12 THE SYMBOL-MANIPULATION SYSTEMS OF POST

12.0 INTRODUCTION

In chapter 5 we examined the idea of effective procedure and identified it with the idea of rule-obeying. We agreed that the notion of effective procedure could be identified with a system that includes

(1) A *language* for describing rules of behavior, and
(2) A *machine* that obeys statements in the language.

Up to now, we have concentrated primarily on the *machine* part of such systems, in studying Turing machines, program machines, and other instruction-obeying mechanisms. We now turn our attention toward the *language* in which rules can be expressed. Our methods will be based on the idea of Emil Post [1943] that the "expressions" or "enunciations" of a logical system or language, whatever else they may seem to be, are in the last analysis nothing but strings of symbols written in some finite alphabet. Even the most powerful mathematical or logical system is ultimately, in effect, nothing but a set of rules that tell how some *strings of symbols* may be transformed into other *strings of symbols*.

This viewpoint, in itself, appears to be an obvious but probably unprofitable truism. But, just as Turing was able to show the equivalence of his broadest notion of a computing machine with the very sharply restricted idea of a Turing machine, Post was able to reduce his broadest concept of a string-transformation system to a family of astoundingly special and simple symbol-manipulation operations. We can summarize Post's view by paraphrasing our statement, in section 5.2, of Turing's thesis:

> Any system for manipulation of symbols which could naturally be called a formal or logical (or

mathematical) system can be realized in the form of one of Post's "canonical systems."

When we see how simple are the canonical systems, this view might seem rash; but this and the following chapters will support it by showing the equivalence of these with our other formulations of effective computability.

In studying Post's systems, we will deal with rules, called "productions," which specify how one can get new strings of symbols from old ones. These productions are not, on the surface, anything like the kinds of rules we imagined for machines or for effective processes. For they are not *imperative* statements at all but are *permissive* statements. A production says how, from one statement, string, or "enunciation," of such and such a form, one *may* derive another string of a specified form. A canonical system, which is a set of such productions and some initially given statements, does not even describe a *process*; instead it specifies the extent of a set of strings by (recursively) specifying how to find things in that set. The interesting thing is how far we can proceed in our exploration without needing the idea of a machine or procedure at all. While eventually we show that the notion of a machine or process can be derived from this idea of canonical system (under certain special circumstances), the weaker, more general, idea of *set* can give us some valuable insights that are harder to see within the framework of machines and processes.

12.0.1 Plan of Part III

In this chapter we develop the idea of canonical systems, mainly by example. The formulation is oriented, as we have said, toward examining what happens to the symbols during a computation; the idea of a machine does not appear explicitly. But, by using a few technical tricks we convert what are, to begin with, hardly more than "rules of grammar" into representations of the behaviors of machines. And by a circuit of reasoning that encompasses the whole of several chapters (10–14) we finally close the loop and show that all our formulations of effectiveness are equivalent. Chapter 13 proves the beautiful theorems of Post about the equivalence of canonical systems in general with his very simple "normal-form" systems. Chapter 14 ties up a number of loose ends required to show the equivalence of Post systems with Turing machines. As by-products, we obtain a simple proof of the unsolvability of Post's famous "correspondence problem," and also a state-transition table that describes the smallest (but not simplest) universal Turing machine presently known.

12.1 AXIOMATIC SYSTEMS AND THE LOGISTIC METHOD

The *axiomatic method* is familiar to all of us from the study of Euclid's geometry. In the axiomatic method we begin with a set of "axioms"—assertions we are willing to accept, either because we believe them to be true or merely because we want to study them. Along with the axioms, we assume the validity of some "rules of inference"; these tell us precisely how we can obtain new assertions—called "theorems"—from the axioms and/or from previously deduced theorems.

Actually, Euclid paid more attention to the problem of formulating the axioms than to the problem of making perfectly clear what were the rules of inference, in geometry. Until the twentieth century, the logical structure of Euclidean geometry (as well as other axiomatic systems in mathematics) was somewhat confused because

> (1) The rules-of-inference were informal and depended on common sense,† and
> (2) the somewhat arbitrary or hypothetical nature of the axioms themselves was not clear until the discovery of the non-Euclidean geometries late in the nineteenth century.

Until the non-Euclidean geometries were discovered, there had always seemed to be a possibility that some of the axioms might be possibly derivable from some more absolute sort of reasoning process, so that it was difficult to think of geometry as a closed logical system.

> Over the last hundred years the study of the foundations of mathematics has become more intense. It was discovered that "common-sense reasoning"—ordinary logical "intuition"—was not always a sufficiently reliable tool for examining the finer details of the foundations. The study of the notion of infinity by Cantor and others led to results that were self-consistent but very surprising, and even counter-intuitive. A number of other seemingly reasonable and innocuous notions led to genuinely paradoxical results; this happened to the reasonable supposition that one could discuss relations between classes of things in terms of common-sense descriptions of the classes. (See chapter 9.) The notion of effective pro-

†Euclid's "postulates" are what we would today call axioms; some of his "axioms" we might consider to be rules of inference. Thus Euclid's axioms concerning the interchangeability of equal things might be considered to be rules of inference concerning permissible substitutions in expressions. Euclid saw, of course, that the statements concerning equal things were more general than the statements with specifically geometric content, and tried to separate these out from the "postulates."

cedure was old in mathematics, in an informal way, but the possibility of effective unsolvability, which plays so large a role in our preceding chapters, is new to mathematics; when Gödel announced his results in the 1930's, their possibility had been suspected only by a very few thinkers.† It became necessary to examine more closely the idea of a mathematical proof—to formalize all the steps of deduction—even more closely than was necessitated in the early 1900's after the discovery of Russell's paradox. The resulting theories—of the nature of "proof" itself—became more and more formal, and indeed became a new branch of mathematics. Since it is, in effect, a mathematical theory of mathematics itself, we call this study "metamathematics."‡

12.2 EFFECTIVE COMPUTABILITY AS A PREREQUISITE FOR PROOF

Before going into detail, we ought to explain how this apparent digression will lead back to the theory of machines and effective processes. The key point is this: We have to be sure, in accepting a logical system, that it is really "all there"—that the methods for deriving theorems have no dubious "intuitive" steps. Accordingly, we need to have an *effective procedure* to test whether an alleged proof of a statement is really entirely composed of deductive steps permitted by the logical system in question. Now "theorems" are obtained by applying rules of inference to previously deduced theorems, and axioms. Our requirement must then be that there is an *effective* way to verify that an alleged chain of inferences is entirely supported by correct applications of the rules.

To make this more precise, we will agree on a few definitions:

A *logistic system L* is a set of axioms and a set of rules of inference, as defined below.

An *alphabet* is a finite set of symbols. In the following definitions it is assumed that all "strings" are strings of symbols taken from some fixed alphabet $A = (a_1, a_2, \ldots, a_r)$

An *axiom* is a finite string of symbols. We will consider here only systems with finite sets of axioms.

A *rule of inference* is an effectively computable function $R(s; s_1, \ldots, s_n)$ of $n + 1$ *strings*. An R function can have only two values; 1 (for "true") and 0 (for "false"). If $R(s; s_1, \ldots, s_n) = 1$ we say that "s is immediately

†Notably Post. See his autobiographical remarks in Davis [1965].

‡There are several "branches" of metamathematics, of which the theory of proofs is one. Others are concerned with theories of sets, descriptions, relations between different formal systems and their decidability properties, independence and consistency of axiom systems, models, etc.

derivable from $s_1, \ldots,$ and s_n by the rule R." Alternatively we may write

$$\boxed{\begin{array}{c} s_1, \ldots, s_n \rightarrow s \\ R \end{array}} \qquad (R)$$

We will consider only finite sets of rules of inference. Often, in a logistic system, we will say simply that "s is immediately derivable from $s_1, \ldots s_n$" (without mentioning an R) if there is an R that can justify it. Since each R-test is effective, so is the test to see if *any* R will do, since there are only a finite set of R's.

A *proof*, in L, of a string s, is a finite sequence of strings

$$s_1, s_2, \ldots, s_K$$

such that

(1) each string s_j is either an axiom of L or else is immediately derivable from some set of strings all of which precede s_j in the sequence, and
(2) s is s_K, the last string in the sequence.

A *theorem* of L is any string for which there is a proof in L. (Show that every axiom of L is a theorem of L.)

Now our goal was to define "proof" in such a way that there is an effective way to tell whether an alleged proof is legitimate. This will be the case for the definitions given above. To see this, consider an alleged proof $s_1, \ldots, s_K$ of a string s. First see if s is s_K. (If not, this is not a proof of s in L.) Next see if s_K is an axiom. This test is effective, since once has to look at only a finite number of axioms, and each axiom is only a finite string of letters.

If s_K is s, but is not an axiom, we have to see if s_K is immediately derivable from some subset of earlier s_j's. There are a finite number of such subsets. For each we have to test a finite number of R's. Each R test is, by definition, effective. If s_K passes these tests, we go on to s_{K-1} and do the same thing (except that we don't require that s be s_{K-1}.). Clearly a finite number (namely, K) of iterations of this process will confirm or reject the proof, and clearly the whole procedure is effective, since it is composed of a finite set of finite procedures.

REMARK

Our definition of *logistic system* has a number of restrictions. One might want to consider infinite sets of axioms. We *can* do this, if we require that there be an effective test of whether an arbitrary string is an axiom; then the above argument still works, and the notion of proof is

still effective. This is often done in logic, where one may use an "axiom schema"—a rule which says that "any string of such and such a form is an axiom." One might also want to allow an infinite set of rules of inference. (Again one could get an effective proof-test if one had an effective schema to determine whether there is an appropriate rule of inference at each step.) For our purposes, the finite-based systems are adequate. Shortly we shall show that any finite-based system can be reduced, without any real loss of generality, to a system with *just one axiom and a rule of inference of a kind that operates on just one previous theorem at a time*! At that point, we will have the symbolic equivalent of a machine—a step-by-step process.

12.3 PROOF-FINDING PROCEDURES

The proof-checking process just described is certainly effective; given an *alleged proof* it gives us a perfectly methodical, unambiguous procedure to decide whether the alleged proof is valid. Now let us consider a slightly different question; given an *alleged theorem*, can we decide whether it is really a theorem—that is, whether there is a proof, in the system, for it. The answer is that, while there are some logistic systems for which such procedures exist, in general there are no such procedures. In any logistic system it *is* always possible to devise a procedure that will search through all possible proofs—so that if the alleged theorem is really a theorem, this fact will be confirmed. But, in general, if the alleged theorem is not really a theorem, there is no way to find this out.

To elaborate on this a little, let us see how one can, effectively, generate all proofs—and thus all theorems—mechanically. To do this, one can use a procedure that generates all *sequences of strings*, treats each as an alleged proof, and tests each by the proof-testing procedure described above. How does one generate all *sequences of strings*? One begins with a procedure that generates all *strings*, separately. How does one do that? First generate all *one-letter strings*; that is, go through the alphabet. Next, generate all *two-letter strings*. How? Take each string generated at the previous state (i.e., one-letter strings) and append to each, in turn, every letter of the alphabet. Next we can generate all *three-letter strings* by taking each two-letter string and appending to each, in turn, every letter of the alphabet. Clearly, one gets any finite string, eventually, this way.

PROBLEM 12.3-1. Design a Turing machine that writes out on its tape all finite strings in the three-letter alphabet (a, b, c) with these strings separated by some special punctuation letter X. The machine should be subject to a constraint (see, for example, section 9.2) that the machine never

moves to the left of an X, so that one can distinguish the strings the machine has enumerated from those it is currently working on.

PROBLEM 12.3-2. Before reading on, design a Turing machine that enumerates all *finite* sequences *of finite strings in* (a, b, c).

Now to enumerate all finite *sequences of finite strings*, one can modify the above procedure for generating all strings. After each step of the above procedure, one has obtained a new n-letter string. Now let us cut the string S into two parts, *in all ways*. (There are $n - 1$ ways to do this.) If we do this, after each string-generating act of the main procedure, then we have obtained all sequences of pairs of strings! In fact, we have done it so that each pair of strings is generated in exactly one way. So this gives us *all possible alleged proofs of length two.* (The original procedure gives us all of length one.) But now, suppose that after each *pair* of strings is generated, we again cut the *first* string of each pair into two parts, in all ways. (There is a variable number of ways, now, depending on where the original string was cut.) This gives us (verify this) *all possible sequences of three strings.* Indeed, suppose that each time a string is cut into two parts, we (recursively) then perform all possible cuts of the first part into two more parts. It is clear that for each originally generated string, this process, while lengthy and tedious, will be finite. In fact for an original string of length n, there are exactly 2^{n-1} sequences of strings that can be made of it.

PROBLEM 12.3-3. Prove that each string of length n can be cut into sequences of strings in exactly 2^{n-1} ways. Hint: There is a trivial proof.

So, finally, we have an effective process that generates all finite sequences of finite strings in the given alphabet. There are many other procedures to do this besides the one we have given; ours has the unimportant technical advantage that each sequence is generated exactly once. The order of generation of the sequences can serve as a Godel numbering (see section 14.3) for (alleged) proofs. In any case, as each sequence of strings is generated, one can apply our effective test to see whether it is a proof of some previously given theorem-candidate. If the string is really a theorem, this process will, in some finite time, yield a proof of it! *But if the string is not a theorem, the process will never terminate.* Hence we have a "proof procedure" for theorems, but we do not have a "theoremhood decision procedure" for strings. We will see, shortly, that in general there is no possibility of such a decision procedure.

REMARK

Some branches of mathematics actually do have decision procedures. For example, Tarski [1951] showed that a certain formulation of Euclid-

ean geometry has this property. The propositional calculus—i.e., the logic of deduction for simple sentences or for Boolean algebra—has a decision procedure; in fact the well known method of "truth tables" can be made into a decision procedure. But "elementary logic"—propositional logic with the quantifiers or qualification clauses "for all x..." and "there exists an x such that..." and symbols for functions or relations—is not, in general, decidable. See Ackermann [1954] for a study of cases where the decision problem is solvable, and Kahr, Moore, and Wang [1962] for some of the best results to date on showing which problems are not decidable. The reader who has read only this book is not quite prepared to study these papers and may have first to read a text like Rogers [1966]. Of course, even showing that a decision procedure exists, and presenting one, as in the case of Tarski's decision procedure for Euclidean geometry, does not necessarily mean that one gets a *practical* method for proving theorems! The methods obtained by logical analysis of a decision problem usually lead to incredibly lengthy computations in cases of practical interest.

12.4 POST'S PRODUCTIONS. CANONICAL FORMS FOR RULES OF INFERENCE

In 12.2 we defined a rule of inference to be an effective test to decide whether a string s can be deduced from a set of strings $s_1, \ldots, s_n$. We required the test to be effective so that we could require the test of a proof to be effective. But we did not really tie things down securely enough, for one might still make a rule of inference depend (in some effective way) upon some understanding of what the strings "mean." That is, one might have some rule of inference depend upon a certain "interpretation" of the strings as asserting things about some well-understood subject matter; the strings might, for example, be sentences in English. (For example, the strings in chapter 4—the "regular expressions"—were understood to represent the "regular sets," and our proofs about regular expressions used references to these understood meanings.)

To avoid such dangerous questions, we propose to restrict ourselves to rules of inference that concern themselves entirely with the arrangement of symbols within strings—i.e., to the visible form of the strings as printed on a page—and we rule out reference to meanings. This will force us, for the present, to direct our attention toward what is often called the domain of "syntax"—questions of how expressions are assembled and analysed—rather than the domain of "semantics"—questions of the meanings of expressions.

To make it plausible that this can actually be done with any prospect of success, we will paraphrase the arguments of Turing (as recounted in

chapter 5) as he might have applied them to the situation of an imaginary finite mathematician who has to manipulate symbolic mathematical expressions.

At all times our mathematician must work with finite strings of symbols using a finite alphabet, a finite set of axioms, and a finite set of rules of inference. Imagine that he is verifying the validity of an alleged proof of an alleged theorem. At each step, then, he will be confronted with some assertion—that is, a string of symbols

$$s = a_{i_1} \ldots a_{i_n}$$

and also with a sequence of assertions whose proofs he has already verified

$$s_1 = a_{11}\, a_{12} \ldots a_{1n_1}$$
$$s_2 = a_{21}\, a_{22} \ldots a_{2n_2}$$
$$s_r = a_{r1}\, a_{r2} \ldots a_{rn_r}$$

where n_j is the number of letters in the jth string. He must use one of the rules of inference, and he may, for example, try to apply each rule systematically to each subset of the established strings.

Now, paraphrasing Turing's argument, we will further suppose that the mathematician's resources are limited to a certain set of talents:

> He can "scan" a string of symbols and recognize therein certain fixed subsequences.†
>
> He can dissect these out of the string, and keep track of the remaining parts.
>
> He can rearrange the parts, inserting certain fixed strings in certain positions, and deleting parts he does not want.

We shall see that these abilities are all that are needed to verify proofs or to perform any other effective procedure!

A rule telling precisely how to dissect a string and rearrange its parts (perhaps deleting some and adding others) is called a "production." Rather than begin with a precise definition, we will start with a few examples of simple but complete and useful systems based on productions. Then, when we formalize the definition, it will be perfectly clear what is meant and why it is done that way.

†Later (in chapters 13 and 14) we will see that all that is really necessary is the ability to identify a single symbol—the first of a string—and act accordingly. This parallels Turing's observation that it is enough for a Turing machine to examine one square of its tape at a time.

EXAMPLE 1: THE EVEN NUMBERS

Alphabet: The single symbol 1.
Axiom: The string 11.
Production: If any string \$ is a theorem, then so is the string \$11. It is convenient to write this rule of inference simply as

$$\$ \rightarrow \$11$$

It is evident that the theorems of this system are precisely the strings

$$11, \quad 1111, \quad 111111, \quad 11111111, \quad \text{etc.}$$

that is, the even numbers expressed in the unary number system.

EXAMPLE 2: THE ODD NUMBERS

Alphabet: 1
Axiom: 1
Production: \$ → \$11

EXAMPLE 3: THE PALINDROMES

Alphabet: *a*, *b*, *c*.
Axioms: *a*, *b*, *c*, *aa*, *bb*, *cc*.
Productions: \$ → *a*\$*a*
\$ → *b*\$*b*
\$ → *c*\$*c*

The "palindromes" are the strings that read the same backwards and forwards, like *cabac* or *abcbbcba*. Clearly, if we have a string \$ that is already a palindrome, it will remain so if we add the same letter to the beginning and end. Also clearly, we can obtain all palindromes by building in this way out from the middle.

PROBLEM 12.4-1. Prove that this gives *all* the palindromes. (It obviously gives nothing else.)

PROBLEM 12.4-2. This example, while quite trivial, is interesting because this is a set of strings that cannot be "recognized" (in the sense of chapter 4) by a finite-state machine. Prove this. Show that if we remove the last three axioms but append the production

$$\$ \rightarrow \$\$$$

we still obtain the same set of theorems.

EXAMPLE 4: SIMPLE ARITHMETIC EQUATIONS

Suppose that we want a system whose theorems are *all true statements about adding positive integers*—that is, all sentences like

$$3 + 5 = 8$$
$$21 + 35 = 56, \quad \text{etc.}$$

For simplicity, we use unary notation, so that the theorems will resemble

$$111 + 11111 = 11111111$$

Alphabet: 1, +, =
Axiom: 1 + 1 = 11
Productions: $\$_1 + \$_2 = \$_3 \rightarrow \$_1 1 + \$_2 = \$_3 1$ (π_1)
$\$_1 + \$_2 = \$_3 \rightarrow \$_1 + \$_2 1 = \$_3 1$ (π_2)

Here the first production means: If there is a theorem that consists of some string $\$_1$, followed by a '+', then another string $\$_2$, then an '=', and finally another string $\$_3$; we can make a new theorem that consists of the first segment $\$_1$, then a '1', then '+', then $\$_2$, then '=', then $\$_3$, and finally another '1'. To see how this works, we derive the theorem that means "2 + 3 = 5"

1 + 1 = 11	axiom
11 + 1 = 111	by π_1
11 + 11 = 1111	by π_2
11 + 111 = 11111	by π_2

Another proof of the same theorem is

1 + 1 = 11	axiom
1 + 11 = 111	by π_2
1 + 111 = 1111	by π_2
11 + 111 = 11111	by π_1

There can be many proofs for the same theorem, in such a system.

PROBLEM 12.4-3. Replace π_2 by $\$_1 + \$_2 = \$_3 \rightarrow \$_2 + \$_1 = \$_3$ and prove the same theorem in the new system.

One can do much the same for multiplication:

Alphabet: 1, ×, =
Axiom: 1 × 1 = 1
Productions: $\$_1 \times \$_2 = \$_3 \rightarrow \$_1 1 \times \$_2 = \$_3 \$_2$
$\$_1 \times \$_2 = \$_3 \rightarrow \$_2 \times \$_1 = \$_3$

Prove that 3 × 4 = 12 in this system.

PROBLEM 12.4-4. Design a system whose theorems include arithmetic statements involving both addition and multiplication. This may be difficult at this point but will be easier after further examples.

EXAMPLE 5: WELL-FORMED STRINGS OF PARENTHESES

In 4.2.2 we defined the set of "well-formed strings of parentheses," namely the strings like

$$(),\quad (()),\quad (()()),\quad (())(),\quad (((())(((()))))$$

in which each left parenthesis has a matching right-hand mate. We can obtain all and only such strings as theorems of the system:

Alphabet: (,)
Axiom: ()
Productions:

$$\$ \rightarrow (\$) \qquad (\pi_1)$$
$$\$ \rightarrow \$\$ \qquad (\pi_2)$$
$$\$_1()\$_2 \rightarrow \$_1\$_2 \qquad (\pi_3)$$

For example, to derive the string (()(())):

()	axiom
(())	by π_1
(())(())	by π_2
((())(()))	by π_1
(()(()))	by π_3

PROBLEM 12.4-5. Prove (()()((()()))) in this system.

PROBLEM 12.4-6. Consider the same alphabet and axiom with the single production:

$$\$_1\,\$_2 \rightarrow \$_1()\$_2$$

Prove that this system generates all and only the well-formed parenthesis strings. Note: a \$ is allowed to represent an empty, or "null" string, so that () → ()() is permitted, for example.

12.5 DEFINITIONS OF PRODUCTION AND CANONICAL SYSTEM

Now let us define "production" more precisely. In each example of 12.4, every production could be written as a special case of the general form

ANTECEDENT	CONSEQUENT	
$g_0\$_1 g_1 \$_2 \ldots \$_n g_n$	$\rightarrow h_0\$'_1 h_1 \$'_2 \ldots \$'_m h_m$	(π)

with the qualifications:

> Each g_i and h_i is a certain *fixed* string; g_0 and g_n are often null, and some of the h's can be null.
> Each $\$_i$ is an "arbitrary" or "variable" string, which can be null.
> Each $\$'_i$ is to be replaced by a certain one of the $\$_i$.

Take for example the production

$$\$_1 \times \$_2 = \$_3 \rightarrow \$_1 1 \times \$_2 = \$_3 \$_2$$

from example 4 of 12.4. We can represent it in the form above by making the assignments:

$$\begin{array}{lll} g_0 = \text{null} & g_3 = \text{null} & h_2 = \text{'='} \\ g_1 = \text{'}\times\text{'} & h_0 = \text{null} & h_3 = \text{null} \\ g_2 = \text{'='} & h_1 = \text{'1}\times\text{'} & h_4 = \text{null} \end{array}$$

and

$$\begin{array}{ll} \$'_1 = \$_1, & \$'_2 = \$_2 \\ \$'_3 = \$_3, & \$'_4 = \$_2 \end{array}$$

Note that two (or more) of the $\$'_i$-s can be the same $\$_i$, as in the production above, where $\$'_4 = \$'_2 = \$_2$. This breaks up, diagrammatically, as:

$$\begin{array}{lll} & \text{ANTECEDENT} & \text{CONSEQUENT} \\ & \$_1 \times \$_2 = \$_3 & \rightarrow \quad \$_1\ 1\times\ \$_2 = \$_3\ \$_2 \\ & \quad | \quad | \quad | \quad | \quad | & \quad\ \ | \quad\ \ | \quad | \quad | \quad | \quad\ \ | \\ g_0\$_1\ g_1\ \$_2\ g_2\ \$_3 g_3 & & \rightarrow\ h_0\$_1\ \ h_1\ \$_2\ h_2\ \$_3 h_3 \$_2 h_4 \end{array}$$

REMARKS

Post's most general formulation allowed each production to have several *antecedents*. This is discussed in 13.2, and we prefer not to introduce this complication here; in 13.2 we show that the more general form is equivalent, in a sense, to the special single-antecedent forms used here.

Also in Post's most general formulations, he allowed two of the \$'s in the *antecedent* to be the same. This meant that the rule of inference would apply only to a string (theorem) in which there was an exact repetition of some (variable) sub-string in two places in the antecedent. We prefer to prohibit antecedents of this form, not because we want to restrict the generality of the systems, but because it would run counter to our intuitive picture of what ought to be permitted as elementary, unitary actions. The recognition of the identity of two arbitrarily long strings ought to have to be done by an iterative process; otherwise it violates Turing's dictum (see 5.3) about what can be "seen at a glance" and what requires a multi-stage process.[1]

DEFINITIONS

A *production* is a string-transforming rule of the general form of π above, or (more generally) of π in 13.2.

A *canonical system* is a logistic system specified by

(1) an *alphabet* A
(2) some *axioms* (strings in A)
(3) some *productions* whose constant strings are strings in A.

12.6 CANONICAL SYSTEMS FOR REPRESENTATION OF TURING MACHINES

Now we can show how a formal system, with only axioms and productions, can be arranged to model a process! On the surface, a formal system seems *permissive* rather than *imperative*; there would seem to be nothing that corresponds to the *process control* in a machine, nothing like an obvious mechanism that dictates "what is to be done *next*." Indeed, there is no notion of time or sequence, except, perhaps, the sequence of steps in a proof of a theorem. *Because, in general, there are many different proofs of a theorem, one would not expect to be able to use the proof-step sequence as a process-control mechanism.* But one can. By using some tricks—mainly the use of special punctuation symbols in various ways—we can, indeed, embed the notion of a machine within the concept of a formal system with only axioms and productions. We will show this by constructing a formal system which, in a very straightforward way, "simulates" the activity of a Turing machine.

EXAMPLE 6: PRODUCTIONS FOR TURING-MACHINE COMPLETE-STATE CHANGES

Let $(s_1, s_2, \ldots, s_r)$ be the alphabet, and $(q_1, q_2, \ldots, q_n)$ the states, of a certain Turing machine T. At any time t, T's tape will contain some finite sequence of symbols

$$s_{i_1}, s_{i_2}, \ldots, s_{i_{(n_t)}} \qquad (S)$$

where n_t is the length of the written part of the tape at time t. To show the complete state of the Turing machine at time t, we have also to specify (1) the current internal state of the machine and (2) where the machine is located on the tape. We can incorporate *all* of these facts, in a single tape-representing string, by including the machine's state-symbol at the appropriate place, e.g., by writing

$$s_{i_1}, s_{i_2}, \ldots, s_{i_{k-1}}, \boxed{q_i}, s_{i_k}, \ldots, s_{i_{n_t}}$$

This string is interpreted as follows: the machine is in state q_i; it is scanning the kth written square of its tape; the tape has on it the letter sequence (S); it is understood that the q_i is *not* written on the *machine's* tape.

Now we can represent the machine's operation by a set of Post productions! Let the quintuples of T be

$$(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$$

If d_{ij} is "*Right*," then the machine ought to proceed from any complete state represented by

$$\ldots s_k q_i s_j \ldots$$

to that represented by

$$\ldots s_k s_{ij} q_{ij} \ldots$$

while if d_{ij} is "*Left*," the machine ought to proceed to that represented by

$$\ldots q_{ij} s_k s_{ij} \ldots$$

This suggests using a set of productions, one for each (i, j, k) triple:

$$\$_1 s_k q_i s_j \$_2 \rightarrow \$_1 s_k s_{ij} q_{ij} \$_2 \qquad (\text{if } d_{ij} \text{ is Right})$$
$$\$_1 s_k q_i s_j \$_2 \rightarrow \$_1 q_{ij} s_k s_{ij} \$_2 \qquad (\text{if } d_{ij} \text{ is Left})$$

These rules cannot work when the machine—that is, when the symbol q_i—comes to an end of the written part of the tape. But we can make the system automatically extend the representation by adjoining the productions

$$\$q_i \rightarrow \$q_i 0 \qquad (\text{all } i)$$
$$q_i \$ \rightarrow 0 q_i \$ \qquad (\text{all } i)$$

These serve to add blank squares whenever the machine comes to an end of the tape. We can thus think of the Turing machine's computation as always represented by a finite string of symbols, with provision for lengthening this string when necessary. We now make the following assertion.

ASSERTION

Given the above productions, and given a string containing one q_i symbol for an axiom, the theorems of this canonical system will be precisely the sequence of all future complete states of the Turing machine, if started in the complete state represented by the axiom. Each theorem of the system will have a single, unique, proof, and the steps of the proof will be exactly those of the steps in that computation. (We include the tape-extending operations as steps in the Turing machine's operation.)

The proof of the assertion is simply that, if a string contains only one q symbol, then only one production can apply to it (Verify this!), and the

result produces a string that represents the next step of the Turing machine's computation because of the way the quintuples are realized in the system of productions.

PROBLEM 12.6-1. Why would the assertion be false if the s_k's were left out of the first (right) productions?

PROBLEM 12.6-2. Carry out this construction for the Turing machine of 6.1.1 (p. 120).

PROBLEM 12.6-3. Suppose that a certain Post canonical system has the single production

$$\$_1 xy\$_2 \rightarrow \$_1\$_2$$

that it is known to have a single axiom, and that it is known that the string xy is a theorem of the system. What can you say about the unknown axiom? More precisely, describe the class of all axioms from which the string xy can be derived, using only this production. Prove your statement.

EXAMPLE 7: A CANONICAL SYSTEM FOR GENERATING THE SQUARE NUMBERS

For our next example, we want to generate the sequence of numbers

$$1, 4, 9, 16, 25, \ldots$$

(in the form of unary strings: $1, 1111, 111111111, \ldots$). Observing that $(n + 1)^2 = n^2 + (2n + 1)$—that is, that we get from one square number to the next by adding the corresponding *odd* number—we use the system:

Alphabet: $1, P$

Axiom: $1\ A$

Production: $\$_1\ A\ \$_2 \rightarrow \$_1 11\ A\ \$_2\$_1$

This generates, in sequence, the strings

$1P$	$= 1^1P1^0$
$111P1$	$= 1^3P1^1$
$11111P1111$	$= 1^5P1^4$
$1111111P111111111$	$= 1^7P1^9$
$11111111P1111111111111111$	$= 1^9P1^{16}$

In a sense, the square numbers are being generated. The symbol P is used to separate two quantities; on the left is computed the next odd number to be used; on the right is the sum of the odd numbers of earlier stages (which is also the desired square number). Generalizing, we see that one could use more punctuators like P to keep track of more different auxiliary quantities that one might want to keep, during a computation. In the next example, we will keep track of three quantities in this way.

PROBLEM 12.6-4. See if you can sketch, at this point, how one could realize the computations of a program machine, of the kind described in 11.1, by a system of productions, using one punctuation letter for each machine register. How many auxiliary letters are really needed? The solution is given in section 12.8.

PROBLEM 12.6-5. (Fairly difficult.) Prove that there is no system of productions whose theorems are the square numbers (in unary notation) which uses only the symbol '1' in its alphabet—that is, which has no extra punctuation letters.

12.7 CANONICAL EXTENSIONS. AUXILIARY ALPHABETS

There is a serious defect in example 7 of the previous section. We would really like to find a canonical system whose *only* theorems are the strings 1, 1111, 11111111, etc. Instead, we found a system which generates, inside its theorems, the desired information but does not produce it in the desired form. If we adjoin one more production

$$\$_1 P \$_2 \rightarrow \$_2$$

then we obtain, in addition to the theorems already found, also the theorems

$$1, 1111, 11111111, 1111111111111111, \text{etc.}$$

We now have the theorems we want, but we also have the "working results"

$$1P, 111P1, 11111P1111, \text{etc.}$$

which we do not want. Now we can easily distinguish between the desired theorems and the working results because the latter all contain the symbol P while the former do not. So we can say that

> *The square numbers are those theorems of the canonical system*
>
> Alphabet: $1, P$
>
> Axiom: $1P$
>
> Productions: $\$_1 P \$_2 \rightarrow \$_1 11 P \$_2 \$_1$
>
> $\$_1 P \$_2 \rightarrow \$_2$
>
> *which are also strings in the smaller alphabet containing only* 1.

It turns out, in general, that auxiliary letters, like the P above, are necessary when canonical systems are used to produce sets of theorems of

theoretical interest. Let us make a general definition, to recognize and deal with this fact.

Suppose that we are interested in a certain system M, whose theorems are expressed in a certain alphabet A. (M is not necessarily a canonical system.) Suppose that M' is another system, with a larger alphabet A'. Some theorems of M' may use only letters in A, other theorems will use additional letters.

DEFINITION

If the theorems of M are precisely those theorems of M' that use only the letters of A, then we say that M' is an extension of M over A. If M' is a Post canonical system, then we say M' is a *canonical extension* of M over A.

In example 7 of section 12.6, the system is a canonical extension of any system whose theorems are the square numbers. The first production does the work, while the second production is used as a sort of output device; it converts a string with a P to one without a P. In doing this, it serves to *release* the result—a square number—which cannot be further modified (because it contains no P). In this way we can use an "auxiliary letter"—one in the extension alphabet but not in the original—to control what is transformed and to protect already-derived theorems from being incorrectly transformed. The next section gives a more elaborate example of such a computation.

PROBLEM 12.7-1. Construct a Post canonical extension for the set of repeated strings, e.g., those that have the form \$\$. The palindromes of section 12.4, example 3, were realized without using any extension letters. Can this be done here? If not, prove it. The alphabet should have two original letters.

EXAMPLE 8: A CANONICAL EXTENSION FOR THE PRIME NUMBERS

This example shows how a few productions can lead to quite complicated behavior. It is an extension of the (unary) prime integers over the alphabet '1'. We have added the string '11' as an axiom because the system otherwise generates only the primes from '111' on.

Alphabet: $1, A, B, C, D$

Axioms: $A111, 11$

Productions:

$$A\$ \rightarrow A\$1 \qquad (\pi_1)$$

$$A\$1 \rightarrow C\$DB\$1 \qquad (\pi_2)$$

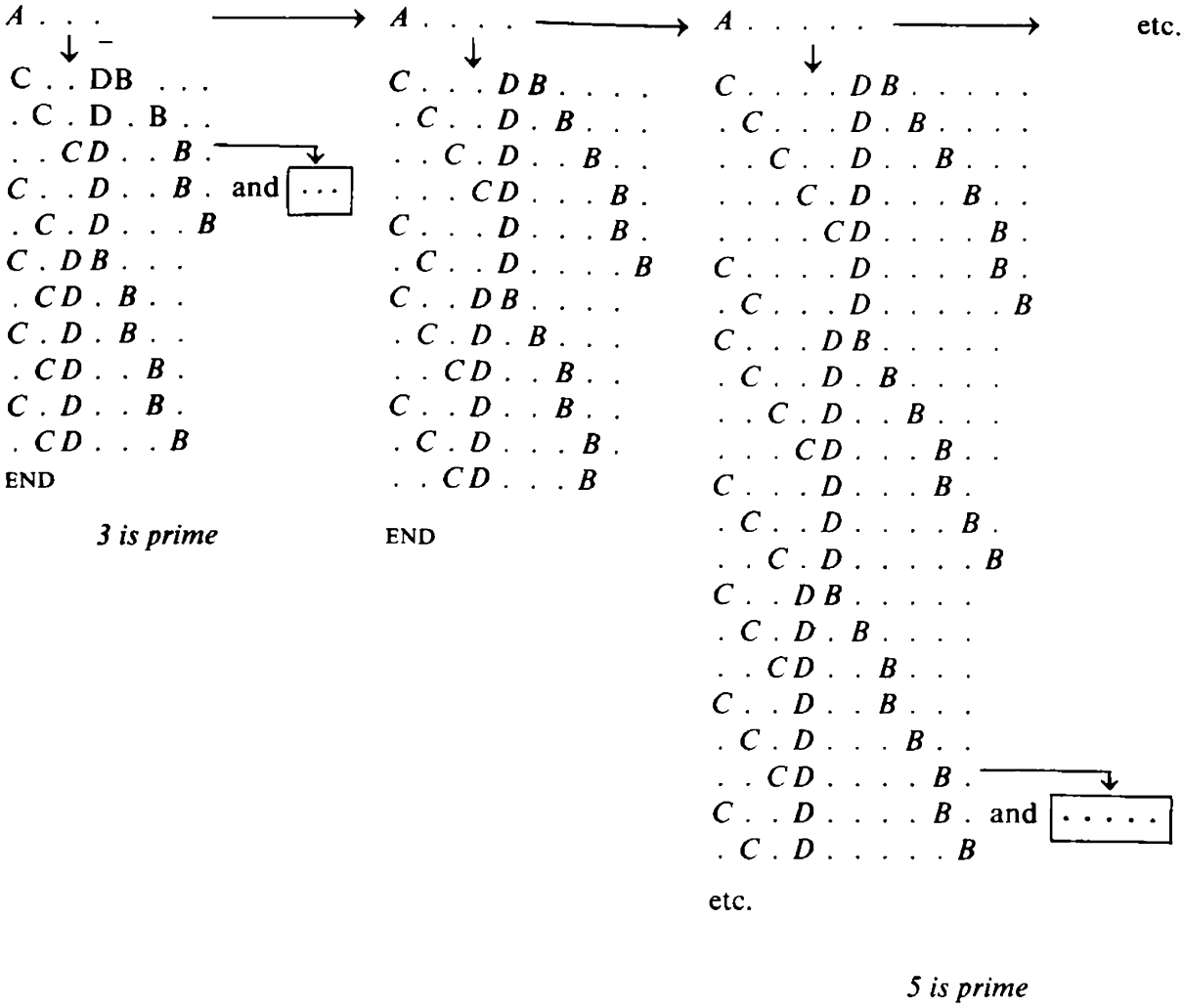
$$\$_1 1 D\$_2 1 \rightarrow 1\$_1 D 1\$_2 \qquad (\pi_3)$$

$$\$_1 CD\$_2 1 \rightarrow C\$_1 D\$_2 1 \qquad (\pi_4)$$

$$\$_1 C\$_2 1 D\$_3 B \rightarrow C\$_1\$_2 DB\$_3 \qquad (\pi_5)$$

$$11CD\$B1 \rightarrow \$1 \qquad (\pi_6)$$

Table 12.7-1



The diagram in Table 12.7-1 traces out all the strings generated by the system. We use a dot instead of '1' for typographical clarity.

12.8 CANONICAL SYSTEMS FOR PROGRAM-MACHINES

By using a rich variety of auxiliary letters, one can design canonical systems to simulate the steps of a complicated process. We have already done this (section 12.6, example 6) by using a new auxiliary letter for each state of a Turing machine. We can do the same for each of the instructions of a program machine (chapter 11). Let the instructions of a program machine P be numbered $I_1, \ldots, I_n$ and let the registers of the machine be $R_1, \ldots, R_r$. Let us consider a machine whose instructions are of the two kinds: Each I_j is either

add 1 to R_k and go to I_{j+1}

or

if R_k contains zero **then** go to $I_{j'}$, **else** subtract 1 and go to I_{j+1}.

By section 11.4, this is a universal base for program machines. We can simulate this program machine with a canonical system constructed as follows:

Alphabet: $I_1, \ldots, I_n, R_1, \ldots, R_r, R_{r+1}, 1$

Axiom: $I_1 R_1 11 \ldots 11 R_2 11 \ldots 11 R_3 \ldots R_r 11 \ldots 11 R_{r+1}$

which is understood to mean that the machine starts with I_1 and has the given unary numbers in its registers at the start.

Productions: If I_j is "Add 1 to R_k and go to I_{j+1}," use:

$$I_j \$_1 R_k \$_2 \rightarrow I_{j+1} \$_1 R_k 1 \$_2$$

If I_j is "**If** $R_k \neq 0$ **then** subtract 1 and go to I_{j+1}, **else** go to $I_{j'}$," use the pair:

$$I_j \$_1 R_k R_{k+1} \$_2 \rightarrow I_{j'} \$_1 R_k R_{k+1} \$_2$$
$$I_j \$_1 R_k 1 \$_2 \rightarrow I_{j+1} R_k \$_2$$

This system is truly "monogenic" (see 14.6). That is, given a string S, it is never possible for more than one production to be applicable to S. For, in the case of an *addition* instruction I_j, there is only one production that begins with the letter I_j. In the case of a *subtraction* instruction I_j, there are two productions beginning with I_j, but since, in any string, R_k is followed either by a '1' or by R_{k+1}, only one of these two productions can apply. To release the final result as a number, suppose that I_n is a halt instruction and the answer is to be found in R_r. Then we adjoin the production

$$I_n \$_1 R_r \$_2 R_{r+1} \rightarrow \$_2$$

which will release the unary string contained in register R_r. For an improved form of this theorem, see the remark at the end of section 14.1.

PROBLEM 12.8-1. Show how to make a canonical extension for the set of strings represented by a regular expression.

PROBLEM 12.8-2. Show how to construct a set of productions that gives the set of strings (as an extension) recognized by a finite-state machine, directly from the state-transition table of the machine, without reference to the regular-expression analysis.

PROBLEM 12.8-3. Show, given a system of Post productions and axioms —i.e., given a canonical system—how to make a Turing machine that will generate all the theorems of this system.

NOTE

1. Several recent advances in computer programming languages are based on string-dissecting operations that resemble closely Post productions. In particular they derive their power, in part, from unrestricted use of occurrences of the same string variable. The first such language was COMIT, developed by Victor Yngve [1962] for use in research on linguistic analysis. Following this came SNOBOL (see Farber [1964]) and a series of related languages embedded in the LISP system, described in Bobrow [1964], Guzman [1966], and Teitelman [1966].