking and the machine does the all the syntactic grunt work.

Turing's ultimate goal, however, is to describe an automatic machine to enumerate theorems and at first it is not obvious how this might be done. Indeed, a machine can determine which rules apply and it can apply them, but deciding which particular rule to apply hardly seems mechanical—hence Turing's image of an external operator making all the decisions. So how do we get rid of the operator? The answer is to

replace inspiration with exhaustion. Rather than hanging at some point in a derivation waiting for an operator to choose a particular rule of inference, an automatic machine will draw every possible inference every time. The trick is to mechanize the bookkeeping so the automatic machine will generate theorems systematically, outputting each and every theorem without ever getting lost in an endless derivation. There are many ways to do the bookkeeping. For example, starting with all the axioms printed (and suitably delimited) on its tape, an automatic machine might scan the left-most axiom, determine which rules apply, enter each of the appropriate subroutines in some prescribed order (where, as before, the inference rules have been hard-coded into the control of the machine), append the resulting theorems to the end of the tape all before moving onto the next left-most axiom. Having applied each of the appropriate inference rules *once* to all the axioms the machine would start again with the left-most axiom and apply all the appropriate rules *twice* all the while appending the results to the end of its tape. Left to its own devices, the automatic machine will continue to grind through derivations of ever increasing length and thereby arrive (eventually) at every theorem that could be discovered by an operator working with a choice machine.

It is remarkable that an automatic machine can do the work of a choice machine, even though an operator working with a choice machine can produce a potentially infinite stock of theorems in any order he pleases. Things work out this way only because at any point in a derivation an operator can choose from at most finitely many rules of inference; hence, there are only finitely many distinct derivations of a given length. The behavior of the choice machine—that is, the sequence of decisions the operator makes—

can be systematically described. Turing exploits this fact by way of a straightforward arithmetization of choice sequences:

We can suppose that the choices are always choices between two possibilities. Each proof will then be determined by a sequence of choices i_1, i_2, \dots, i_n ($i_1 = 0$ or $1, i_2 = 0$ or $1, \dots, i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + \dots + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3 ... (Turing 1965a, p. 138).³

Here, finally, is the smoking gun we have been looking for. Although brief, Turing's three sentence discussion of choice machines and automatic machines provides just enough detail to construct a full blown proof for the equivalence between deterministic and nondeterministic Turing machines. Hence, credit for the nondeterministic Turing machine and the proof of its equivalence to the deterministic machine should be given to Turing and not, for example, to Evey nor anyone else working in the late '50s or early '60s. It is odd that the paper that first introduced the

³ There are few technical points to make here: First, there is no loss in generality in restricting our attention to choices between two possibilities. Suppose we are given a machine that must choose among n possibilities, c_1, c_2, \dots, c_n (where $n > 2$), at some point in its computation. A choice among n possibilities can be simulated by *series* of choices between two possibilities in the following way: We introduce new choices a_1, a_2, \dots, a_{n-2} ("a" for "all the other choices") to the original set of choices c_1, c_2, \dots, c_n . Rather than choose among all n possibilities at once, the machine first chooses between c_1 and a_1 , then (if necessary) between c_2 and a_2, \dots , then (if necessary) between c_{n-1} and c_n (see Hopcroft (1979, pp. 92-93) for their proof of the Chomsky Normal Form theorem).

Second, there is a subtlety in Turing's coding scheme. At first blush, it might seem that Turing is simply using binary numerals read as strings from left to right to represent a sequence of choices. This is almost correct, but we must remember that *leading zeroes are significant*. Hence, $2^n + i_1 2^{n-1} + i_2 2^{n-2} + \dots + i_n$ gives a decimal expression for a binary string of length n allowing for the possibility that we might be coding a string with leading 0's (or perhaps all 0's).

Finally, there is an unmistakable family resemblance between Turing's '36 proof and the equivalence proof given in (Hopcroft and Ullman 1979, p.164).

Turing machine is the last place people think to look for the nondeterministic Turing machine, but it is not too surprising given that Turing himself misdirects the reader's attention. In fact, immediately after introducing the choice machine in §2 Turing announces, "In this paper I deal only with automatic machines" (Turing 1965a, p.118). Given that disclaimer, the reader is likely to forget about choice machines altogether unless he reads the appendix to *Computable Numbers*. There, after reminding the reader about the footnote in §9, Turing makes a second appeal to choice machines in his proof of the equivalence between Church's λ -definable terms and his own computable sequences. But once again, it is easy to overlook Turing's mention of the choice machine; the reference is so brief as to appear incidental and it comes at a point when Turing seems to be wrestling with foundational questions about his entire approach. And thus we come to a more difficult question: if it is so easy to overlook the choice machine in Turing's analysis of the computable numbers why did he bother with it at all?

The answer is not straightforward and it requires that we disentangle several issues. To begin, let us consider the context in which §9 and the appendix were written. Recall that for his proof of the unsolvability of the *Entscheidungsproblem* to work, Turing needs a problem unsolvable by any systematic means; nothing is gained by proposing a notion of *computable* that happens to be just narrow enough to preclude a solution to the *Entscheidungsproblem*. So, how narrow is too narrow? Or, to put the question in its more familiar form, Is Turing's notion of *computable* wide enough? Turing himself poses the question in §9, "The extent of the computable numbers," but despairs, "All arguments which can be given are bound to be, fundamentally, appeals to

intuition, and for this reason rather unsatisfactory mathematically" (Turing 1965a, p.135). It is well known, however, that the appendix to *Computable Numbers* was added after Turing became aware of Church's work and its application to the *Entscheidungsproblem*. Suddenly there were three formal counterparts to the informal notion of algorithm, two of which (thanks to Kleene's 1935 work on λ -definability and the general recursive functions) were provably equivalent. Church's work raised the mathematical possibility that Turing's notion of *computable* might *prove* to be too narrow. The appendix is more than just a nod to priority; the demonstration that any number (or function) that is λ -definable is computable, and conversely, is crucial because it establishes Turing's notion of computable as a legitimate alternative to the two formalisms described by Church (1965).

Obviously, the demonstrations of extensional equivalence in §9 and the appendix dovetail nicely with arguments that the notion of computable is sufficiently wide. The fact that several independent accounts can be shown mathematically to pick out the same class of functions is compelling evidence that we have identified a robust notion of *computable*. In this light it is tempting to read the arguments of §9 and the appendix as more of the same kind of evidence—another two results in a long list of equivalences. Unfortunately, if we think only about equivalence we come no closer to understanding the role of the choice machine in Turing's argument. In fact, in the context of an equivalence theorem, Turing's discussion of choice machines seems hopelessly far removed from the end-result. Turing introduces the choice machine to enumerate theorems in Hilbert's logic, then argues that such a machine can be replaced by an

automatic machine, all to establish a result that is, ultimately, just a lemma needed in the proof that any number defined by a Hilbert-style first-order theory is computable and vice versa. Not only are Turing's references to the choice machine deeply buried, but those willing to follow this long chain of argument back to its beginnings are likely to regard the choice machine as an inessential, heuristic step in a more important equivalence proof.

To understand Turing's appeal to the choice machine, we must reconsider the role of equivalence results. While such results are significant in their own right, they are not at the center of Turing's attention. In fact, the arguments for the extent of the computable numbers come in three kinds. There is the "direct appeal to intuition," the celebrated analysis of man-as-computer working with a pen and paper. And there is an argument by way of example as Turing points out "large classes of numbers which are computable." The argument for extensional equivalence is actually sandwiched between these two other arguments. Moreover, it comes with an important qualification: the equivalence proof is given because "the new definition has a greater intuitive appeal" (Turing 1965a, p.135). Although Turing's remark is made parenthetically, it suggests that his goal is not an equivalence result *per se*, but rather to show how a given formalism can yield quite naturally to a computational analysis. Of course a formal proof of equivalence cannot establish one definition as more natural or more intuitive than another, but there is room for such subjective judgments when we consider the constructions that make up an equivalence proof—and it is here that the choice machine fits into Turing's argument.

The more difficult implication in the equivalence proof of §9 is showing that any

number that is definable is computable. (Turing sketches the proof for the converse implication in a single sentence.) Turing's sense of *definable* is peculiar; rather than give first-order definitions of numbers directly, Turing focuses on the binary sequences that represent numbers, in particular, those sequences that can be described by a finite conjunction of first order formulae. The idea is that a sequence will be definable when it can be described bitwise, so to speak. More formally, Turing introduces a predicate $G_\alpha(x)$ for each sequence α , which is read as "the x -th figure of α is 1." ($\sim G_\alpha(x)$ is read as "The x -th figure of α is 0.") Sequences are definable in Turing's sense only if, for each $n \in \mathbb{N}$, there is a provable formula asserting that the n^{th} bit of the sequence is 1 or there is a provable formula asserting that the n^{th} bit is 0 (but not both). So, given a definable sequence α , we construct a machine to proceed digit by digit; to compute the j^{th} digit the machine enumerates theorems until it finds a formula asserting $G_\alpha(j)$, in which case it prints a "1," or it finds a formula asserting $\sim G_\alpha(j)$, in which case it prints a "0" before moving to the next $(j+1)^{\text{th}}$ digit. By hypothesis, exactly one of the two formulae is provable and hence the machine always prints either a "1" or a "0" for each digit in the sequence. There is nothing surprising here: the machine works exactly as one would expect *given Turing's sense of definition*. But we have yet to see a natural, computational account of *definable*. Indeed, the real question is whether Turing's computational analysis is itself intuitively appealing; it is not enough to piggyback an account of computable onto an admittedly peculiar sense of definition. Whatever intuitive appeal there is in Turing's proof rests with the operation of the machine, especially when it comes to enumerating theorems. Unfortunately, Turing does not specify the automatic machine.

Instead, he claims that "The author has found a description of such a machine" (Turing 1965a, p.138). He does, however, preface that remark: "It is most natural to construct first a choice machine (§2) to do this. But it is then easy to construct the required automatic machine" (Turing 1965a, p.138). Again, some might claim here that the choice machine is heuristic; as Turing himself goes on to say, the goal is to go from choice machine to automatic machine and the choice machine helps with this step. Intuitively speaking, however, our interests run in the opposite direction. The choice machine grounds the automatic machine. Indeed, there is nothing natural or intuitive about an automatic machine spitting out theorems, but there is something familiar about the operation of a choice machine: the combination of operator and machine produces theorems in exactly the same way a mathematician working alone would. Even if we require the automatic machine for a formal equivalence proof, the choice machine makes the analysis intuitively compelling. The progression from choice machine to automatic machine shows off the computational aspects of proving theorems more clearly, and more naturally, than would be possible if Turing had jumped directly to the description of an automatic machine. In this sense, the choice machine is more than heuristic; it gives us reason to believe that the proof, and indeed the entire analysis, is on the right track.

Similar remarks apply to Turing's use of choice machines in the appendix. As we saw above, the formal equivalence between the computable and the λ -definable numbers is crucial to Turing's argument, but it is also significant that the choice machine of §9 can do the work of conversion in the λ -calculus. There is, of course, a deliberate and obvious similarity between derivation in a formal system and conversion in the λ -calculus.

Nevertheless, it is not clear that either will yield to Turing's computational analysis; even worse, it would be disconcerting if Turing could account for one formalism but not the other. The fact that the choice machine naturally relates both formalisms to Turing's notion of *computable* is, again, intuitively compelling. As we would hope, similar tasks are performed by similar machines.

The emphasis on intuitively appealing proofs of equivalence is clear not only in Turing's 1936 work but also in his (1937). In that paper, he gives a more detailed demonstration that every λ -definable function is computable, as well as a proof that every computable function is general recursive; these results together with the Kleene's 1935 proof that a function is λ -definable if and only if it is recursive established the equivalence between all three definitions. It might seem that Turing's work here is superfluous given that he had already sketched a proof for the equivalence of computable and λ -definable sequences in the appendix to the 1936 paper, but he goes on to say in the 1937 paper,

The identification of 'effectively calculable' functions with computable functions is possibly more convincing than an identification with the λ -definable or general recursive functions (Turing 1937, p.153).

Davis (1982) gives us good reason to believe that Turing was indeed correct when he suggested that his account of 'effectively calculable' might be more convincing. Kleene echoes similar sentiments, "[f]or rendering the identification with effective calculability the most plausible—indeed, I believe compelling—Turing computability has the advantage of aiming directly at the goal as is clear (and as Turing modestly suggested in 1937 p.153)" (Kleene 1981, p. 61). In fact, it is well documented that Gödel

found Turing's work on effective computability far more compelling than the work done by Church or Kleene. Davis points out that Gödel (and to some extent, Post) believed that an adequate analysis of *effective procedure* would begin with the familiar intuitions about algorithms and only then work toward a particular formal definition. Turing proceeds in exactly that way. For example, the 1936 paper begins, famously, with Turing's description of man-as-computer, where the extraneous details are stripped away until Turing arrives at the essential operations that define the Turing machine. Likewise, in his 1937 proof that every λ -definable function is computable. Turing first describes constituent machines which perform mundane tasks on λ -terms such as marking symbols, comparing symbols, matching parentheses, swapping symbols and searching a string for a given symbol. These machines are then composed in an entirely straightforward manner to produce a machine that enumerates all the possible immediate conversions from a given λ -term (i.e., the machine either reduces or expands the λ -term), and hence the entire process of λ -conversion is shown to be mechanical. The proof proceeds by way of a piece-meal analysis, which starts at a familiar, intuitive level and works its way to a more formal result.

Turing's progression from familiar to formal is best exemplified by the choice machine. Let us think about deduction in very general terms: particular formal systems are characterized by the axioms they admit, or the rules of inference they employ, while deduction more generally can be characterized as a series of choices. The difference between deriving one theorem rather than another is, at the most basic level, a matter of deciding to apply one rule rather than another. Turing presents the choice machine to

make this familiar aspect of theorem proving mechanical. Indeed, as Wang points out, the choice machine corresponds to "what mathematicians in fact do" (Wang 1974, p. 84). That is to say, the choice machine allows Turing's computational analysis of two different formal systems to start off on the right intuitive foot. Turing moves from the familiar to the formal. By contrast, Church and Kleene began with a formal definition, which was only later identified with the class of effectively computable functions. This is not to say that the work of Church and Kleene was any less important but, rather, to underscore Turing's emphasis on intuitively compelling accounts of *computable*.⁴

⁴ But we should mention in passing that, despite their mathematical significance, neither Kleene's account of the λ -definable terms nor Church's original appeal to the general recursive function hold much intuitive appeal as accounts of "effectively calculable." For instance, Kleene identifies algorithms for computation with a process of repeated reduction, which starts with the term representing the function as it is applied to a numeral and ends with a unique normal form for the terms that represent numerals. While Kleene's approach makes for a perfectly reasonable algorithm, it also makes for some hairy function terms. In fact, Kleene reports that coming up with a term to compute the predecessor function—a function trivially computed by a Turing machine—was something of a discovery (see Kleene 1981, pp. 56-57). Moreover, as pointed out by Davis (1982), even Gödel had a hard time with Kleene's 1936 proof of the equivalence between the general recursive and λ -definable functions. As for Church's appeal to the general recursive functions, although the primitive recursive functions capture intuitions about effectiveness in a rather obvious way, Ackermann's discovery of an intuitively computable function that is not primitive recursive makes formal definitions of general recursive functions decidedly less perspicuous. Suddenly recursion became a business of substitution of the most general kind. We might "see" addition and multiplication in the set of equations:

$$+(x, 0)=0, +(x, S(y))=S(+(x, y)), *(x, 0)=0, *(x, S(y))=+(x, *(x, y)).$$

But it is hard to see minimization, F^* , anywhere in the recursion equations given in (Church 1965, p.97) for a two-valued recursive function $F(x, y)$ (minimizing on y):

$$\begin{array}{lll} i_2(1, 2) = 2, & g_2(x, 1) = i_2(f_2(x, 1), 2), & i_2(S(x), 2) = 1, \\ h_2(S(x), y) = x, & j_2(1, y) = y, & i_2(x, S(S(y))) = 3, \end{array}$$

Although we just have taken a somewhat lengthy detour through the equivalence results of the mid 1930s, our goal is not to embark on the intellectual history of the Church Turing thesis. Rather, by looking seriously at the way Turing presents these results, we might understand his rather brief references to the choice machine. Contrary to the recent work of Spaan, Torenvliet, and van Emde Boas (1989) it does not seem that Turing regarded choice machines "an aberration to the notion of computability." The fact that such machines depend on the input of an external operator does not make them defective; it makes their operation more familiar and, hence, more natural. Moreover, Turing's emphasis on naturalness proves to be important both historically and conceptually. In summary, Turing presents the choice machine as a natural mechanical analogue for reasoning in formal systems. The behavior of the mathematician is clearly reflected in the workings of the choice machine, which makes Turing's computational analysis all the more compelling.

2. Rabin and Scott's Nondeterministic Automata

In the last chapter we argued that the formal notion of a nondeterministic algorithm is far removed from the intuitive notion. Now we can appreciate the irony in that development: initially the choice machine was presented as an intuitive touchstone, but now it is the source of some very counter-intuitive results. What's more, this conceptual break reflects a historical discontinuity as well. Although our notion of

$$\begin{array}{ll} h_2(g_2(x, y), x) = j_2(g_2(x, y), y), & f_1(x) = h_2(1, x), \\ j_2(S(x), y) = x, g_2(x, S(y)) = i_2(f_2(x, S(y))), & g_2(x, y), i_2(x, 1) = 3, \end{array}$$

where the functional variables f_2 and f_1 denote the functions F and F^* respectively and 2

nondeterminism can be found fully articulated in Turing's 1936 discussion of choice machines, and although that discussion is motivated by the very concerns for naturalness that made Turing's work so influential, the idea of a choice machine completely disappears for some twenty years. It is not until Rabin and Scott's 1959 article that the idea of a nondeterministic machine resurfaces. We can only speculate about the causes of this historical gap. As we indicated above, Turing's mention of the choice machine is brief, and he does a good job deflecting the reader from what little discussion there is. It is also possible that the choice machine was ignored as a theoretical oddity in the push to build machines that actually do something. Whatever the reasons, by 1959, both the theoretical and practical state of the art in computing had changed dramatically; Rabin and Scott's nondeterministic automata addressed concerns entirely different from those that motivated Turing.

By 1959, the Turing machine was "widely considered to be the abstract prototype of digital computers" (Rabin and Scott 1959, p. 114). Recall from Chapter 3 that, at roughly the same time, Rogers described the theory of recursive functions as an investigation into what might be accomplished by a digital computer working with "explicit deterministic programs of instructions" (Rogers 1969, p. 130). The question that had plagued Gödel, whether an adequate formal account of recursion was even possible, had lost its urgency by 1959. In fact, Rabin and Scott found themselves dealing with an altogether contrary worry that the Turing machine might be *too general* a model of computation. The ability to compute any recursive function was overkill for most

and 3 are abbreviations for $S(1)$ and $S(S(1))$ respectively.

practical applications. Hence Rabin and Scott focused their attention on the finite automaton as a more restricted model of computation.

A finite automaton is, essentially, a Turing machine without the tape.⁵ The automaton is finite in the sense of having only finitely many internal states, which must be used for both memory and control. In this way, the potentially infinite number of combinations possible in a Turing machine between state and input are avoided and, it was hoped, "a better approximation to the idea of a physical machine" would be achieved (Rabin and Scott 1959, p.114).⁶ But the theory of finite automata was also pursued for its own sake, and toward that end Rabin and Scott introduced nondeterminism as a possible, and they claimed, novel generalization of the finite automaton. Like the nondeterministic Turing machine, a nondeterministic automaton is described by a transition relation rather than a transition function; at some (perhaps every) step in its computation a nondeterministic automaton will assume one state among

⁵ Actually, Rabin and Scott originally imagined finite automata as "defining sets of tapes." The image was of an automaton scanning a finite, segmented input tape one square at a time, and upon reaching the end of the tape (after a single pass) either accepting or rejecting the input. The set of tapes thus accepted is the set of tapes "defined" by the automaton. These days we talk about the language accepted by a machine rather than the tapes defined by it, but the idea is the same. (In their formal exposition, the word *tape* is an abbreviation for *finite sequence of symbols*.) There is, however, a surprising result that hangs on the tape imagery. Rabin and Scott not only considered what could be decided after a single, one-way pass over an input tape (the so-called one-way automaton), but they also considered what could be decided by a machine allowed to run back and forth over its input (the so-called two-way automaton). It can be proved that two-way automata are no more powerful than one-way automata.

⁶ But not everyone thought this was a good idea; see, e.g., McCarthy (1962) who quips that approximating the finiteness of the IBM mainframe computer is hardly practical (or useful). Rabin and Scott are more sanguine: "An actual existing machine may have billions of such internal states, but the number is not important from the theoretical

a handful of possible next states. How, exactly, this happens is left to the reader: "We are not concerned with how the machine is built but with what it can do" (Rabin and Scott 1959, p.115). All that matters is that some sequence of state-transitions leads to an accepting state; if no such sequence is possible, the input is rejected. Moreover, as with Turing machines, it can be proved that the nondeterministic automata are no more powerful than deterministic automata; any language decided by a nondeterministic automaton can be decided by a deterministic automaton.

As before, we must ask, why bother with a generalization that proves to be no more general? Unlike Turing, who introduces the choice machine to buttress the intuitive appeal of his account, Rabin and Scott are motivated by purely pragmatic concerns. For them nondeterminism yields a useful "versatility," which can be "utilized for showing quickly that certain sets are definable by automata" (Rabin and Scott 1959, p.115). Their emphasis is on shorter proofs for well known results. Their proof of equivalence between deterministic and nondeterministic automata is likewise unrevealing from a philosophical perspective. While we can see something computational in the exhaustive, deterministic simulation of a nondeterministic Turing machine, such intuitions are harder to find in the workings of an equivalent deterministic automaton. Instead, there is a more mathematical (and less intuitive) construction that considers the power set of the automaton's set of states. Given a nondeterministic automaton the idea is to consider all the states that might be reached from a given combination of state and input. This set of states essentially becomes a label for a single new state in the constructed deterministic automaton. The

standpoint—only the fact that it is finite" (Rabin and Scott 1959, p.115).

nondeterministic transition from a given combination of state and input which leads to any one of a number of states now leads to a single state and thus the transition *relation* of the nondeterministic automaton becomes a transition *function* for a deterministic automaton.⁷ We might say that the deterministic automaton "keeps track in its finite control of all the states that the NFA [i.e., nondeterministic finite automaton] could be in after reading the same input" but it is not clear that the sense of simulation here is the same as that of a deterministic Turing machine simulating a nondeterministic machine. Indeed, in exactly the same way that we construct a universal machine, we can construct a single deterministic machine that takes as input the description of any nondeterministic machine and its input. Given this input, the deterministic machine is then able to unfold the computation tree the nondeterministic machine. The sense of simulation is thus quite general and is inherent in the workings of a single all-purpose machine. We might think of such a universal machine in the same way that we think of emulation software that allows a Mac to run *various* programs written for a PC. The PC software executes as if it were running on a Windows machine. Although there is an algorithm for constructing a deterministic automaton that accepts the language of a given nondeterministic automaton, there is no universal automaton in the sense we described above. The relationship between the given nondeterministic automaton and the deterministic is inherent in the

⁷ More formally, if Q is the set of states for the nondeterministic automaton, then 2^Q , the powerset of Q , is the set of states for the deterministic automaton. Likewise, if δ is the nondeterministic transition relation from $Q \times \Sigma$ to Q , then we define a deterministic transition function δ' from $2^Q \times \Sigma$ to 2^Q as follows,

$$\delta'([q_1, q_2, \dots, q_i], \sigma) = [q_1, q_2, \dots, q_k] \text{ iff } \delta(\{q_1, q_2, \dots, q_i\}, \sigma) \subseteq \{q_1, q_2, \dots, q_k\}.$$

particular construction and the resulting deterministic device cannot "simulate" any other automaton. Moreover, except for some tendentious labeling, the two automata are distinct; they accept the same language, but it is hard to see the workings of one in the behavior of the other.

Rabin and Scott's 1959 paper is consistently cited as having first introduced the nondeterministic automaton (cf. Greibach 1981, p. 18). It is thus tempting to assume that Rabin and Scott present the first general discussion of nondeterminism. But this is not the case; their discussion is neither first nor general. In fact, for Rabin and Scott nondeterminism is simply a convenient and conservative construction. Where Davis has described Turing's 1936 paper as "a remarkable piece of applied philosophy"—a characterization that applies particularly well to Turing's discussion of choice machines—Rabin and Scott's discussion of nondeterminism is more applied mathematics than applied philosophy (Davis, 1982, p. 14).

Although it was an influential paper, Rabin and Scott's work leaves little room for philosophical discussion. If there is a philosophical moral to be drawn at all, it is to recognize the conspicuous lack of philosophy in the theoretical work of the late 1950s—a theory that began some twenty years earlier amid pressing philosophical concerns. There was, however, another development in 1959 worth noting. As mentioned by Sipser (1992), and discussed more fully by Trakhtenbrot (1984), Russian theoreticians were worried about *perebor*, or brute-force algorithms. In particular, Yablonskii (1959) discussed problems that could be solved algorithmically in principle but which required prohibitive computational resources in practice.

At present there is an extensive field of problems in cybernetics where the existence of certain objects or facts may be established quite trivially and, within the limits of the classical definition of algorithms, completely effectively, yet a solution is, in practice, often impossible because of its cumbersome nature ... It is here that the necessity of making the classical definition of an algorithm more precise naturally arises. It is to be expected that this will, to a greater extent than at present, take into account the peculiarities of certain classes of problems, and may, possibly, lead to such developments in the concept of algorithm that different types of algorithms will not be comparable (Yablonskii (1959) p.401; also quoted in Sipser (1992)).

In many respects, the Russian notion of a *perebor* algorithm anticipates the class NP. Recall (Chapter 3) that problems in NP are those for which a putative solution can be discovered nondeterministically and verified in polynomial time. Just as Yablonskii describes *perebor* problems, problems in NP have trivial solutions in the sense that a machine could work them out, but the number of possible solutions that must be tried is vast. "Algorithms" for NP-problems run in polynomial time only because we imagine a nondeterministic search to take the place of an exhaustive search of an exponential solution space. In this sense, NP is, oddly enough, a way of classifying those problems for which we have *no* polynomial-time solution. The algorithms we *actually* have for such problems are *perebor* algorithms—inefficient, brute-force searches.

Yablonskii's work is remarkable for two reasons: First, from a historical point of view it is worth noting his anticipation of a class of problems that was not fully articulated in the West until 1972.⁸ Second, and more important from a philosophical

⁸ However, Yablonskii himself might not be too happy with the identification of problems solvable (only) by *perebor* with the complexity theorist's class of NP-problems. In fact, Trakhtenbrot (1984) reports that Yablonskii "distrusted the role that computational complexity and algorithm complexity could play in the *perebor* subject." Apparently, Yablonskii was something of a constructivist and he did not see how

perspective, is the idea that the classical notion of an algorithm might not adequately characterize the problems solvable by brute-force searches. If we allow ourselves to identify *perebor* searches with the "guessing" implicit in nondeterministic computation, Yablonskii's suggestion that we might be dealing with fundamentally different kinds of algorithm leaves us in an awkward position with respect to the proven equivalence between (unbounded) deterministic and nondeterministic Turing machines. How can incomparable notions of an algorithm be proven equivalent? Does such an equivalence really support the Church Turing thesis? These questions underscore the concerns raised in Chapter 3, only now our worries are more general. Questions about the equivalence of deterministic and nondeterministic computation need not be tied to a particular open problem in complexity theory, but extend to the very notion of a nondeterministic algorithm. We shall return to these questions in the next chapter.

3. 1959-1965 and Kuroda's nondeterministic linear-bounded automaton

Trakhtenbrot (1984) recalls that Russian work on complexity theory proceeded "independently and in parallel" to the work going on in the West in the early 1960s. The fact that such work was independent, together with the fact that the Russian theoretical community itself was bitterly divided over Yablonskii's "proof" that certain *perebor*

diagonalization—and, by extension, a good deal of complexity theory—could apply to the solution to combinatorial problems.

Yablonskii's mistrust of complexity theory notwithstanding, his assertion that *perebor* might be unavoidable resonates with our understanding of the P versus NP problem. In fact, when we address that problem we are asking whether some problems are inherently difficult (i.e., the exhaustive exponential search is unavoidable) or whether we have simply not yet discovered an efficient way of solving them. Yablonskii's assertion about *perebor* amounts to an assertion that $P \neq NP$.

algorithms were unavoidable, suggests that worries about incomparable algorithms might not have met with a wide audience outside the Soviet Union. What is more certain is that nowhere on the bibliographic trail to the landmark papers of Cook (1971) and Karp (1972) is there any mention of the *perebor conjecture*.

In the early-to-mid 1960s Western theoretical research took on a more pragmatic slant. Rabin and Scott's motivation "to give a better approximation to a physical machine" would find expression in many other influential papers.⁹ For example, Yamada stated, "In using digital computers, it is important to know the time required to compute a given function"(1962, p.754). As a start to such a theory, Yamada introduced the notion of *real-time computation* by way of a (restricted) Turing machine as one attempt at "a mathematical model for digital computers which is more realistic in particular aspects." Likewise, in their seminal 1965 work Hartmanis and Stearns chose a multi-tape Turing machine as their model of computation because "it closely resembles the operation of a present day computer"(p. 287). The time-bounded, *deterministic* Turing machine was the model of choice to bridge the gap between the theoretical and the practical. The intuitive notion of a time-step found a natural analog in the basic operations of a Turing machine (e.g., one transition between states = one unit of time), and the well-established

⁹ Obviously, a theory is a theory about something and, in the case of theoretical computer science, one would expect the work to apply to real computers. It does not follow, however, that the researchers of the early 1960s were simply paying lip-service to a theoretical cliché. Quite the contrary, complexity theory had yet to establish itself at the time and it was important to make the potential real-world payoff clear. Contrast that situation with the present situation where complexity theorists whole-heartedly admit that their theory is premised, in part, on *unrealistic* models of computation and often pursued for its own sake (cf. §2, Chapter 3).

robustness of the model itself provided a secure foundation for a general theory of computational complexity (i.e. a theory that is sufficiently invariant with respect to the model of computation). Cobham (1964) used the time-bounded Turing machine to point out the wide range of functions that can be computed in polynomial time and by 1965 the identification of the polynomial time algorithms with the class of tractable algorithms was made explicitly by Edmonds (1965).

It is not surprising that the nondeterministic Turing machine is conspicuously absent from the computer science of the early sixties. Given the theoretical desiderata for a model of real-world computers and a characterization of tractable algorithms, it would have made little sense to consider nondeterministic computation. A digital computer is characterized by a predicable flow of control, and from this point of view, a nondeterministic Turing machine is more apt to be seen as a model of a malfunctioning computer than a useful abstraction.

Still, there were theoretical applications somewhat removed from the computer science of the day where nondeterminism proved to be an interesting generalization. Recall that in 1959 Chomsky introduced his hierarchy relating formal grammars and automata. At each level of the hierarchy the question naturally arises whether allowing nondeterministic automata will affect the class of languages accepted. Rabin and Scott's 1959 equivalence proof showed that the class of languages accepted at the bottom of the hierarchy (i.e., the type-3 languages) is the same whether we consider deterministic or nondeterministic finite automata. It was also well known that the class of languages accepted at the top of the hierarchy (i.e., the type-0 languages) is unaffected if we allow

nondeterministic Turing machines rather than deterministic ones.¹⁰ In the middle of the hierarchy, however, determinism makes a difference. Although it was a routine matter to show that the class of languages accepted by deterministic pushdown automata is properly contained in the class of all context-free languages (i.e. the type-2 languages), Greibach recalls that in 1963 it was "an interesting open question whether [linear-bounded automata] corresponded to Type 1 grammars as [pushdown automata] to Type 2 grammars" (Greibach 1981, P.24).

In 1964 Kuroda introduced the nondeterministic linear-bounded automata—

¹⁰ Common knowledge or not, it is still difficult to find explicit references to an original equivalence proof. For instance, Kuroda (1964) makes an *apparent* reference to the equivalence between the unrestricted grammars (type-0) and Turing machines to which he refers to as "Theorem 0.3." He remarks, "It is easy to see that Landweber's proof of Theorem 0.3 does not depend on the determinacy of the automaton" (Kuroda 1964, p.209). Here it seems we have found a reference, albeit elliptical, to an equivalence proof. There is, however, no such proof in (Landweber 1963). I am certain Kuroda meant to refer to Landweber's proof that every language accepted by a LBA is context sensitive, which Kuroda had earlier labeled as "Theorem 0.4." It would be a completely trivial typo except for the fact that Kuroda goes on to say, "and Theorem 4 remains valid, if we understand, under our convention, the phrase 'linear-bounded automaton' as meaning 'nondeterministic linear-bounded automaton'." At first glance, it seems that Kuroda is talking about the determinacy of both Turing machines and LBA's—even if that isn't really the case.

There is room for a more subtle obfuscation in Greibach's citation of Evey's 1963 proof of the equivalence between (unbounded) deterministic and nondeterministic *acceptors*. I suppose this reference would be robust if we were to distinguish between Turing machines as *transducers* and Turing machines as *acceptors*. Strictly speaking, the choice machines Turing describes compute functions (where, e.g., $f(1)$ = the first theorem, $f(2)$ = the second theorem. etc.), and hence they are transducers. Evey, on the other hand, describes machines that accept or reject their input. In this light, there are two proofs to consider and perhaps credit is due to Turing for his transducers and to Evey for his acceptors. But it is just as easy to view acceptors as transducers that compute characteristic functions. So even if, strictly speaking, Turing and Evey describe different machines, it is hard to view their discussions as (conceptually) independent with respect to questions about nondeterminism *per se*.

essentially a nondeterministic Turing machine where the available work tape is strictly limited by the length of the input—and proved that every context-sensitive language is accepted by some nondeterministic linear-bounded automaton. This result, together with Landweber's proof that any language accepted by a linear-bounded automaton¹¹ is context-sensitive, establishes the equivalence between the Type-1 languages and nondeterministic linear-bounded automata. Kuroda's work is significant for several reasons. It completes the Chomsky hierarchy and, as mentioned in the last chapter, it marks a confluence of ideas; questions about computation, formal language theory and resource bounds can all be tied, both historically and conceptually, to questions about the nondeterministic linear-bounded automaton. From our present point of view, however, the most remarkable aspect of Kuroda's work is its seemingly contradictory legacy regarding nondeterminism. On one hand, Kuroda's work is celebrated as a first step toward a more thorough understanding of nondeterminism. For example, reflecting on the effort that went into establishing the correspondence between the Type-1 grammars and the linear-bounded automata Greibach reports,

¹¹ Notice there is no reference to the determinacy of the automaton. In fact, following Myhill's original discussion (1960), Landweber's LBA's were deterministic. But as Kuroda points out (cf. note 13), there is no loss of generality here. The main idea in Landweber's proof is to specify productions that mimic the behavior of an accepting automaton "in reverse." In other words, we construct a grammar that will generate strings representing complete configurations in an accepting sequence of transitions in an LBA starting with the final accepting configuration. We introduce nonterminal symbols to record the state and the tape alphabet, and the productions are defined in a straightforward way from the machine's transition table. With respect to the grammar, it makes no difference whether a string on the left side of the production leads to a unique string on the right.

With hindsight, it is hard to see why this was a difficult problem at all. Part of the reason was that we were not really used to nondeterministic machines—to thinking nondeterministically—and Myhill's paper defined only deterministic LBA's (Greibach 1981, p.24).

Presumably, Kuroda's work helped people "think nondeterministically." On the other hand, Kuroda posed two questions in 1964: first, are the context sensitive languages closed under complementation? and second, are nondeterministic linear-bounded automata more powerful than deterministic linear-bounded automata? Although the first question was, at long last, answered positively by Immerman (1988) (the second remains open), the problems proved to be so difficult in the intervening years that in 1974 Hartmanis and Hunt lamented "our inability to answer them indicates that we have not yet understood the nature of nondeterministic computation" (Hartmanis and Hunt 1973, p.3). Kuroda's 1964 work was a mixed blessing; whatever initial promise there was after the completion of the Chomsky hierarchy must have faded quickly when it became clear just how difficult it would be to think nondeterministically.

It is also strange that Kuroda (1964) figures so prominently in so many discussions of nondeterminism. This is not to deny Kuroda credit for introducing the nondeterministic linear-bounded automaton, nor do I mean to overlook the influence of the questions he posed on subsequent research, but when we look at what Kuroda actually said in 1964 we find that his discussion of nondeterminism is quite brief. Having devoted a full paragraph to the formal definition of a deterministic linear-bounded automaton, Kuroda needs only a single sentence to define the nondeterministic linear-bounded automaton as one where the transition function is multi-valued. There is also a

perfunctory remark that on a given input, nondeterministic computation can lead to both accepting and rejecting states.¹² But there is none of the exposition one might expect from a paper in which the central result depends on the introduction of a nondeterministic device. In fact, there is no mention of determinacy at all in Kuroda's proof. Instead his efforts are directed toward showing a normal form theorem for context-sensitive grammars in which no string of length greater than 2 appears in any of the productions and that such grammars are "length preserving" and "linear-bounded" in the sense that in any production, $\phi \rightarrow \psi$, not involving the start symbol, the lengths of ϕ and ψ are equal, and that if S is the start symbol then if $S \rightarrow EF$ then $E=S$. Kuroda claims, "It follows immediately from these lemmas that for any context-sensitive language there exists a linear-bounded automaton which generates it" (Kuroda 1964, p.214).¹³ Immediacy is in the eye of the beholder, but the emphasis of the proof is still clear: for each context sensitive grammar construct an equivalent grammar in which we can apply productions without exceeding the length of the derived sentence. Kuroda's emphasis stands in contrast to contemporary proofs of the same theorem where the emphasis is on the

¹² As expected, Kuroda says that input is accepted nondeterministically when there is a single accepting computation. Curiously, however, he says the same thing about rejecting: "a string is said to be rejected by M if there is a computation of M which, given the string as input, never ends, or ends up off the left end of the tape, or, finally, ends up off the right end of the tape in a nonfinal state" (Kuroda 1964, p. 209 emphasis added). This differs from the usual convention of accepting when there is a single accepting computation and rejecting only when *all* possible computations reject.

¹³ As before, we might be tempted to distinguish between LBA's as acceptors and LBA's as transducers, but I don't think it makes any difference here. There is no essential difference between an automaton that accepts nondeterministically and one that enumerates sentences nondeterministically. It is simply a matter of how we choose to decorate one and the same computation tree (e.g., w as input at the root vs. w as output at

nondeterminacy of the computation; we imagine a machine presented with an input, w , nondeterministically choosing productions and tape positions in an attempt to derive $S \Rightarrow w$. Since the productions are all non-contracting (recall that a grammar is context-sensitive if for each of its productions $\phi \rightarrow \psi$, ψ is at least as long as ϕ), we will never have an intermediate x , $S \Rightarrow x \Rightarrow w$, where the length of x outstrips the input length, and hence the machine will accept w iff $S \Rightarrow w$.

Kuroda needs both non-expanding derivations and nondeterministic automata to establish the equivalence between context-sensitive grammars and linear-bounded automata. It is strange that Kuroda's proof would emphasize the former while saying very little about the latter. Apparently it was enough to display length-preserving, linear-bounded grammars; the nondeterministic operation of the machine could be left to the reader. But such a presumption is especially odd considering that Kuroda's paper is considered somewhat of a landmark work on nondeterministic computation. If theoreticians had to learn to think nondeterministically, it is not clear to me that Kuroda's proof would have shown them how. In any case, the Chomsky hierarchy was complete and by 1965 nondeterminism had become a central feature of theoretical computer science.

4. 1965-1972

There is a disconnection between the received view of Kuroda's work and the work itself. Similar disconnections are evident in the period from 1965 to 1972. We begin with the work of Hartmanis and Stearns (1965), which initiated the theory of

a leaf).

computational complexity. The basic idea is straightforward: use the time it takes a Turing machine to compute a function as a measure of that function's intrinsic computational difficulty. At first blush, there is sense of continuity here; Hartmanis and Stearns present their theory as a natural extension of Turing's 1936 work, even going so far as to talk about computable *sequences* rather than functions. Like Yamada, and Rabin and Scott, Hartmanis and Stearns recognized the need to move away from the unrestricted Turing machine in order to model working digital computers.¹⁴ They also suggest that their work complements Myhill's discussion of a linear-bounded automaton as a space-bounded measure of computational complexity. And finally, there is a recursion-theoretic feel to many of the results they present (e.g., the set of all computable sequences is recursively enumerable, for any set of sequences time-bounded by a given function a diagonal procedure can be used to find a sequence not computable in that time-bound, the set of all complexity classes is countable and hierarchical, etc.). In this light, the 1965 work of Hartmanis and Stearns fits in nicely with the work that came

¹⁴ Unlike Rabin and Scott, Hartmanis and Stearns considered the Turing machine a perfectly appropriate model of computation:

This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines (Hartmanis and Stearns 1965, p. 285).

Although he reports being strongly influenced by automata theory, Hartmanis recalls that when he and Stearns started working on a theory of computational complexity, "we realized that finite automata did not provide us with a sufficiently rich model of computing to develop the quantitative theories that we believed were needed and could be created" (Hartmanis 1981, p.45).

before it. There is, however, an important difference. Hartmanis and Stearns are explicitly and exclusively concerned with deterministic computation.

In hindsight, it is remarkable that the seminal paper in complexity theory—a theory now dominated by questions about nondeterminism—does not mention the nondeterministic Turing machine. Hartmanis recalls that by 1962, when he and Stearns began a serious investigation of computational complexity, they had been "seriously exposed" to aspects of formal language theory and were well familiar with automata theory (Hartmanis 1981, pp. 45-46). In particular, Hartmanis had read Chomsky (1962) and Rabin (1959). Nondeterministic automata figure prominently in both papers. Hartmanis also recalls that he and Stearns had been able to do their work outside the traditional framework; they were "surprisingly ignorant" of the traditional theory of effective computability, and, unlike many of their peers, they were not driven to find *the* automaton that would model real computing" (Hartmanis 1981, p. 47 emphasis in the original). It would seem that Hartmanis and Stearns were not only sufficiently acquainted with the idea of a nondeterministic automaton, they were free of the recursion-theoretic scruples that might have caused them to avoid nondeterminism. They approached computational complexity as mathematicians interested in the most general theory, so they were not limited just to realistic generalizations (even if, following Yamada, they had placed a premium on developing a theory of actual digital computers), but at the same time they could remain blissfully ignorant of the recursion theorist's exclusive focus on deterministic computation. And yet there is no mention of a nondeterministic Turing machine—not as a possible generalization of computation (a subject they consider at

length), nor as a source of open questions for further study.

We can only speculate about the non-appearance of the nondeterministic Turing machine in 1965. In the introduction, Hartmanis and Stearns describe the paper's third section,

One section is devoted to an investigation as to how a change in the abstract machine model might affect the complexity classes. Some of these classes are related by a "square law," including the one-tape-multitape relationship: that is if σ is T -computable by a multitape Turing machine, then it is T^2 -computable by a single tape Turing machine. *It is gratifying, however, that some of the more obvious variations do not change the classes* (Hartmanis and Stearns 1965, emphasis added).

Although it seems that the generalization to a nondeterministic machine would have been obvious, the question as to how it might affect complexity classes would have been (and is still) far from obvious. Perhaps Hartmanis and Stearns anticipated the difficulty of this question and decided to omit the nondeterministic Turing machine from their work and spare themselves the headache. It seems more likely, however, that Hartmanis and Stearns were more than happy to settle on *a single* abstract model of *real* computing. The generalization to nondeterministic Turing machines has little real-world currency. On the other hand, the generalizations Hartmanis and Stearns do consider in the third section of their 1965 paper all concern tape arrangements (e.g., multiple tapes and two-dimensional tapes) that correspond to obvious variations in the architecture of real computers and their memory. The fact that these variations do not affect complexity classes supports Hartmanis and Stearns' claim that "all digital computers in a slightly idealized form belong to the class of multitape Turing machines" and vindicates the choice of the

deterministic Turing machine as an abstract model of time-bounded complexity.¹⁵

Quite apart from such speculation, it is strange that Hartmanis and Stearns did not consider nondeterministic machines—here we have another instance where the discussion of nondeterminism we might expect, *prima facie*, never materializes. Nondeterminism is one of the defining features of today's complexity theory and yet the nondeterministic Turing machine is conspicuously absent from Hartmanis and Stearns' seminal work. Looking at the example the other way around we find that the emphasis at the outset was to develop a theory of complexity that would apply to real-world computation; but now, with complexity theory dominated by questions about nondeterminism, the concern for realistic computation is all but lost. The fact that the theoretical emphasis has shifted so dramatically reinforces the doubts we expressed in the last chapter about reconciling nondeterminism and realistic computation.

Next we turn to Cook's 1971 work and find, once again, an odd mix of continuities and discontinuities. The paper is famous for establishing Cook's theorem—a constructive proof of the existence of an NP-complete problem. The sense of *complete* comes from recursion theory (and not logic) and it is important to the complexity theorists because

¹⁵ It is interesting that Hartmanis later collaborated with Hopcroft to write "An Overview of the Theory of Computational Complexity" (Hartmanis and Hopcroft 1971) in which the emphasis on the real-world application is reiterated more strongly than it was in 1965 despite the intervening work on nondeterministic computation: "Furthermore, this theory [i.e., computational complexity] *must* eventually reflect some aspects of real computing to justify its existence by contributing to the general development of computer science" (Hartmanis and Hopcroft 1971, p. 444, emphasis added).

We feel we have completely understood and categorized the complexity of a problem only if the problem is known to be complete for its complexity class... They [complete problems] are the link that keeps complexity classes alive and anchored in computational practice (Papadimitriou 1994, p.166).

Intuitively, a *complete problem* is at least as hard to solve as any other problem in the class and is thus thought to capture—or characterize—the difficulty of all the problems in the class. We establish completeness by way of *reduction* (another notion from recursion theory). Informally, one problem reduces to another when instances of the first problem can be transformed into instances of the second. A problem is complete for a given class when any problem in the class can be reduced to it. In Cook's case the target problem is deciding the set of propositional tautologies in disjunctive normal form. The basic idea is that we are given some language that is decided by a nondeterministic Turing machine in polynomial time, and we construct a formula of propositional logic, in conjunctive normal form, which will be satisfiable if and only if the machine accepts a given input within the given polynomial-time bound.¹⁶ If the machine does not accept the input, the constructed formula will be unsatisfiable, and hence the denial of the formula (which can be rendered in disjunctive normal form using De Morgan's laws in

¹⁶ There are two technical points to make here. First, as we saw in Chapter 2, constructing the propositional formula is straightforward (all the more so, since the polynomial bound allows us to use disjunctions and conjunctions in place of existential or universal quantification): it is a conjunction of several disjunctive subformulas asserting, e.g., that a particular input string appears on the input tape, that at each time-step the tape head scans exactly one cell, that each cell contains exactly one symbol, that at each time-step the machine is in exactly one internal state, that tape updates occur according to the machine's transition table, etc. Second, for any machine and any input the corresponding formula can be constructed in polynomial time. The second point is important because if we allow inefficient reductions, the sense of completeness becomes trivial. That is, very hard problems can be reduced to very easy problems if we allow

linear time) will be tautological. Thus, if we can decide propositional tautologies we can solve any problem that is decided by a nondeterministic Turing machine in polynomial time.

Cook's proof bears an obvious resemblance to Turing's proof for the insolubility of the *Entscheidungsproblem*; in both cases we have a reduction from the operation of a Turing machine to logic. Also, Cook's motivations to explore the scope and limits of mechanical-theorem proving are reminiscent of the questions that motivated Turing's original work. Finally, Cook puts a recursion-theoretic spin on his results. In this sense, Cook's 1971 work can be seen as continuing earlier work. By contrast, continuity after Cook, although often taken for granted, is not so clear.

It is not surprising that Cook is celebrated for initiating the study of NP-completeness and for formulating the P=NP question. When we look back at his 1971 work it is hard *not* to see modern theory inchoate in the results and proof techniques which are now so methodologically central to complexity theory. Trakhtenbrot (among others) reinforces this view when he points out that the significance of Cook's work was not fully appreciated until Karp (1972) demonstrated by way of example the wide extent of natural combinatorial problems that are NP-complete (Trakhtenbrot 1984, p. 396). This observation, together with the fact that Karp himself credits Cook's work as an inspiration, leads to the widely accepted view that the work started by Cook found its fullest expression in Karp. But hindsight can be misleading and in this case it leads to an anachronistic view of theory.

sufficiently complex reductions.

The fact that it was Karp's later work that popularized Cook's is far from an unprecedented situation in the history of science, but it is noteworthy here because it reminds us that Cook's work was *re-introduced* into a context that was slightly different from the one in which it was originally written. While it is tempting to assume that Karp (1972) simply extended Cook's work, the real story is slightly different. In fact, there are theoretical discrepancies between Cook and Karp (and for that matter, between Cook's 1971 work and the contemporary approach to NP-completeness). First, Cook and Karp rely on different senses of *reduction*. Cook defines *reduction* in terms of deterministic query machines (a.k.a. oracle machines):

A query machine is a multi-tape Turing machine with a distinguished tape called the query tape, and three distinguished states called the query state, yes state, and no state respectively. If M is a query machine and T is a set of strings, then a T-computation of M is a computation of M in which initially M is in the initial state and has an input string w on its input tape, and each time M assumes the query state there is a string u on the query tape, and the next state M assumes is the yes state if $u \in T$ and the no state if $u \notin T$...

A set S of strings is P-reducible (P for polynomial) to a set T of strings iff there is some query machine M and a polynomial $Q(n)$ such that for each input string w , the T-computation of M with input w halts within $Q(|w|)$ steps ($|w|$ is the length of w), and ends in an accepting state iff $w \in S$ (Cook 1971, p.151).

Karp, on the other hand, says nothing about oracle computation in his definition of *reduction*.

Let Π be the class of functions from Σ^* to Σ^* computable in polynomial time by one-tape Turing machines. Let L and M be languages. We say that L is reducible to M if there is a function $f \in \Pi$ such that $f(x) \in M \Leftrightarrow x \in L$ (Karp 1972, p.86).

So-called *Cook-reduction* is considered to be a weaker, more general notion than *Karp-*

reduction; however, *Karp-reduction* is more often used and is the *de facto* sense of reduction traditionally associated with the study of NP-completeness. Second, Karp's presentation of Cook's theorem has SATISFIABILITY as the target problem in the reduction and not that of deciding DNF tautologies. While SATISFIABILITY is the canonical example of a problem in NP, deciding tautologies is a typical example of problems in coNP. In fact, given that the problem of deciding tautologies is coNP-complete, which, together with the current thinking that $NP \neq coNP$, suggests that Cook's example of an NP-complete problem is not really in NP (recall note 3 from the previous chapter).¹⁷ Finally, it is somewhat misleading to attribute the explicit formulation of the

¹⁷ On one hand, it is hardly surprising that Cook used DNF tautologies as his target problem; his interests were in theorem-proving, where validity is a far more important property than satisfiability. On the other hand, however, it is remarkable that the main result in Cook's 1971 paper is so often misremembered as an example of an NP-complete problem. This situation can be explained by pointing out again that Cook's work was not really appreciated until 1972, when Karp directed attention toward *complete* problems. Thus Cook's original emphasis on theorem-proving and his reduction to DNF tautologies seem to have been overlooked (cf. Miller 1972). Unfortunately, we won't find much philosophical comfort in this explanation. Quite to the contrary, it is strange that Karp's work should be seen as an extension of Cook's when Karp lays the foundation for distinguishing the two approaches. Cook's target problem is decided by an oracle machine. Consequently, it makes no difference whether we consider the problem itself or the complement problem. For example, Cook notes that, like other problems, the problem of deciding whether a number is prime can be reduced to the problem of deciding DNF tautologies because it "*or its complement, is accepted by a nondeterministic Turing machine*" (Cook 1971, p.152 emphasis added). Although it turns out that a nondeterministic machine can decide whether a number is prime (in fact, the problem of deciding primality is in both NP and coNP), that result wasn't established until 1975, which suggests that Cook must have been thinking about the complementary problem of deciding whether a number is composite when he added primality to his list of P-reducible problems—a problem that is easily decided by a nondeterministic polynomial-time algorithm (guess and verify factors). By contrast, Karp relies on nondeterministic machines to decide the target problem and in this context it is widely believed that there is a difference between deciding a problem or its complement. And exactly this

P=NP question to Cook (cf. Greibach 1981; Sipser 1992; Trakhtenbrot 1984). Although Cook does suggest that it is "fruitless" to search for certain polynomial time decision procedures, there is no mention of NP *per se* and he is remarkably sanguine when it comes to the prospects of proving that DNF tautologies cannot be decided in polynomial-time: "I feel it is worth spending considerable effort trying to prove this conjecture" (Cook 1971, p.154).¹⁸ While the $P \neq NP$ conjecture is certainly implicit in Cook's 1971 work, the main thrust of that paper is toward establishing various complexity measures and in this sense formulating the larger question whether $P=NP$ almost seems incidental. Karp, on the other hand, speaks explicitly in terms of P and NP and his extensive list of "classic" problems that turn out to be NP-complete makes it obvious that the $P=NP$ question is more than incidental. In addition, at the time his work was presented, Karp reported a preoccupation with the question of equivalence between polynomial-time deterministic and nondeterministic Turing machines (Miller 1972, p.177). The $P=NP$ question as we know it is far more evident in Karp's 1972 work than it is in Cook's 1971 work.

While it might seem that we are splitting theoretical hairs, there is an important philosophical point to be made here. Cook (1971) and Karp (1972) differ in their sense of reduction, their statement of Cook's theorem and in overall emphasis, and yet the two papers are regarded as continuous efforts. Indeed, one can hardly read about Cook's 1971

difference divides Cook and Karp.

¹⁸ Although Cook does note, despairingly, that the kind of diagonalization which establishes the halting problem (a complete problem in its own right) as non-recursive does not seem to carry over to a proof that DNF tautologies cannot be decided in poly-

paper without also reading about Karp's 1972 paper. How does this happen? How does a theoretical tradition that is driven by a variety of subtle distinctions seemingly overlook these more obvious discrepancies at its foundations? We might answer that not all differences are irreconcilable and in this case the differences we have noted are not differences in kind. From a theoretical point of view this is a perfectly reasonable answer. But from a more philosophical perspective we find that the theoretical answer flies only because there is flexibility in the conceptual underpinnings of nondeterminism—so much flexibility that our intuitions about nondeterminism are collapsed to suit the theory.

The difference between Cook-reductions and Karp-reductions is a case in point. There was never any theoretical question about the distinction between the two notions; recursion theorists had studied analogous reducibilities long before 1972, Karp himself acknowledged Cook-reduction as a weaker notion, and by 1975 the differences among resource-bound reductions were well established.¹⁹ At the same time, the distinction between Cook-reductions and Karp-reductions also relates the two notions. Cook-reductions are described as those in which a polynomial number of calls to an oracle are allowed while Karp-reductions are described as those in which exactly one call to the oracle is allowed. Oddly enough, however, Karp never mentions oracles in his formal definition of reduction. Instead, he defines nondeterministic algorithms in very general terms and discusses how one might think about a nondeterministic machine "guessing" or pursuing parallel computation paths (Karp 1972, pp. 91-92). Moreover, Karp notes:

time (Cook 1971, p.155).

¹⁹ See, e.g. (Ladner, Lynch, and Selman 1975).

The reader will not go wrong by identifying P with the class of languages recognizable by digital computers (with unbounded backup storage) which operate in polynomial time *and Π with the class of string mappings performed in polynomial time by such computers* (Karp 1972, p.88, emphasis added).

Contrast that view with Cook's informal discussion of reduction, where he describes an oracle that "knows" the target problem and can decide membership questions "instantly." On an intuitive level, it is far from obvious that the nondeterministic machine at the end of a Karp-reduction is doing the same thing as the oracle machine in a Cook-reduction—and yet, theoretically, Cook-reductions generalize Karp-reductions.

The theoretical reconciliation between Cook-reductions and Karp-reductions demands that we think about nondeterministic machines as oracle machines. Not only does this blur the intuitive contrast between Cook (1971) and Karp (1972), but it also obscures an important historical difference between oracle and nondeterministic computation. As mentioned in the last chapter, the introduction of the oracle machine is due to Turing. There is no question about the when or why for oracle computation. In a 1939 paper Turing explicitly defines oracle machines and then uses them to prove that there would still be unsolvable problems even if certain problems could be solved by some "unspecified means" (Turing 1965b). According to Feferman (1988), Turing's idea was "striking" and even if Turing himself did nothing else with oracle machines, the idea resurfaced in Post's work in the '40s, with due credit to Turing, and would eventually "change the face of recursion theory." While the nondeterministic Turing machine has had a comparable impact on complexity theory, we have seen that its pedigree is far less

certain.

For such a ground breaking paper, Cook (1971) is remarkably conservative; its motivation and proof-techniques recall Turing's original efforts, while the notion of an oracle machine has its roots in a well-defined theoretical tradition. Karp (1972), on the other hand, sets the stage for the contemporary discussion of NP-completeness. In this sense the two papers are discontinuous; Cook's paper is something of a throwback while Karp's anticipates future work. Even if one were to deny that the transition from Cook to Karp is so sharply kinked, it should still be clear that a presumption of continuity is unfounded. At the very least, we have noted a peculiar theoretical ellipsis (cf. note 19) and what seems to be the conflation of oracle and nondeterministic computation. Once again, we see that the received view of nondeterminism is in need of both historical and philosophical work.

5. Conclusions

It should be clear by now that if we are hoping for philosophical understanding, the events from 1936 to 1972 reveal little continuity in the conceptual development of nondeterminism. First, there was Turing's discussion of choice machines. Although it seems to have been overlooked, Turing's is clearly the first formal discussion of nondeterminism. Moreover, Turing's motivation for proving the equivalence between (unbounded) deterministic and nondeterministic machines anticipates contemporary appeals to the equivalence as evidence for the Church Turing thesis. Next, there was Rabin and Scott's nondeterministic finite automaton. Unlike Turing, Rabin and Scott's motivation was far more pragmatic than philosophical; for them the (re)introduction of

nondeterminism was a conservative expedient on the way toward a more realistic theory of working computers. Then there was Kuroda's work, which began with a problem from formal language theory and produced a result that is now celebrated for bringing together a broad community of theoretical computer scientists and teaching them how to think nondeterministically. But as we saw, Kuroda is surprisingly diffident when it comes to explaining how his nondeterministic linear-bounded automata works. Finally, there is the work of Cook and Karp, which is celebrated for giving us our first true insight into the class of problems solvable by polynomial-time bounded nondeterministic Turing machines. Unfortunately, this insight blinds us to theoretical and historical tensions. From a technical point of view, the idea of a nondeterministic automaton remains constant across these four episodes—we simply allow transition relations rather than transition functions—but from a conceptual point of view, we find the same technical device being applied in very different contexts. No single motivation unifies these various discussions of nondeterminism.

V. Computer Science and the Philosophy of Science

1. Looking Back and Looking Forward

It is time, finally, to take stock of our efforts. We began with general questions about the Turing machine. Given its central place in theory, doubts about the Turing machine cut straight to the core of theoretical computer science. But rather than initiate a complete conceptual overhaul, we opted to take the theory on its own terms in an attempt to make sense of nondeterministic algorithms. After examining some of the more mundane results from complexity theory we found that nondeterministic algorithms lack many of the qualities we intuitively associate with algorithms and we argued that the prospects for a philosophical reconciliation between ideas about resource bounds, nondeterminism and the informal notion of algorithm seem rather grim. Next, we tried to fit these ideas into a tidy historical context; but despite our best efforts, the history of nondeterminism is still a mess. Although we have pinned down a definite date for the introduction of the nondeterministic Turing machine, it is hard to find much continuity in the landmark papers that followed. In fact, apart from a few anecdotal remarks, very little has been said about how theoreticians *thought* about nondeterminism. The history of nondeterminism is characterized by disparate motivations and theoretical ellipses. Even worse, it is not clear that even this early in the history of theoretical computer science we will be able to fill in the gaps.¹

¹ Cf. Davis (1988a) where he despairs at the prospect of a robust intellectual history of theoretical computer science. His example of the inherent difficulties of such a history

On one hand, these results are disappointing. We have done all this work merely to conclude that the received view of complexity theory is in need of both historical and conceptual work—hardly a constructive conclusion. On the other hand, however, we have uncovered a deep tension at the heart of theoretical computer science. This by itself is progress. But more important, in coming to grips with this tension we become better philosophers. Questions about theoretical computer science force us to clean up loose rhetoric and might even shed new light on traditional positions in the philosophy of science.

First, let us reiterate, explicitly, the lurking philosophical tension: given that history fails to reveal a univocal motivation for nondeterminism, we must face the possibility that the notion has been put to mutually exclusive theoretical ends. In some cases nondeterminism is marshalled as evidence in support of the Church Turing thesis, while in other cases it strongly suggests that our formal notion of an algorithm might be incomplete. For example, we noted at the beginning of Chapter 3 that Rogers takes it as philosophically obvious that the informal notion of an algorithm is deterministic insofar as it entails ideas about mechanism and discrete step-wise operation. But he also goes on to argue that the informal notion of an algorithm need not be constrained by any bound on how long the computation takes. Although Rogers himself does not consider the possibility, it is clear in this context how the equivalence between unbounded deterministic and nondeterministic Turing machines might further buttress what Rogers calls the *Basic Result*—the fact that a wide variety of formal characterizations of

reflect many of the problems we have encountered.

computable pick out the same class of functions. It makes no difference whether our formal notion of algorithm is deterministic or nondeterministic; exactly the same class of functions ends up being computable. The equivalence gives us more reason to believe that we have found in the Turing machine an adequate formal analogue for our informal intuitions about algorithms. At the same time, however, Rogers concedes that there is room to debate whether the informal notion of an algorithm entails a constraint on the time of computation. The nod to the theory of complexity is obvious, but there is far more at stake here than a simple acknowledgement. As soon as we impose resource bounds, the equivalence between deterministic and nondeterministic Turing machines goes up for grabs. Indeed, the fact that there is wide-spread belief that $P \neq NP$, together with the idiosyncratic features we noted in Chapter 3, suggests that nondeterministic algorithms are fundamentally different from deterministic ones. Either we can ignore resource bounds and celebrate the equivalence between deterministic and nondeterministic Turing machines as evidence for the Church Turing thesis, or we can impose resource bounds and forget about the equivalence. As we said from the outset, we cannot have it both ways.

Returning to the questions we raised in Chapter 4, the idea of a nondeterministic Turing machine might split the classical sense of algorithm into incomparable notions. Such a split would make it easier to account for the fact that in one context we have an assumption that is theoretically conservative, while in another context the same assumption leads to a variety of (putative) distinctions. It might also allow us to explain away the counter-intuitive features of nondeterministic algorithms—for it is hardly

surprising that a fundamentally different sense of algorithm would rest on intuitions far removed from those that underwrite the classical sense of algorithm. Still, if there are really incomparable senses of algorithm here, we must radically readjust our view of the ongoing work in complexity theory. Theoretical computer science presupposes a robust notion of algorithm as its object of study, and it is natural to assume that ongoing work reflects a deeper understanding of that notion. But if complexity theory rests on a sense of algorithm that undermines or, perhaps, even supplants the classical sense, then we must question how the contemporary theory relates to the work that came before it. In this light, we might come to regard the persistent mention of Turing machines and the oft-cited analogies to recursion theory as historical curiosities in the development of complexity theory, but it would be hard to justify conceptual ties to much of the previous work if it turns on a fundamentally different sense of algorithm. It is one thing to refine theoretical foundations but quite another to replace those foundations altogether.

In the end, these questions about determinacy and resource bounds betray deeper questions about what it is to be algorithmic. Although Church, Turing and Kleene (among others) have been celebrated for providing a definitive answer to that question, they only opened the subject; they did not close it. We have explored just one avenue of investigation and discovered that nondeterminism has motivated a variety of haphazard theoretical developments and might ultimately reflect a splintering sense of algorithm. At the very least, it should be clear that the complexity theory of today is dealing with a sense of algorithm far removed from that put forward in the '30s.

Even if we cannot immediately make sense of these developments, it is important

that we take notice of them nonetheless. For one reason, the language of computation is now entering into philosophical debate. Unfortunately, the usage tends to be uncritical and consequently a good number of philosophical debates generate heat and smoke but very little light. For example, consider Searle's (1990) discussion of "multiple realizability." Computers are said to be multiply realizable in the sense that they are characterized by the manner of their construction and not the materials; if it is built in the right way a machine constructed from water pipes and plumbing valves will be just as much a computer as a machine built from MSI chips and printed circuit boards. From this view a machine need only display the right kind of functional relationship between "input" and "output" to be a computer. Although multiple realizability reflects a remarkable separation of form and function, Searle worries if such a view commits us to a view that *any* physical object is a computer at some level of description; he even goes so far as to suggest that, "the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements which is isomorphic with the formal structure of Wordstar" (Searle 1990, p. 27).

Searle raises an interesting question: if computation cannot be characterized by the stuff of which computers are made, what makes something computational? Moreover, if we cannot say what it is to be computational, what sense is there in asserting a computational theory of mind? One response comes from Copeland, who claims "To compute is to execute an algorithm" (1996, p. 335). The idea is straightforward: we call something a computer when we can specify (in advance) a relation between an underlying architecture (whatever the physical implementation might be) and a step-by-

step flow of control. Presumably, we cannot even begin to describe how the movement of molecules relates to what Searle glibly identifies as "the formal structure of Wordstar" and so we do not regard the wall behind his back as a computer.

While this exchange between Searle and Copeland is engaging, it also strikes me as ill-framed. To be fair to both Searle and Copeland, we have caricatured a debate about the computational theory of mind, which is set in a much wider context in the philosophy of mind. At the same time, however, the philosophy of mind is one place where the unbounded sense of algorithm is clearly inappropriate. Indeed, if there is philosophical agreement about anything in that context it is that brains are finite. So, if mind is the brain's execution of an algorithm, then the sense of algorithm had better be both resource-bounded and tractable (in some sense). Moreover, the possibility that modern complexity theory might reveal a splintered sense of algorithm should weigh heavily on this debate. The homely sense of algorithm we associate with Turing's work—intuitions both Searle and Copeland exploit—might be too coarse a notion or, perhaps, the wrong notion altogether for a discussion of algorithms of such manifest complexity. This is not to suggest that complexity theory will settle the debate between Searle and Copeland, but rather to point out that before we can even engage the debate we had better look at complexity theory and decide whether we have framed the debate in the appropriate terms.

Obviously, a philosophical appeal to theoretical notions should be informed and critical, but there is also another reason that philosophers, and philosophers of science in particular, should pay attention to the recent developments in complexity theory.

Consider what Hartmanis has to say about theoretical computer science as science:

I see computer science as a brand new species among other sciences, and I believe it differs fundamentally from the older sciences. As a matter of fact, I am convinced that in large parts of computer science the classic research paradigms from the physical sciences or mathematics do not apply and that we have to develop and understand the new paradigms for computer science research. The fundamental difference between, say, physics and computer science is that in physics we study to a very large extent a world that *exists*, and our main objective is to observe and explain the existing (and predict new observable) phenomena. The relations between experiments and theory are quite well understood and richly illustrated by successful examples. Computer science, on the other hand, is primarily interested in what *can exist* and how to describe and analyze the possible in information processing. It is a science that has to conceptualize and create the intellectual tools and theories to help us imagine, analyze, and build the feasibly possible (Hartmanis 1981, p.43).

Are we really witnessing the birth of a new science? There are several reasons to think we are. For instance, Hartmanis goes on to say that understanding the relation between theory and practice will be important for making sense of computer science as a new science. The role of nondeterminism is certainly one example where the traditional views of theory and practice break down. Here we have a notion that, from a practical point of view, is virtually inscrutable. As we have said before, nondeterminism is something of an anathema to digital design; real computers do not guess what to do next. And yet nondeterminism is a central feature of theoretical computer science. Contrast the state of affairs we just described with the situation in the "other sciences;" although theorists and experimentalists are engaged in different activities, the objects of study presumably remain constant. One might say that the study of computation unites the theorist and the engineer, but at best, this connection is degenerate insofar as theory mostly provides hardware engineers with examples of what they *cannot* hope to do, and at worst it seems

that theory and practice could disconnect altogether. Consider again Pitowski's suggestion that there might be NP-complete problems with physical polynomial time solutions even if it happens theoretically that $P \neq NP$. If it is possible to exploit physical analogues to provide tractable solutions to theoretically "hard" problems, will such solutions be computational? For better or worse, the theoretical sense of the word is rooted in the plodding, discrete behavior of a Turing machine, and it is far from clear that the same intuitions (much less the theoretical results) would apply to an analog "computer." The possibility of what *can exist* in this case forces the philosopher of science to rethink the relation between theory and practice. It is hard enough to see the theoretical payoff even when we presume a constant sense of *computational*.² I am not sure how we would explain the situation if theory and practice ultimately revolved around fundamentally different conceptual foundations.

It is hard to sort out the relation between theory and practice because it is hard to

² A sociologist would have field day studying the advent of computer science as an academic profession. The struggle for disciplinary identity was hard fought and the division between those who view computer science as applied mathematics and those who view it as electrical engineering exists to this day. So, it comes as no surprise when those in opposing camps cast a skeptical eye on each other's work. But it is surprising when the theorist expresses doubts about the value of his own work. We have already mentioned Hartmanis' worry that too much of the theory is hidden "behind obscure mathematical formalizations" and that "[t]ime and again, we have valued the difficulty of proofs over the insights the proved results give us about computing; we have been hypnotized by mathematical elegance and pursued abstraction for its own sake" (Hartmanis 1981, pp. 49-50). Even more telling is the fact that such worries have plagued the theory from the beginning. At the 1972 conference on the Complexity of Computer Computations, a distinguished panel of theorists (Karp, Rabin, Hopcroft, to name a few in attendance) was asked, point blank, to discuss "[w]hat specific examples have been found to demonstrate how real computers computations were improved from studies of this type" (Miller 1972, p.170). It is striking that such a question would come up at all,

sort out the theory itself. Here again, the role of nondeterminism forces us to rethink traditional views. Hartmanis suggests there are no natural kinds for the computer scientists to study. There is no independently existing species *algorithm* against which we can compare our theoretical results. There are only intuitions and the formal analogues we introduce. We have mentioned the sort of evidence given to support the claim that the unbounded Turing machine captures all our intuitions about algorithms. That evidence trades on a combination of unexpected convergence and intuitive appeal—it is the kind of theoretical evidence easily recognized by the philosopher of science. But we have also argued contrary to the received view that ideas about resource bounds and nondeterminism do not fit very easily into this body of evidence. In fact, we have seen that the idea of a resource-bounded nondeterministic algorithm actually contradicts many of our intuitions about algorithms. So, if it is not evidence (at least not in any usual sense of the word), what exactly is the theoretical status of nondeterminism? It is tempting to characterize nondeterminism as an unrealistic but useful model of computation. Such a view resonates with the notion of idealization—another notion familiar to the philosopher of science. But if nondeterminism is an idealization, it is a peculiar kind. In general, idealizations allow us to impose a simpler, albeit unrealistic structure on the flux of phenomena we observe. An idealization is useful insofar as it makes matters more perspicuous. It is odd to think that we might do the same to the "conceptualizations" and "intellectual tools" of our own construction. Can we idealize our own ideas?

Some will argue that we have simply misconstrued complexity theory. Perhaps it

much less at a conference devoted to theory.

is better understood as theory concerned with "the intricate and exquisite interplay between *computation* (complexity classes) and *applications* (that is, problems)" (Papadimitriou 1994, p. v). The problems themselves—as opposed to the algorithmic solution to those problems—become the objects of study. Nondeterminism is neither evidence nor idealization; rather, it is a useful abstraction, a generalization that leads to a rich classification of problems. The emphasis here is decidedly mathematical and it avoids what otherwise seems like a procrustean fit between complexity theory and our traditional views of scientific theory. At the same time, however, it is hard to understand nondeterminism even as part of an ongoing mathematical investigation. In many ways, the mathematical development of complexity theory reminds me of the development of set theory near the turn of the century. Once again, we find that a seemingly innocuous generalization can take a theory originally motivated by familiar and robust intuitions and turn it on its head. Although nondeterminism does not lead to outright paradox, it does lead to a number of notoriously open questions. Just as the intuitionist worried about the path to Cantor's paradise, we do well to ask about the assumption that got us here.

There are hints of a Platonic attitude toward nondeterminism. Time and again theorists speak of the "nature of nondeterministic computation" and our "limited understanding of it." While such an attitude might be understandable in the face of so many open problems, it is still odd that it would find expression in a discipline so firmly rooted in the constructivist tradition of finite combinatorial mathematics. There are also more pragmatic attitudes toward nondeterminism. Realistic or not, it induces a rich theoretical structure and allows us to draw analogies to the more established branches of

mathematics (e.g., recursion theory, model theory etc). Such an attitude is reminiscent of Russell's defense of the "logical" axioms of *Principia Mathematica*; it is a sort of wait-and-see approach. (Take the axiom now and decide later if you like where you end up theoretically.) Unfortunately, even the most productive theorist must admit that a good part of his work is tentative. At present it is very hard to judge the theoretical utility of nondeterminism. In fact, as we suggested in Chapter 3, the assumption has led to a sort of theoretical regress, as researchers introduce ever more remote notions to prove that $P \neq NP$, while some have even begun to pursue independence proofs that neither the conjecture nor its negation is provable—not exactly the kinds of results one would like to see when judging the theoretical utility of nondeterministic Turing machines. On the other hand, it is hard to ignore the feeling that we actually have discovered something robust and natural when we point to the unbounded equivalence between deterministic and nondeterministic machines as evidence for the Church Turing thesis, and we cannot deny that nondeterminism leads to an interesting, albeit tentative, theory. It is thus difficult to categorize the study of nondeterminism along the lines philosophers have traditionally imposed upon mathematical investigation; it is not entirely Platonic, nor formal, nor constructive. But it is an investigation of something, and when we understand what that something is we will, I think, have a finer-grained philosophy of mathematics.

It is fitting that we should conclude with a nod toward the philosophy of mathematics. Sorting out the conceptual tensions here is really a problem for the philosopher of mathematics. Complexity theory began as a sort of applied mathematics, but now the state of the art is mathematical through and through. Intuitions about

machines and the solutions to specific combinatorial problems have given way to variety of inter-theoretic reductions between logic, model theory and complexity theory. What we really want to know is how a theory that began with a concern for concrete machines and real life algorithms can have so much to say about a highly abstract tradition in mathematical logic, and vice versa. If we could get a better grip on these mathematical motivations and connections, we might be able to address our concerns about nondeterministic algorithms.

We might never have an answer to the $P=NP$ question, and the nondeterministic Turing machine might be forever lost in theoretical limbo. It might also be impossible to give a philosophically satisfying account of development of complexity theory. Nevertheless, it is doubly important that philosophers of science try to make sense of the role of nondeterminism in complexity theory. Not only do we afford ourselves the opportunity to get in on the ground floor as a nascent science sorts out the relation between theory and practice, we might also find a middle ground on some long standing questions from the philosophy of mathematics. There is work to be done, and it is worth doing.

Bibliography

- Bovet, Daniel Pieere and Pierluigi Crescenzi. 1994. *Introduction to the Theory of Complexity*. New York: Prentice Hall.
- Chandra, A. K. and L. J. Stockmeyer. 1976. Alternation. In *FOCS*, 17.
- Chomsky, Noam. 1959. On certain formal properties of grammars. *Information and Control* 2: 137-167.
- Chomsky, N. 1962. Context-free grammars and pushdown storage. *MIT Quarterly Progress Report* 65: 187-194.
- Church, Alonzo. 1965. An Unsolvable Problem of Elementary Number Theory. In *The Undecidable*, ed. Martin Davis. Hewlett: Raven Press.
- Cobham, A. 1964. Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science, ed. Y. Bar-Hillel:24-30: North-Holland.
- Cook, S A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the ACM Symposium*:151-158. Shaker Heights.
- Copeland, Jack B. 1996. What is Computation. *Synthese* 108, no. 3: 335-359.
- Davis, Martin. 1958. *Computability and Unsolvability*. New York: McGraw-Hill Book Company.
- Davis, Martin. 1965. *The Undecidable*. New York: Raven Press.
- Davis, Martin. 1982. Why Gödel didn't have Church's Thesis. *Information and Control* 54: 3-24.
- Davis, Martin. 1988a. Influences of Mathematical Logic on Computer Science. In *The Universal Turing Machine A Half-Century Survey*, ed. Rolf Herken. Oxford: Oxford University Press.
- Davis, Martin. 1988b. Mathematical Logic and the Origin of Modern Computers. In *The Universal Turing Machine A Half-Century Survey*, ed. Rolf Herken. Oxford: Oxford University Press.

Edmonds, J. 1965. Paths, trees, and flowers. *Canadian Journal of Mathematics* 17: 449-467.

Evey, James R. 1963. The Theory and Applications of Pushdown Store Machines. Doctoral, Harvard University.

Feferman, Solomon. 1988. Turing in the Land of $O(z)$. In *The Universal Turing Machine A Half-Century Survey*, ed. Rolf Herken. Oxford: Oxford University Press.

Gödel, Kurt. 1965a. On Undecidable Propositions of Formal Mathematical Systems. In *The Undecidable*, ed. Martin Davis. New York: Raven Press.

Gödel, Kurt. 1965b. Remarks Before the Princeton Bicentennial Conference on Problems in Mathematics. In *The Undecidable*, ed. Martin Davis. Hewlett: Raven Press.

Greibach, S A. 1981. Formal Languages: Origins and Directions. *Annals of the History of Computing* 3, no. 1: 14-41.

Hartmanis, Juris. 1981. Observations About the Development of Theoretical Computer Science. *Annals of the History of Computing* 3, no. 1: 42-51.

Hartmanis, J and J E Hopcroft. 1971. An overview of the theory of computational complexity. *Journal of the Association for Computing Machinery* 18: 444-475.

Hartmanis, J. and H. B. Hunt. 1973. The LBA problem And Its Importance In The Theory Of Computing. In *Proceedings of a Symposium in Applied Mathematics of the A.M.S. and S.I.A.M.*, ed. Richard M. Karp, 7. New York City.

Hartmanis, J and R E Stearns. 1965. On the computational complexity of algorithms. *Transactions of the American Mathematical Society* 117: 285-306.

Hilbert, David and W Ackermann. 1950. *Principles of Mathematical Logic*. Translated by Hammond, L .M. Leckie, G. G. Steinhardt, F. New York: Chelsea Publishing Company.

Hodges, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.

Hopcroft, John E. and Jeffrey D. Ullman. 1969. *Formal Languages and Their Relation to Automata*. Reading: Addison-Wesley.

Hopcroft, John E and Jeffrey D Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Reading: Addison-Wesley Publishing Company.

- Immerman, N. 1988. Nondeterministic space is closed under complementation. *SIAM Journal on Computing* 17: 935-938.
- Kannan, R. 1981. Towards separating nondeterministic time from deterministic time. In *FOCS*, 22:235-243. Nashville.
- Karp, R M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, ed. R . E. Miller and J. W. Thatcher:85-103: Plenum Press.
- Kleene, Stephen C. 1952. *Introduction to Metamathematics*. Edited by M H Stone, O Zariski, S S Chern, and L Nirenberg. The University Series in Higher Mathematics. Princeton: D. Van Nostrand Company, Inc.
- Kleene, S C. 1981. Origins of Recursive Function Theory. *Annals of the History of Computing* 3, no. 1: 52-67.
- Kleene, Stephen C. 1988. Turing's Analysis of Computability, and Major Applications of It. In *The Universal Turing Machine A Half-Century Survey*, ed. Rolf Herken. Oxford: Oxford University Press.
- Kuroda, S. Y. 1964. Classes of languages and linear-bounded automata. *Information and Control* 7: 207-223.
- Ladner, R. E., N. A. Lynch, and A. L. Selman. 1975. A comparison of polynomial time reducibilities. *Theoretical Computer Science* 1: 103-123.
- Landweber, P. S. 1963. Three theorems on phrase structure grammars of type 1. *Information and Control* 6: 131-137.
- McCarthy, J. 1962. Towards a Mathematical Science of Computation. In *Information Processing 1962*, ed. Cicely M Popplewell:21-28. Amsterdam: North-Holland Publishing Company.
- Miller, Raymond E. 1972. Panel Discussion. In *Complexity of Computer Computations*, ed. Raymond E. Miller and James W. Thatcher. New York: Plenum Press.
- Minsky, Marvin L. 1967. *Computation: Finite and Infinite Machines*. Englewood Cliffs: Prentice Hall.
- Myhill, J. 1960. Linear Bounded automata. *WADD Tech. Note No. 60-165* .

- Papadimitriou, Christos H. 1994. *Computational Complexity*. Reading: Addison-Wesley Publishing Company.
- Paul, W. J., N. Pippenger, E. Szemredi, and W. T. Trotter. 1983. On determinism versus non-determinism and related problems. In *FOCS*, 24:429-438. Tucson.
- Pitowsky, I. 1990. The Physical Church Thesis and Physical Computational Complexity. *Iyyun* 39: 81-99.
- Post, Emil. 1965. Recursive Unsolvability of a Problem of Thue. In *The Undecidable*, ed. Martin Davis. New York: Raven Press.
- Prosser, Franklin P and David E Winkel. 1996. *The Art of Digital Design*. Englewood Cliffs: Prentice-Hall.
- Rabin, M O and D Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research* 3, no. 2: 115-125.
- Rogers, Hartley, Jr. 1967. *Theory of recursive Functions and Effective Computability*. Cambridge: The MIT Press.
- Rogers, Hartley. 1969. The Present Theory of Turing Machine Computability (1957). In *The Philosophy of Mathematics*, ed. Jaakko Hintikka. London: Oxford University Press.
- Savitch, W J. 1969. Deterministic simulation of non-deterministic Turing machines. In *Conference Record ACM symposium on the Theory of Computing*:247-248.
- Searle, John R. 1990. Is the Brain a Digital Computer. In *Proceedings and Addresses of the American Philosophical Association*, 64:21-37. Los Angeles.
- Shepherdson, J. C. and H. E. Sturgis. 1963. Computability of recursive functions. *Journal of ACM* 10: 217-255.
- Sipser, M. 1992. The history and status of the P versus NP problem. In *Proceedings of the 24th Annual ACM symposium on the theory of Computing*:pp. 603-618.
- Smith, Jeffrey D. 1989. *Design and Analysis of Algorithms*. Boston: PWS-KENT Publishing Company.
- Spaan, E, L. Torenvliet, and P. van Emde Boas. 1989. Nondeterminism, fairness and a fundamental analogy. *EATCS Bulletin* 37: 186-193.
- Stewart, Iain A. 1996. The Demise of the Turing Machine in Complexity Theory. In

Machines and Thought: The Legacy of Alan Turing, ed. P J R Millican and A Clark, 1. Oxford: Clarendon Press.

Trakhtenbrot, B. A. 1984. A Survey of Russian Approaches to Perebor (Brute-Force Search Algorithms). *Annals of the History of Computing* 6, no. 4: 384-400.

Turing, A. M. 1937. Computability and Lambda-Definability. *The Journal of Symbolic Logic* 2, no. 4: pp. 153-164.

Turing, Alan M. 1965a. On Computable Numbers, with an Application to the Entscheidungsproblem. In *The Undecidable*, ed. Martin Davis. Hewlett: Raven Press.

Turing, Alan M. 1965b. Systems of Logic Based on Ordinals. In *The Undecidable*, ed. Martin Davis. Hewlett: Raven Press.

Wang, Hao. 1974. *From Mathematics to Philosophy*. New York: Humanities Press.

Webb, Judson C. 1980. *Mechanism, Mentalism, and Metamathematics*. Boston: D. Reidel Publishing Company.

Yablonskii, S. B. 1959. The Algorithmic Difficulties of Synthesizing Minimal Switching Circuits. *Problems of Cybernetics* 2: 401-.

Yamada, H. 1962. Real-Time computation and recursive functions not real-time computable. *IRE Transactions EC-11*: 753-760.

CURRICULUM VITAE

Walter Warwick, Research Analyst

EDUCATION

PhD Candidate, History and Philosophy of Science, Indiana University.
MS, Computer Science, Indiana University 6/00.
Logic Certificate, Program in Pure and Applied Logic, Indiana University 6/00.
MA, Philosophy, Tufts University, 2/96.
BA, Philosophy, Magna cum Laude, University of Colorado, 8/92.

AWARDS

Indiana University Fellowship, 1995.
Philosophy Scholarship, Tufts University, 1993.

EMPLOYMENT

Research Analyst, Micro Analysis and Design, from 1999 to present.
Associate Instructor, Indiana University, from 1996 to 1999.

PROFESSIONAL EXPERIENCE

As a Research Analysts at Micro Analysis and Design, I am responsible for researching and developing computational models of Recognition Primed Decision making (RPD). Growing out of the school of Naturalistic Decision Making, RPD theory is regarded as a foil to traditional, rational-choice strategies of decision-making. RPD presents interesting challenges from a computational point of view; where rational-choice strategies of decision-making are often simulated by straightforward cost-benefit routines, my approach to RPD depends on an abstract representations of the decision-maker's knowledge together with a fuzzy-recognition algorithm.

My graduate research concerns foundational questions about theoretical computer science. In particular, I have examined the historical and philosophical development of nondeterministic algorithms. It is an area that has received remarkably little attention—especially given the central role nondeterminism plays in the host of notoriously open questions in complexity theory.

As an Associate Instructor at Indiana University, I taught for both the Department of Mathematics and the Department of the History and Philosophy of Science.

PUBLICATIONS

Papers Published in Journals, Proceedings or Books

Archer, S., Warwick, W., & Oster, A. (2000). *Current Efforts to Model Human Decision Making in a Military Environment*. Paper presented at the Proceedings of the Military, Government, and Aerospace Simulation Symposium, Washington, DC.

Walter Warwick, "The Indeterminacy of Nondeterminism," Forthcoming in Proceedings for the 1999 New Trends in Cognitive Science Conference, Vienna, Austria.

Papers Presented at Meetings

"The Indeterminacy of Nondeterminism," presented to the Indiana University Logic Group, February 10, 1999.

"Russell and the Decline of Logicism," presented to the Indiana University Logic Group, April 15, 1998.

"Discrepancy and the Truth of Explanatory Law," presented to the Rocky Mountain Regional conference for the Philosophy of Science, Spring 1995.