1 Recursive Functions

§1.1 THE INFORMAL NOTION OF ALGORITHM

In this chapter we give a *formal* (i.e., mathematically exact) characterization of *recursive function.* The concept is basic for the remainder of the book. It is one way of making precise the *informal* mathematical notion of function computable "by algorithm" or "by effective procedure." In this section, as a preliminary to the formal characterization, we discuss certain aspects of the *informal* notions of *algorithm* and *function computable by algorithm* as they occur in mathematics.

Roughly speaking, an algorithm is a clerical (i.e., deterministic, bookkeeping) procedure which can be applied to any of a certain class of symbolic *inputs* and which will eventually yield, for each such input, a corresponding symbolic *output.* An example of an algorithm is the usual procedure given in elementary calculus for differentiating polynomials. (The name *calculus,* of course, indicates the algorithmic nature of that discipline.)

In what follows, we shall limit ourselves to algorithms which yield, as outputs, integers in some standard notation, e.g., arabic numerals, and which take, as inputs, integers, or k-tuples of integers for a fixed k, in some standard notation. Hence, for us, an algorithm is a procedure for computing a *function* (with respect to some chosen notation for integers). For our purposes, as we shall see, this limitation (to numerical functions) results in no loss of generality. It is, of course, important to distinguish between the notion of *algorithm,* i.e., procedure, and the notion of *function computable by algorithm,* i.e., mapping yielded by procedure. The same

function may have several different algorithms. We shall occasionally refer to functions computable by algorithm as *algorithmic functions.*†

Here are several examples of functions for which well-known algorithms exist (with respect to the usual denary notation for integers).

a. λx[xth *prime number*]. (The method of *Eratosthenes' sieve* is an algorithm here.) (We are assuming Church's lambda notation. To say that $f = \lambda x$[xth *prime number*] is to say that for all x, $f(x) = x$th *prime number.*‡)

b. λxy[*the greatest common divisor of* x *and* y]. (The *Euclidean algorithm* serves here.)

c. λx[*the integer* ≤ 9 *whose arabic numeral occurs as the* xth *digit in the decimal expansion of* $\pi = 3.14159 \cdots$]. (Any one of a number of common approximation methods will give an algorithm, e.g., quadrature of the unit circle by Simpson's rule.)

Of course there are even simpler and commoner examples of functions computable by algorithm. One such function is

d. $\lambda xy[x + y]$. Such common algorithms are the substance of elementary school arithmetic.

Several features of the informal notion of algorithm appear to be essential. We describe them in approximate and intuitive terms.

*1. *An algorithm is given as a set of instructions of finite size.* (Any classical mathematical algorithm, for example, can be described in a finite number of English words.)

*2. *There is a computing agent, usually human, which can react to the instructions and carry out the computations.*

*3. *There are facilities for making, storing, and retrieving steps in a computation.*

*4. *Let P be a set of instructions as in* *1 *and L be a computing agent as in* *2. *Then L reacts to P in such a way that, for any given input, the computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.*

*5. *L reacts to P in such a way that a computation is carried forward deterministically, without resort to random methods or devices, e.g., dice.*§

Virtually all mathematicians would agree that features *1 to *5, although inexactly stated, are inherent in the idea of algorithm. The reader will note an analogy to digital computing machines: *1 corresponds to the

† Beginning in §1.5, we shall extend our use of the word *algorithm* to include procedures for computing nontotal partial functions.

‡ As we proceed, we shall assume, without further comment, the conventions of notation and terminology set forth in the Introduction. In addition to the lambda notation, the restriction of *function* and *partial function* to mean mappings on (nonnegative) integers is important for Chapter 1.

§ In a more careful discussion, a philosopher of science might contend that *4 implies *5. Indeed, he might question whether there is any real difference between *4 and *5.

program of a computer, *2 to its *logical* elements and circuitry, *3 to its storage *memory*, *4 to its *digital* nature, and *5 to its *mechanistic* nature.

A straightforward approach to giving a *formal* counterpart to the idea of algorithm is, first, to specify the symbolic expressions that are to be accepted as sets of instructions, as inputs, and as outputs (we might call this the *P-symbolism*), and, second, to specify, in a uniform way, how any instructions and input determine the subsequent computation and how the output of that computation is to be identified (we might call this the *L-P specifications*).

Once we begin a search for a useful choice of *P*-symbolism and *L-P* specifications, *1 to *5 serve as a helpful intuitive guide. There are, however, several features of the informal idea of algorithm that are less obvious than *1 to *5 and about which we might find less general agreement. We discuss them briefly here, formulating them as questions and answers. Later, after we have settled on a particular formal characterization, we shall return and see how our answers accord with our chosen formal characterization. There are five questions. They are closely interrelated, as will be evident, and all have to do with the role of arbitrarily large sizes and arbitrarily long times.

The first three questions are:

*6. *Is there to be a fixed finite bound on the size of inputs?*

*7. *Is there to be a fixed finite bound on the size of a set of instructions?*

*8. *Is there to be a fixed finite bound on the amount of "memory" storage space available?* (For each of *6, *7, and *8, size could be measured by the number of elementary symbols (or English words) used.)

Most mathematicians would agree in answering "no" to *6. They would assert that a general theory of algorithms should concern computations which are possible *in principle*, without regard to practical limitations. For the same reason, they would agree in answering "no" to *7. However, *7 raises an issue that is already implicit in *6, namely, what sort of intellectual "capacity" do we require of *L*? If instructions are to be unbounded in size, will not this require unbounded "ability" of some kind on the part of *L* in order that *L* may comprehend and follow them? We consider this further under *9 below.

Question *8 is interesting in that physically existing computing machines *are* bounded in their available storage space. One might at first suppose that a negative answer to *8 is implied by our negative answers to *6 and *7, since arbitrarily large inputs and sets of instructions would, in themselves, require arbitrarily large amounts of space for storage. We can interpret *8, however, as referring to that storage space which is necessary over and above the space needed to store instructions, input, and output. Under this interpretation, *8 becomes of interest, apart from our answers to *6 and *7. We might conceive, for instance, of an ordinary computing machine of fixed finite size and fixed finite memory where the instructions *P* take

the form of a finite printed tape fed into the machine, where the input is fed in on a second tape which (unlike the instruction tape) moves in only one direction, and where the output is printed, digit by digit, on a third tape which moves in only one direction. It is not difficult to show that a number of simple functions, including $\lambda x[2x]$, can be computed by an arrangement of this kind.† It is possible, however, to make a rather convincing and general argument that the function $\lambda x[x^2]$ cannot be computed by any such arrangement; as input x increases, larger and larger amounts of space for "scratch work" are required. On account of this narrowness, most mathematicians would answer "no" to any form of question *8. We therefore take "no" as our answer to questions *6, *7, and *8.

Our comments on *7 lead us to a fourth question about the informal notion of algorithm.

*9. *Is there to be, in any sense, a fixed finite bound on the capacity or ability of the computing agent L?* Let the reader imagine the following situation: he is given unlimited supplies of ordinary paper and pencil; he is given two tapes upon each of which is written a 1-million digit integer; and he is asked to apply the Euclidean algorithm to these integers and to write the result on a third tape. After some reflection, the reader will find it credible that he could work out a bookkeeping and cross-reference system whereby he could keep track of his progress and mark his place at various stages of the computation, and whereby he could indeed carry out the computation satisfactorily, given enough time. Indeed, the reader could doubtless find a uniform system that would work for input integers of arbitrary size. By such a system, he would, in effect, transfer excessive demands on his own mental capacities as L into additional demands on his (unlimited) paper-and-pencil memory storage. Similar "place-marking" systems can be introduced when the set of instructions P is of great length and complexity, provided that P is sufficiently well organized and detailed. Such a system would serve to "mark one's place" in P as well as in the input, output, and computation. In fact, we would expect that such a place-marking system could, in some sense, be made a part of P itself, if the P-symbolism is sufficiently flexible. We therefore answer "yes" to question *9.

When we later present and discuss our formal characterization, we shall see that these rather vague plausibility arguments can be substantiated (see §1.8). Indeed, once the P-symbolism and computation symbolism are given in sufficiently detailed form, it is possible to limit L to the following (without otherwise limiting the notion of algorithm): (*a*) a few simple clerical operations, including operations of writing down symbols, operations of moving one symbol at a time backward or forward in the

† Such functions are sometimes called *functions computable by finite-state machine.* (What functions are so computable depends, in part, on the choice of symbolism for inputs and outputs.) See Exercise 2-14.

computation to or from symbols previously written, operations of moving one symbol at a time backward or forward in P to or from symbols previously examined, and operations for writing the output; (*b*) a finite short-term memory of fixed size which at any point preserves symbols written or examined in various of the preceding steps; and (*c*) a fixed finite set of simple rules according to which the clerical operation next to be performed and the next state of the short-term memory are uniquely determined by the contents of the short-term memory together with the symbol written or examined last. (This remark will become clearer after §§1.5 and 1.8.)

We now turn to a final and somewhat deeper question about the informal notion of algorithm. It is a question upon which considerable disagreement can exist.

*10. *Is there to be, in any way, a bound on the length of a computation? More specifically, should we require that the length of a particular computation be always less than a value which is "easily calculable" from the input and from the set of instructions P? To put it more informally, should we require that, given any input and given any P, we have some idea, "ahead of time," of how long the computation will take?*

The question is vague. If one is to give an affirmative answer without begging the question, one must define "easily calculable" with care. Nevertheless, an affirmative answer to *10 is an essential feature of the notion of algorithm for many mathematicians.

We propose, however, to make no such affirmative answer to the question, arguing that it is simpler and more natural to accept such a restriction only if it proves to be a consequence of our other assumptions. We thus require only that a computation terminate after *some* finite number of steps; we do not insist on an a priori ability to estimate this number. As we shall see, this attitude toward *10 will accord with the formal characterization we select. To the extent that a reader can make *10 precise and can give an affirmative answer to *10 which is not a consequence of our formal characterization—to that extent will his informal notion be narrower than our formal characterization.

As we shall see (Theorem XI in §1.10), our position on *10 is fundamental. The absence of any such a priori requirement is a distinctive feature of the discipline developed in the remainder of this book.

§1.2 AN EXAMPLE: THE PRIMITIVE RECURSIVE FUNCTIONS

One method for characterizing a class of functions is to take, as members of the class, all functions obtainable by certain kinds of *recursive definition.* A recursive definition for a function is, roughly speaking, a definition wherein values of the function for given arguments are directly related to

values of the same function for "simpler" arguments or to values of "simpler" functions. The notion "simpler" is to be specified in the chosen characterization—with the constant functions, among others, usually taken as the simplest of all. This method of formal characterization is useful for our purposes, in that recursive definitions can often be made to serve as algorithms.

Recursive definitions are familiar in mathematics. For instance, the function f defined by

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 1, \\ f(x+2) &= f(x+1) + f(x), \end{aligned}$$

gives the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ... (The study of *difference equations* concerns the problem of going from recursive definitions to algebraic definitions. The Fibonacci sequence is given by the algebraic definition

$$f(x) = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^{x+1} - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^{x+1}.)$$

The *primitive recursive functions* are an example of a broad and interesting class of functions that can be obtained by such a formal characterization.

Definition The class of *primitive recursive functions* is the smallest class $\mathfrak{C}$ (i.e., intersection of all classes $\mathfrak{C}$) of functions such that

(i) All *constant functions*, $\lambda x_1 x_2 \cdots x_k[m]$, are in $\mathfrak{C}$, $1 \leq k$, $0 \leq m$;

(ii) The *successor function*, $\lambda x[x+1]$, is in $\mathfrak{C}$;

(iii) All *identity functions*, $\lambda x_1 \cdots x_k[x_i]$, are in $\mathfrak{C}$, $1 \leq i \leq k$;

(iv) If f is a function of k variables in $\mathfrak{C}$, and $g_1, g_2, \ldots, g_k$ are (each) functions of m variables in $\mathfrak{C}$, then the function $\lambda x_1 \cdots x_m[f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))]$ is in $\mathfrak{C}$, $1 \leq k,m$;

(v) If h is a function of $k+1$ variables in $\mathfrak{C}$, and g is a function of $k-1$ variables in $\mathfrak{C}$, then the unique function f of k variables satisfying

$$\begin{aligned} f(0,x_2, \ldots, x_k) &= g(x_2, \ldots, x_k), \\ f(y+1, x_2, \ldots, x_k) &= h(y,f(y,x_2, \ldots, x_k),x_2, \ldots, x_k) \end{aligned}$$

is in $\mathfrak{C}$, $1 \leq k$. (For (v), "function of zero variables in $\mathfrak{C}$" is taken to mean a fixed integer.)

It follows directly from the definition that, for any f, f is primitive recursive if and only if there is a finite sequence of functions $f_1, f_2, \ldots, f_n$ such that $f_n = f$ and for each $j \leq n$, either f_j is in $\mathfrak{C}$ by (i), (ii), or (iii), or f_j is directly obtainable from some of the f_i, $i \leq j$, by (iv) or (v). (To show this, let $\mathfrak{D}$ be the class of all functions f for which such a sequence $f_1, \ldots, f_n$ exists. $\mathfrak{D}$ is evidently contained in every class $\mathfrak{C}$ that is closed under (i) to (v); furthermore $\mathfrak{D}$ is itself closed under (i) to (v). It follows that $\mathfrak{D}$

coincides with the intersection of all such $\mathfrak{C}$.) If such a sequence for f is described, together with a specification of how each f_j is obtained for $j \leq n$, we say that we have a *derivation* for f as a primitive recursive function.

For an example, consider the function f given by the derivation

$f_1 = \lambda x[x]$	by (iii)	(a function of 1 variable)
$f_2 = \lambda x[x + 1]$	by (ii)	(1 variable)
$f_3 = \lambda x_1 x_2 x_3[x_2]$	by (iii)	(3 variables)
$f_4 = f_2 f_3$	by (iv)	(3 variables)
f_5 to satisfy		
$f_5(0,x_2) = f_1(x_2)$		
$f_5(y + 1, x_2) = f_4(y,f_5(y,x_2),x_2)$	by (v)	(2 variables)
$f = f_6 = f_5(f_1,f_1)$	by (iii)	(1 variable).

It is easy to verify that f_6 is the function $\lambda x[2x]$ (and, incidentally, that f_5 is $\lambda xy[x + y]$). Hence we can conclude that the function $\lambda x[2x]$ is primitive recursive.

A derivation can be written down in any one of a number of standard symbolic forms. A written derivation can serve as a set of instructions for effectively computing the function which it defines. For instance, to compute $f(2)$ in the preceding example, the derivation leads us to the computation

$$\begin{aligned} f(2) &= f_6(2) \\ &= f_5(f_1(2),f_1(2)) \\ &= f_5(2,f_1(2)) \\ &= f_5(2,2) \\ &= f_4(1,f_5(1,2),2) \\ &= f_4(1,f_4(0,f_5(0,2),2),2) \\ &= f_4(1,f_4(0,f_1(2),2),2) \\ &= f_4(1,f_4(0,2,2),2) \\ &= f_4(1,f_2(f_3(0,2,2)),2) \\ &= f_4(1,f_2(2),2) \\ &= f_4(1,3,2) \\ &= f_2(f_3(1,3,2)) \\ &= f_2(3) \\ &= 4. \end{aligned}$$

We obtain the computation uniquely by working from the inside out and from left to right.

All this suggests that we include the precise notion of primitive recursive function within our informal notion of function computable by algorithm. How does this accord with our discussion in §1.1? The computing agent is human (and *not* formally defined); nevertheless, the computation depends on the derivation in so simple and direct a way and via such obviously

mechanical steps, that *1 to *5 are evidently satisfied. We can choose a standard P-symbolism for expressing derivations, and the L-P specifications are the simple substitution rules according to which a derivation and input determine a computation. Note, in passing, that questions *6 to *8 receive the same answers for primitive recursive functions as were given in §1.1. Question *9 must remain vague, since the computing agent is not formally defined. Question *10, as we shall indicate in a moment, can be given the answer "yes."

How inclusive is the class of primitive recursive functions? Perhaps it is broad enough to include all desired algorithms, and perhaps, in consequence, one can contend that it is an accurate formal counterpart to the informal notion of *function computable by algorithm.* Although the defining rules for primitive recursive functions might at first seem limited, one can supply an impressive body of evidence to support this contention. Virtually all the algorithmic functions of ordinary mathematics can be shown to be primitive recursive. (All the examples so far mentioned in this chapter are primitive recursive.) Ways to illustrate and demonstrate the breadth of primitive recursiveness are found in Péter [1951, pp. 1–67].

Unfortunately, it is possible to construct functions, with obvious algorithms, which are not primitive recursive. One such is the *Ackermann generalized exponential*, a function f of three variables such that

$$
\begin{aligned}
f(0,x,y) &= y + x,\\
f(1,x,y) &= y \cdot x,\\
f(2,x,y) &= y^x,\\
&\cdots\cdots\cdots\\
f(z+1,\ x,\ y) &= \text{result of applying } y \text{ to itself } x-1 \text{ times}\\
&\qquad\qquad \text{under the } z\text{th level operation } \lambda uv[f(z,u,v)].
\end{aligned}
$$

A more formal (and "recursive") definition for this f is given by the conditions:

$$
\begin{aligned}
&f(0,0,y) = y,\\
&f(0,\ x+1,\ y) = f(0,x,y) + 1,\\
&f(1,0,y) = 0,\\
&f(z+2,\ 0,\ y) = 1,\\
&f(z+1,\ x+1,\ y) = f(z,f(z+1,\ x,\ y),y).
\end{aligned}
$$

There is no primitive recursive derivation for this function (see Péter [1951, p. 68]). Indeed, as Péter in effect shows, a function similar to f can be used to obtain an "easily calculable" function that gives an affirmative answer to *10 for the primitive recursive functions (if we take "easily calculable" to mean having simple defining conditions, like those for f above).

Since the generalized exponential would be almost universally accepted as a function computable by algorithm, and since it is not primitive recur-

sive, we must reject the primitive recursive functions as an accurate formal counterpart to the informal notion of algorithmic function.†

§1.3 EXTENSIONALITY

As was remarked in §1.1, it is important to distinguish between the notion of *algorithm* and the notion of *algorithmic function.*‡ We now give several examples further to emphasize this distinction. In particular, we define a function g for which we can prove that an algorithm exists but for which we do not know how to get a specific algorithm. Consider the functions f and g defined by

$$f(x) = \begin{cases} 1, & \text{if a consecutive run of } \textit{exactly } x \text{ 5's occurs in the} \\ & \text{decimal expansion of } \pi; \\ 0, & \text{otherwise;} \end{cases}$$

and

$$g(x) = \begin{cases} 1, & \text{if a consecutive run of } \textit{at least } x \text{ 5's occurs in the} \\ & \text{decimal expansion of } \pi; \\ 0, & \text{otherwise.} \end{cases}$$

At the present time, no algorithm is known for computing f. Indeed, it may be that no algorithm exists for f. (Once our formal characterization is given, the notion of a *function having no algorithm* will become precise. We shall see that such functions exist.) In contrast to our ignorance about f, we do have the knowledge that g is primitive recursive. For either g must be the constant function $\lambda x[1]$, or else there must exist some fixed k such that

$$g(x) = 1, \quad \text{for } x \leq k,$$

and

$$g(x) = 0, \quad \text{for } k < x.$$

In either case, a primitive recursive derivation exists (see Exercise 2-1), but no one knows, at the present time, how to identify the correct derivation.

For an even simpler example, take an unsettled conjecture of mathematics, e.g., Goldbach's conjecture that every even number greater than 2 is the sum of two primes, and define a function h by

$$h(x) = \begin{cases} 1, & \text{if conjecture true;} \\ 0, & \text{if conjecture false.} \end{cases}$$

† The question naturally arises: does there exist a function of *one variable* which is algorithmic but not primitive recursive? It can be shown that $\lambda x[f(x,x,x)]$, where f is the Ackermann generalized exponential, is such a function. We shall see another example in §1.4.

‡ Note that one and the same primitive recursive function can have an infinite number of different derivations, i.e., algorithms. One trivial way of obtaining such derivations is to insert additional appearances of $\lambda x[x]$ in a given derivation.

h is evidently a constant function. Hence it is primitive recursive, though again we do not know how to identify its correct derivation.†

We shall be concerned both with functions and with algorithms. Our chief emphasis will be on functions. In traditional logical terminology, our emphasis will be *extensional*, in that we shall be more concerned with objects *named* (functions) than with objects *serving as names* (algorithms).

§1.4 DIAGONALIZATION

In §1.2 we gave an example of an (intuitively) algorithmic function that is not primitive recursive. We now look at a method which can be applied to a variety of formally characterized classes of algorithmic functions and which, in each case, produces an algorithmic function falling outside of the given formally characterized class. We call this method *diagonalization* and describe it through an example. In the example, we apply the method to the primitive recursive functions.

Consider all possible primitive recursive derivations. It is easy to set up a precise formal symbolism for derivations which uses only a finite number of basic symbols. These symbols would include a function symbol; several symbols for variables; digits for subscript numerals; digits for ordinary numerals; parentheses; the comma; plus and equals signs; several special symbols for indicating constant, successor, and identity functions; and a special symbol to mark the end of a line. Any derivation could then be represented as a single finite string of these basic symbols. Furthermore, an obvious effective (i.e., algorithmic) test would exist for determining, given any string of basic symbols, whether or not that string constituted a legitimate primitive recursive derivation. Hence we could list, in sequence, all possible primitive recursive derivations by first examining all strings of length 1, then examining all strings of length 2, etc. Indeed, we could give a definite, if informal, algorithmic procedure for making this list. (The list is infinite, but each derivation is reached at some finite point.) From this, we could, in turn, devise an algorithmic procedure which would list just the derivations for primitive recursive functions of one variable. Let Q_x be the $(x + 1)$st derivation in this latter list. Let g_x be the function determined by Q_x. Define h, by

$$h(x) = g_x(x) + 1.$$

† The proofs that g and h are primitive recursive use the logical principle of *the excluded middle*. Such nonconstructive methods are qualified or rejected in various "constructive" reformulations of mathematics, such as that of the *intuitionists*. Throughout this book we allow nonconstructive methods; we use the rules and conventions of classical two-valued logic (as is the common practice in other parts of mathematics), and we say that an object exists if its existence can be demonstrated within standard set theory. We include the axiom of choice as a principle of our set theory.

Evidently, we have an algorithm for computing h; namely, to get $h(x)$ for given x, generate the list of derivations out to Q_x, then employ Q_x to compute $g_x(x)$, then add 1. On the other hand, h cannot be primitive recursive. If it were, we would have $h = g_{x_0}$ for some x_0. But then we would have $g_{x_0}(x_0) = h(x_0) = g_{x_0}(x_0) + 1$, a contradiction. (The reader will note an analogy to Cantor's *diagonal proof* of the nondenumerability of the real numbers, in classical set theory.)

It is evident that the diagonalization method has wide scope, for it is applicable to any case where the sets of instructions in the P-symbolism can be effectively (i.e., algorithmically) listed. At first glance, it is difficult to see how a formal characterization can avoid such effective listing and still be useful. The diagonal method would appear to throw our whole search for a formal characterization into doubt. It suggests the possibility that no single formally characterizable class of algorithmic functions can correspond exactly to the informal notion of algorithmic function. Perhaps, no matter what P-symbolism and L-P specifications we choose, that symbolism and those specifications can be augmented by stronger symbolism and more complex "effective" operations to yield new functions. Even if we use the entire English language as P-symbolism, it may be that there are more complex clerical operations that demand new names. Perhaps, indeed, the algorithmic functions form a nondenumerable class, and perhaps there is a spectrum of algorithmic computability upon which *all* functions fall.

These are some of the considerations and difficulties, albeit vague and informal, that surround the problem of getting a satisfactory characterization of algorithm and of algorithmic function. They had to be faced by mathematicians who first addressed themselves to that problem in the 1930's, mathematicians who were stimulated in their work by recent successes of formal logic and its methods.

§1.5 FORMAL CHARACTERIZATION

We can avoid the diagonalization difficulty by allowing sets of instructions for nontotal partial functions as well as for total functions. Of course, situations may then arise where there is no evident way to determine whether a set of instructions yields a total function or not. Assume, for example, that we can have an expression of the P-symbolism which embodies the instructions: "To compute $f(x)$, carry out the decimal expansion of π until a run of at least x consecutive 5's appears; if and when this occurs, give the position of the first digit of this run as output." Or, for a simpler example, take: "To compute $g(x)$, examine successive even numbers greater than 2 until one appears which is not the sum of two primes; if and when this occurs, give the output $g(x) = 0$." In each example, unlike the illustra-

tions in §1.3, where we had nonconstructive definitions for specific functions (but no algorithms), we have a specific computing procedure but do not know whether this procedure gives a function, i.e., whether it always terminates and yields an output. What we *can* conclude is that each procedure gives a *partial function*. If it should happen to be true that there are runs of eight 5's but none of greater length in π, then the first example would give a set of instructions for a partial function whose domain consisted of the first nine integers. If Goldbach's conjecture is true, then the second example would give the empty partial function; if the conjecture is false, then the second example would give the constant function $\lambda x[0]$. In any case, each example provides specific calculating instructions which determine a specific partial function.

With a formal characterization for a class of *partial functions* we are not immediately subject to the diagonalization difficulty. For let ψ_x be the partial function determined by the $(x + 1)$st set of instructions Q_x, and let x_0 be chosen so that ψ_{x_0} is the partial function φ defined by the following instructions: to compute $\varphi(x)$, find Q_x, compute $\psi_x(x)$, and if and when a value for $\psi_x(x)$ is obtained, give $\psi_x(x) + 1$ as the value for $\varphi(x)$. The equation $\psi_{x_0}(x_0) = \varphi(x_0) = \psi_{x_0}(x_0) + 1$ does not yield a contradiction, since $\varphi(x_0)$ does not need to have a value. We might perversely hope to reinstate diagonalization by effectively selecting just those sets of instructions which do yield total functions; however, as we have noted, there may be no evident way to do this. Indeed, if we are to avoid diagonalization, it must be the case that no algorithm for such a selection procedure can exist. (These comments are related to the basic *incompleteness theorems* of mathematical logic. We discuss this further in Chapter 2.)

The approach by way of partial functions is, in essence, the approach taken by Kleene [1936], Church [1936], Turing [1936], and others in the 1930's. Each obtained a formal characterization for a wide class of partial functions. The characterizations differed both in outline and in detail. They had the common features, however, *first*, of giving (through a *P*-symbolism) a formal counterpart to the notion of *algorithm* (for partial functions) and *second*, in consequence (and via *L-P* specifications), of giving a counterpart to the notion of *partial function computable by algorithm*.†,‡

† Virtually all the discussion and terminology of §§1.1 and 1.3 can be applied, *mutatis mutandis*, to the problem of characterizing algorithm for a *partial* function and *partial* function computable by algorithm.

‡ Historically, in a number of instances, e.g., that of Kleene, the investigator did not give an explicit first treatment of partial functions but rather gave a single, more complex characterization for total functions computable by algorithm. In retrospect, each of these more complex characterizations can be analyzed into two steps: first, characterization of the algorithmic partial functions; and second, identification of the algorithmic functions as those algorithmic partial functions which happen to be total. Our discussion of the Kleene characterization below will make this retrospective modification, although it will detract, in certain respects irrelevant to our purposes, from the simplicity of Kleene's original formulation.

The Turing Characterization

We look at Turing's characterization first (as presented in Davis [1958]). The Turing characterization will be taken as basic in this book. It is convenient and instructive to approach the Turing characterization by way of a physical picture, although the ultimate characterization is entirely mathematical.

Consider a finite mechanical device which has associated with it a paper tape of infinite length in both directions. The tape is marked off, along its length, into spaces of equal size. We refer to these spaces as *cells*. The device is arranged so that the tape runs through it and so that there is room for one tape cell to lie within it. We refer to the tape cell lying within the device as the cell being *examined* by the device. We refer to the two directions along the tape as *right* and *left*. The device is equipped to perform any of four basic operations: (1) it can write a "1" on the cell it is examining, if no "1" already appears in that cell; (2) it can erase the cell it is examining and thus make it blank, if that cell is not blank already; (3) it can shift its attention one cell to the right along the tape (by shifting the tape one cell to the left); (4) it can shift its attention one cell to the left along the tape (by shifting the tape one cell to the right). The device, when active, performs basic operations at a rate of one operation per unit time. We sometimes refer to the performance of a basic operation as a *step* in the working of the device. At the conclusion of any step, the device itself (as distinct from the tape) takes on one of a fixed finite set of possible internal (mechanical) configurations. We refer to these internal configurations as *internal states*. We use the symbols q_i, $i = 0, 1, 2, \ldots$, to denote distinct internal states. Finally, the device is so constructed that it behaves according to a certain finite list of deterministic rules. These rules determine, from the current internal state and the condition of the cell being examined, the operation next to be performed and the internal state next to be taken on (at the end of that next operation).† Let 1 and B denote the possible conditions (B for *blank*) of any cell. Let 1, B, R, and L denote the basic operations (1), (2), (3), and (4), respectively. (Operation (1) makes no change on the tape if the cell being examined already has a "1" in it; and operation B makes no change on the tape if the cell being examined already is blank.) Then the set of rules determining the behavior of the device may be expressed as a set of *quadruples*. Each quadruple consists of symbols (in order) for (i) an internal state, (ii) a possible tape-cell condition, (iii) an operation, (iv) an internal state. A quadruple (i, ii, iii, iv) expresses the rule that given (i) and (ii), the device performs (iii) and takes on (iv). We allow any set of such quadruples as a set of rules for a device, subject only to the restriction that any two distinct quadruples must differ

† We assume that the device is equipped to "read" the condition of the tape cell lying within it.

at (i) or (ii). We call this the *consistency restriction;* it prevents a set of rules from requiring two or more different courses of action at the same time. We do not, however, ask that every combination of (i) and (ii) be provided for in a set of rules; thus we permit a device to perform *no* operation under certain circumstances. In such circumstances we say that the device *stops.*

As an illustration, consider a device with states q_0 and q_2 whose behavior is determined by the two quadruples $q_0 1 B q_2$ and $q_2 B R q_0$. If such a device is given a tape with a finite run of consecutive 1's and is started in state q_0 on the leftmost cell of the run, it will erase all the 1's and then stop. If such a device is started on a tape consisting entirely of 1's, it will never stop.

A device of the general kind described above is called a *Turing machine.* For our purposes, a Turing machine can be identified with the set of quadruples defining its behavior. Any set of quadruples, using any (finite) number of internal states, constitutes a Turing machine, provided only that the consistency requirement is satisfied.

It is easy to define *Turing machine* in more orthodox mathematical language. Let $T = \{0,1\}$ and $S = \{0,1,2,3\}$. Then a Turing machine can be defined as a mapping from a finite subset of $N \times T$ into $S \times N$. Here T represents conditions of a tape cell, S represents operations to be performed, and N gives possible labels for internal states.

Given a Turing machine, a tape, a cell on that tape, and an initial internal state, the Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. We can associate a partial function with each Turing machine in the following way. To represent an input integer x, take a string of $x + 1$ consecutive 1's on the tape. Start the machine in state q_0 on the leftmost cell containing a 1. As output integer, take the total number of 1's appearing anywhere on the tape when (and if) the machine stops. Under this convention for inputs, outputs, and initial conditions, each Turing machine yields a partial function.† For example, the following machine yields the function $\lambda x[2x]$, as the reader can verify.

$$
\begin{aligned}
&q_0 1 B q_1\\
&q_1 B R q_2\\
&q_2 1 B q_3\\
&q_3 B R q_4\\
&q_4 1 R q_4\\
&q_4 B R q_5\\
&q_5 1 R q_5
\end{aligned}
$$

† If q_0 does not explicitly occur in the set of quadruples, then the machine takes no action, and the partial function determined is $\lambda x[x + 1]$. Although the convention for inputs, outputs, and initial conditions is rather arbitrary, we shall see that the same *class* of partial functions is determined under a wide variety of such conventions.

q_5B1q_6
q_61Rq_6
q_6B1q_7
q_71Lq_7
q_7BLq_8
q_81Lq_1
q_11Lq_1.

This machine terminates either in state q_2 (for input 0) or in state q_8 (for input other than 0). The reader should carry through by hand the computation of this machine for input 2, i.e., for input tape $\cdots B111B \cdots$. He will see that the quadruples can be grouped according to various *subroutines* for (*a*) erasing an input digit, (*b*) moving right to the output digits, (*c*) adding two new output digits, (*d*) moving left to remaining input digits, etc.

We can also associate a partial function of k variables with each Turing machine by representing an input $<x_1, \ldots, x_k>$ as a tape with a string of $x_1 + 1$ consecutive 1's followed by a B followed by $x_2 + 1$ consecutive 1's followed by a $B \cdots$ followed by $x_k + 1$ consecutive 1's; and by again starting the machine in state q_0 on the leftmost cell containing 1. The reader can verify that, for a function of k variables, the particular machine described in the preceding paragraph yields $\lambda x_1 \cdots x_k[2x_1 + x_2 + \cdots + x_k + k - 1]$.

Each stage in a Turing-machine calculation can be described by giving (i) the condition of the tape, (ii) the internal state of the Turing machine, and (iii) the location of the cell being examined. We say that such information determines an *instantaneous description*. This information can be expressed in the form

$$\cdots q_i \cdots,$$

where $\cdots\cdots$ is a string of adjacent 1's and B's from the tape which includes all that portion of the tape that is not blank, where q_i is the current state, and where "q_i" is inserted immediately to the left of the symbol for the cell currently being examined.

For example, if the above machine for $\lambda x[2x]$ is applied to the input tape for 2, then the following instantaneous descriptions give the first few steps in the computation.

q_0111
q_1B11
q_211
q_3B1
q_41
$1q_4B$
$\cdots$.

At first glance, the class of algorithmic partial functions given by Turing machines might appear rather limited. Nevertheless, the definition does provide us with P-symbolism and L-P specifications. Each set of quadruples defining a machine may be viewed as a set of instructions P. As basic symbols for our P-symbolism we need only q, 1, B, R, L, and digits for numeral subscripts. We may take the computing agent L to be human. The L-P specifications are the simple rules according to which a succession of machine-tape configurations is determined from the initial tape and from P. The relation between the Turing-machine definition and the discussion of §1.1 will be considered further in §§1.6 and 1.8.

The Kleene Characterization

We next look at Kleene's formal characterization. Consider recursive relations of the general kind employed in §1.2 to define the Ackermann exponential. A set of instructions P consists of a set of such "recursion equations." A *computation* is a finite sequence of equations, beginning with P, where each equation after P is obtained from preceding equations *either* by the substitution of a numeral expression for a variable symbol throughout an equation *or* by the use of one equation to substitute "equals for equals" at any occurrence in a second equation *or* by the evaluation of an instance of the successor function $\lambda x[x + 1]$.

In P, we allow *auxiliary function symbols*, in addition to the *main function symbol* in whose evaluation we are interested. Thus the set of equations

$$\begin{aligned} f(0) &= 0, \\ g(x) &= f(x) + 1, \\ f(x + 1) &= g(x) + 1, \end{aligned}$$

with g as auxiliary function symbol and f as main symbol, determines the function $\lambda x[2x]$, as can be verified.

Three related difficulties arise in connection with this notion of computation: (1) the course of a computation is not uniquely determined by the input and by P; (2) it is possible that two different outputs may be obtained from the same input (by different computations); (3) it is possible that *no* output may be obtainable from a given input.†,‡

We avoid difficulties (1) and (2) in the following way. We say that an equation is *deducible* from a given P if it is obtainable in the course of *some* computation from P. It is possible to describe a uniform procedure

† Kleene, in his original characterization, simply accepts difficulty (1) and goes on formally to define algorithmic functions as those functions which are obtainable from sets of instructions which happen not to be subject to difficulties (2) and (3). The fact that there is no evident way of identifying which those sets of instructions are is the price paid for avoiding diagonalization trouble.

‡ If, in the Turing characterization, the consistency restriction were dropped, then the Turing characterization (thus modified) would be subject to a difficulty similar to (2).

according to which, given any P, we can effectively generate a list of all equations deducible from P. (The procedure makes, in effect, an exhaustive list of all possible computations and is similar to the procedure discussed in §1.4 for listing all possible primitive recursive derivations.) Formal details of such a procedure would be complex, and we do not give them. Assume that we have made a fixed and permanent selection of such a procedure. Then given any P and given any input, we define the *principal output* for that input as the *first* output associated with it in the standard list of deducible equations generated from P. Each input can yield at most one principal output; thus the relation between inputs and principal outputs defines a partial function. We can hence associate a partial function with every set of equations P; and we have a formal characterization for a class of algorithmic partial functions.

The P-symbolism in this case can be described in greater detail as follows. Take as basic symbols f, g, x, $+$, $=$, parentheses, comma, digits for ordinary numerals, and digits for numeral subscripts (for use with g and x).

x, x_0, x_1, . . . are called *variables.*

f, g, g_0, g_1, . . . are called *function symbols,* and f is the *principal* function symbol.

Terms are defined inductively as follows:

A variable is a term;

An ordinary numeral is a term;

If τ is a term, τ followed by "+ 1"
is also a term;

Let σ consist of a function symbol followed by parentheses containing a string of terms which are separated by commas; then σ is also a term.†

The expression formed by placing "=" between two terms is called an *equation.*

Any finite set of such equations constitutes a set of instructions P.

The *L-P* specifications describe how the uniform list of all derivable equations (from any given P) is to be generated. We omit details.

The reader will note, incidentally, that every primitive recursive derivation is expressible in the P-symbolism in an obvious way. The derivation for $\lambda x[2x]$ in §1.2 would become a set of equations beginning

$$\begin{aligned} g_1(x) &= x, \\ g_2(x) &= x + 1, \\ g_3(x_1,x_2,x_3) &= x_2, \\ &\cdots\cdots\cdots. \end{aligned}$$

It is easy to show that the function determined by these equations in the new formalism is identical with the function determined by the original primitive recursive derivation according to the procedures of §1.2.

† An expression such as "$x + 2$" can, in effect, be used as a term but must be written as "$x + 1 + 1$".

§1.6 THE BASIC RESULT

In §1.1 we raised the question of whether or not a characterization can be found which supplies a satisfactory formal counterpart to the informal notions of algorithm and algorithmic function. In §1.4 we indicated difficulties that any such characterization must face; in particular, we mentioned (§1.4) the possibility that there might be no single, maximal, formally characterizable class of algorithmic functions. In §1.5 we gave the formal characterizations of Turing and of Kleene. During the 1930's and since then, separate formal characterizations have been proposed by Church [1936], Post [1936], Markov [1951], and others. These characterizations have varied widely in form; each, however, can be represented as a certain choice of P-symbolism and a certain choice of L-P specifications.

How satisfactory are these various characterizations? What is their relation to each other? How successfully do they avoid the difficulties of §1.4? Extensive work has been done on these questions. We summarize this work in the following *Basic Result,* which is fundamental for the remainder of the book.

Basic Result

Basic Result, Part I *By means of detailed combinatorial studies* (see, for example, Turing [1937] and Kleene [1936*a*],) *the proposed characterizations of Turing and of Kleene, as well as those of Church, Post, Markov, and certain others, were all shown to be equivalent; that is to say, exactly the same class of partial functions (and hence of total functions) is obtained in each case.*

Definition The functions falling within this class are called *recursive functions.* The partial functions of this class might, naturally, be termed "recursive partial functions." It has become standard usage, however, to call them *partial recursive functions.*

These equivalence demonstrations can be generalized to show that over certain very broad families of enlargements of these formal characterizations the class of partial functions obtained remains unchanged. (For example, if we allow more than one tape, or other symbols than 1 and B, in the definition of Turing machine, the partial functions obtainable are still partial recursive functions; see Turing [1936]. For a development based on programming techniques, see Shepherdson and Sturgis [1963].) *In fact, if certain general (and reasonable) formal criteria are laid down for what may constitute a P-symbolism and L-P specifications, it is possible to show that the class of partial functions obtained is always a subclass of the "maximal" class of all partial recursive functions.*

Basic Result, Part II *A wide variety of particular partial functions, each agreed to be intuitively algorithmic, have been studied. Each has been*

demonstrated to be a partial recursive function; that is to say, a set of instructions for it has been found within one of the standard formal characterizations. A variety of useful principles and techniques have been developed for making these demonstrations. (Parts I and II thus provide strong empirical evidence that the formal characterizations are sufficiently inclusive.)

Basic Result, Part III *The proofs for the results in Part I have the following common structure. In every instance, the fact that one formally characterized class of partial functions is contained in another is demonstrated by supplying and justifying a uniform procedure according to which, given any set of instructions P from the first characterization, we can find a set of instructions P from the second characterization for the same partial function. Although it does not operate on integers, this uniform procedure itself happens, in each instance, to be* algorithmic *(in the unrestricted informal sense of that word, with no restriction to numerical inputs and outputs).*

Comment. In Parts I and II, emphasis was extensional, i.e., on the class of algorithmic partial functions defined rather than on the class of "algorithms" defined. Parts I and II show that there is a sense in which each standard formal characterization appears to include all possible algorithmic *partial functions.* Part III, taken together with Part I, now shows that there is a sense in which each standard characterization appears to include all possible *algorithms* (for partial functions). For, given a formal characterization of the kind mentioned at the end of Part I, there is a uniform effective way to "translate" any set of instructions (i.e., algorithm) of that characterization into a set of instructions of one of the standard formal characterizations. (We discuss this further in §1.7; see also Exercise 2-11.)

Some of the detailed work upon which the Basic Result rests will be found in Davis [1958] and in Kleene [1952]. (These references include several principles and techniques of the kind mentioned in Part II of the Basic Result.) In work on the Basic Result, the Turing definition has been especially useful as a standard to which other characterizations can be reduced.

From a mathematical point of view, the formal characterizations of §1.5 are *noninvariant,* i.e., they are dependent on arbitrary choices. This is true of all known characterizations. The Basic Result shows that the characterizations nevertheless provide a natural and significant class of partial functions. It has been remarked that this class is one of the few *absolute* mathematical concepts to originate in work on foundations of mathematics.†

† Where *absolute* means "existing apart from and largely independer symbolic formulations." Quantificational provability (for classical two is another such concept.

§1.7 CHURCH'S THESIS

The claim that each of the standard formal characterizations provides satisfactory counterparts to the informal notions of *algorithm* and *algorithmic function* cannot be proved. It must be accepted or rejected on grounds that are, in large part, empirical. (That the claim for one characterization is equivalent to the claim for another follows from Parts I and III of the Basic Result.) The Basic Result provides impressive evidence that the class of partial functions defined is a natural one (Part I) and that it is sufficiently inclusive (Parts I and II). The Turing characterization provides convincing evidence that every partial function in the class is computable by a procedure that is, intuitively, "mechanical." (In §1.10 we shall discuss further the possibility that the formal class is too inclusive; see question *10 in §1.1.) On the basis of this evidence, many mathematicians have accepted the claim that the standard characterizations give a satisfactory formalization, or "rational reconstruction," of the (necessarily vague) informal notions. This claim is often referred to as *Church's Thesis.* Church's Thesis may be viewed as a *proposal* as well as a claim, a proposal that we agree henceforth to supply certain previously intuitive terms (e.g., "function computable by algorithm") with certain precise meanings.

In recent theoretical work, the phrase "Church's Thesis" has come to play a somewhat broader role than that indicated above. In Parts II and III of the Basic Result, we noted that a number of powerful techniques have been developed for showing that partial functions with informal algorithms are in fact partial recursive and for going from an informal set of instructions to a formal set of instructions. These techniques have been developed to a point where (*a*) a mathematician can recognize whether or not an alleged informal algorithm provides a partial recursive function, much as, in other parts of mathematics, he can recognize whether or not an alleged informal proof is valid, and where (*b*) a logician can go from an informal definition for an algorithm to a formal definition, much as, in other parts of mathematics, he can go from an informal to a formal proof. Recursive-function theory, of course, deals with a precise subject matter: the class of partial functions defined in §1.5. Researchers in the area, however, have been using informal methods with increasing confidence. We shall rely heavily on such methods in this book. They permit us to avoid cumbersome detail and to isolate crucial mathematical ideas from a background of routine manipulation. We shall see that much profound mathematical substance can be discussed, proved, and communicated in this way. *We continue to claim, however, that our results have concrete mathematical status as results about the class of partial functions formally characterized in* §1.5. *Of course, any investigator who uses informal methods and makes such a claim must be prepared to supply formal details if challenged.*

Proofs which rely on informal methods have, in their favor, all the evidence accumulated in favor of Church's Thesis. Such proofs will be called *proofs by Church's Thesis.*

We meet our first examples of such informal methods in the remaining sections of this chapter. Almost all the proofs in this book will use Church's Thesis to some extent. The analogy to informal methods of proof in other parts of mathematics is instructive. In both cases, the use of informal methods is a matter not of extremes but of degree. The degree of formalization of a proof usually depends upon the complexity and abstraction (what might be called the "danger") of the argument. The degree of formal detail we employ in this book will similarly vary with circumstances.

The beginning reader, who does not possess first-hand knowledge of the evidence for Church's Thesis, may be troubled by our arguments. To whatever extent he experiences doubt, we urge him to use the books of Davis and Kleene, in which he will find the tools needed to formalize our arguments fully.

§1.8 GÖDEL NUMBERS, UNIVERSALITY, *s-m-n* THEOREM

We have adopted the Turing-machine characterization as basic. We saw in §1.5 that a set of instructions is a set of quadruples satisfying the consistency restriction. It is possible to list all sets of instructions by a procedure similar to that indicated in §1.4 for listing all primitive recursive derivations. This procedure is itself algorithmic (in our first, unrestricted, informal sense of that word). It can be viewed as a procedure which associates with each integer x the set of instructions falling at the $(x + 1)$st place in the list of all sets of instructions. We assume now that we have selected one such listing procedure. We keep it fixed for the remainder of the book. We do not give formal details.

Definition P_x is the set of instructions associated with the integer x in the fixed listing of all sets of instructions. x is called the *index* or *Gödel number* of P_x.

$\varphi_x^{(k)}$ is the partial function of k variables determined by P_x. x is also called an *index* or *Gödel number* for $\varphi_x^{(k)}$. (We shall drop the superscript (k) when its value is clear from context or when $k = 1$. We shall be most often concerned with functions of one variable.†)

Clearly the listing procedure gives us *both* (a) an algorithm for going from any x to the corresponding P_x, and (b) an algorithm for going from any consistent set of quadruples P to a corresponding integer x such that P is P_x.

† The notation $\{x\}$ (for our φ_x) also appears in the literature. We use the φ_x and P_x notations to emphasize further the distinction between extension and name, i.e., between partial function and set of instructions.

Our fixed choice of Gödel numbering will be used throughout the book. It appears to be a rather noninvariant feature of our theory. We shall see, however (Chapter 4 and Exercise 2-10), that our results possess an invariant significance independent of this choice. In this respect, our use of a particular Gödel numbering is much like the use of a particular coordinate system to establish coordinate free results in geometry.

Several facts, already implicit in the Basic Result, can now be stated.

Theorem I *There are exactly $\aleph_0$ (a countable infinity of) partial recursive functions, and there are exactly $\aleph_0$ recursive functions.*

Proof. All constant functions are recursive, by Church's Thesis. Hence there are at least $\aleph_0$ recursive functions. The Gödel numbering shows that there are at most $\aleph_0$ partial recursive functions. ☒

Theorem II *There exist functions which are not recursive.*

Proof. By Cantor's theorem, there are $2^{\aleph_0}$ (continuously many and therefore nondenumerably many) functions. The theorem follows. ☒

The next fact appears to depend, for its proof, upon our particular characterization and Gödel numbering.

Theorem III *Each partial recursive function has $\aleph_0$ distinct indices.*

Proof. We need only show that there are at least $\aleph_0$ indices. Let a partial recursive function φ_{x_0} be given. Let m be an integer greater than the subscript integer of any internal-state symbol occurring in P_{x_0}. Then if P_{x_0} is modified by adding to it the quadruples $q_m 1 1 q_m$, $q_{m+1} 1 1 q_{m+1}$, . . . , $q_{m+k} 1 1 q_{m+k}$, the partial function determined remains unchanged, since none of the states q_m, . . . , q_{m+k} can be entered. As k varies, this gives $\aleph_0$ distinct sets of instructions for φ_{x_0}. ☒

Theorem III is not an accident of our chosen formal characterization and Gödel numbering. Exercise 2-10 will give it invariant significance.

Observe that the following is, informally, an algorithm for a partial function ψ: given any input $\langle x,y \rangle$, find P_x (by generating the standard effective list of all sets of instructions until P_x is reached), then apply P_x to input y to calculate $\varphi_x(y)$; if and when $\varphi_x(y)$ gets an output, take this as output value for $\psi(x,y)$. Therefore,

$$\psi(x,y) = \begin{cases} \varphi_x(y), & \text{if } \varphi_x(y) \text{ convergent;} \\ \text{divergent}, & \text{if } \varphi_x(y) \text{ divergent.} \end{cases}$$

By appeal to Church's Thesis, we conclude that ψ is partial recursive and that $\psi = \varphi_{x_1}^{(2)}$ for some x_1. (Formal details of P_{x_1} can be found in Davis [1958].) Stating this as a theorem, we have Theorem IV.

Theorem IV *There exists a z such that for all x and y, $\varphi_z(x,y) = \varphi_x(y)$ if $\varphi_x(y)$ is defined, and $\varphi_z(x,y)$ is undefined if $\varphi_x(y)$ is undefined.*

The $\varphi_z^{(2)}$ of Theorem IV is called a *universal partial function* for the partial recursive functions of one variable. P_z is a single Turing machine

which can be used to duplicate *any* partial recursive function of one variable. Theorem IV, sometimes called the *enumeration theorem*, was a main result of the early (and more formal) work on recursive function theory. Clearly, the proof of Theorem IV can be generalized to yield, for each $k \geq 1$, a function of $k + 1$ variables which can serve as a universal partial function in the obvious sense for the partial functions of k variables. Theorem IV is the case $k = 1$.

Theorem IV has a nontrivial practical significance. It shows that, for computing partial functions of one variable, there is a critical degree of "mechanical complexity" (that of P_z) beyond which all further complexity can be absorbed into increased size of program and increased use of memory storage. (In Exercise 2-5, we shall see that the restriction to partial functions of a particular fixed number of variables is not essential.) P_z is called a *universal machine.*

If we compare our formal characterization with questions *6 to *10 given in §1.1, Theorem IV shows that the computing agent L need not be human, and it substantiates our claim in §1.1 that L can be severely limited in its "abilities." Thus our formal Turing-machine concept is compatible with our answers to questions *6 to *9. It is also compatible with our response to *10 in that it makes no restriction of the kind that an affirmative answer to *10 would require. We shall look further at *10 in Theorem XI and in Exercise 2-8.

Theorem V, now to be given, also appears to depend on the particular characterization and Gödel numbering used. Like Theorem III, however, it has invariant significance. In conjunction with Church's Thesis, it will be basic in later work.

Theorem V *For every $m,n \geq 1$, there exists a recursive function s_n^m of $m + 1$ variables such that for all $x, y_1, \ldots, y_m$,*

$$\lambda z_1 \cdots z_n[\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)] = \varphi_{s_n^m(x,y_1,\ldots,y_m)}^{(n)}.$$

Proof. Take the case $m = n = 1$. (Proof is analogous for the other cases.) Consider the family of all partial functions of one variable which are expressible as $\lambda z[\varphi_x^{(2)}(y,z)]$ for various x and y. Using our standard formal characterization for functions of two variables, we can view this as a new formal characterization for a class of partial recursive functions of one variable. By Part III of the Basic Result, there exists a uniform effective procedure for going from sets of instructions in this new characterization to sets of instructions in the old. Hence, by Church's Thesis, there must be a recursive function f of two variables such that

$$\lambda z[\varphi_x^{(2)}(y,z)] = \varphi_{f(x,y)}.$$

This f is our desired s_1^1. ☒

The informal argument by appeal to Church's Thesis and Part III

of the Basic Result can be replaced by a formal proof. (Indeed, the functions s_n^m can be shown to be primitive recursive.) We refer the reader to Davis [1958] and Kleene [1952]. Theorem V is known as the *s-m-n theorem* and is due to Kleene. Theorem V (together with Church's Thesis) is a tool of great range and power. We give one illustrative application here and shall see a further application in §1.9. On many occasions in later chapters the *s-m-n* theorem, like Church's Thesis, will be tacitly used.

Theorem VI *There is a recursive function g of two variables such that for all x,y,*

$$\varphi_{g(x,y)} = \varphi_x\varphi_y \qquad (\text{i.e., } = \lambda z[\varphi_x(\varphi_y(z))]).$$

Proof. By Church's Thesis it is immediate that, for any given x, y, $\eta = \varphi_x\varphi_y$ is a partial recursive function. It remains to get the recursive function g; i.e., we must show that a Gödel number for η can be found in a uniform effective way from x and y as x and y vary. The reader who has reflected on the Basic Result will regard this as quite likely. Theorem V gives a tool for proving it. Define

$$\theta(x,y,z) = \varphi_x(\varphi_y(z)) = \varphi_{x_1}(x,\varphi_{x_1}(y,z)),$$

where φ_{x_1} is the universal function of Theorem IV. By Church's Thesis, θ is partial recursive and has an index w_0. Applying Theorem V, we have

$$\varphi_x\varphi_y = \lambda z[\varphi_{w_0}(x,y,z)] = \varphi_{s_1^2(w_0,x,y)},$$

and $\lambda xy[s_1^2(w_0,x,y)]$ is our desired g.⊠

§1.9 THE HALTING PROBLEM

Is there an effective procedure such that, given any x and y, we can determine whether or not $\varphi_x(y)$ is defined, i.e., whether or not P_x applied to input y yields an output? By Church's Thesis this question can be put in the following equivalent and precise form. Is there a recursive function g such that $g(x,y) = 1$ if $\varphi_x(y)$ is convergent and $g(x,y) = 0$ if $\varphi_x(y)$ is divergent? We answer the question in the following theorem.

Theorem VII *There is no recursive function g such that for all x, y,*

$$g(x,y) = \begin{cases} 1, & \text{if } \varphi_x(y) \text{ convergent;} \\ 0, & \text{if } \varphi_x(y) \text{ divergent.} \end{cases}$$

Informal proof. Assume there is such a recursive g. It can be used to define a new partial function ψ as follows:

$$\psi(x) = \begin{cases} 1, & \text{if } g(x,x) = 0; \\ \text{divergent}, & \text{if } g(x,x) = 1. \end{cases}$$

Since $g(x,x)$ must $= 0$ or 1, this gives an algorithm, and by Church's Thesis,

ψ will be a partial recursive function. (When $g(x,x) = 1$, the instructions for ψ can make use of some infinite cyclic condition—a pair of quadruples $q_n 11 q_n$, $q_n BB q_n$ will suffice—to guarantee that $\psi(x)$ be undefined.) Let y_0 be a Gödel number for ψ. Then, by the definition of ψ, $\varphi_{y_0}(y_0)$ convergent $\Leftrightarrow g(y_0,y_0) = 0$; but, by our initial assumption about g, $g(y_0,y_0) = 0 \Leftrightarrow \varphi_{y_0}(y_0)$ divergent. This is a contradiction, and hence there can be no such recursive g.⊠

The proof just given uses Church's Thesis in a proof by contradiction. The reader may well ask what formal counterpart can exist to the use of Church's Thesis in such "hypothetical" context. It is instructive to trace the path that a more formal proof takes.

More formal proof. Define ψ by

$$\psi(z,x) = \begin{cases} 1, & \text{if } \varphi_z^{(2)}(x,x) = 0; \\ \text{divergent}, & \text{if } \varphi_z^{(2)}(x,x) \neq 0 \text{ or divergent.} \end{cases}$$

By Church's Thesis, ψ is partial recursive. Applying Theorem V, a Gödel number for $\lambda x[\psi(z,x)]$ can be found uniformly effectively from z; i.e., there is a recursive function h such that for all z, $\varphi_{h(z)} = \lambda x[\psi(z,x)]$. $(\lambda z[s_1^1(w_1,z)]$ is such an h, where w_1 is a Gödel number for ψ.)

Now assume $g = \varphi_{z_0}$ for some z_0. By the definition of h, $\varphi_{h(z_0)}(x)$ convergent $\Leftrightarrow \varphi_{z_0}(x,x) = 0$. Substituting $h(z_0)$ for x, we have

$$\varphi_{h(z_0)}(h(z_0)) \text{ convergent} \Leftrightarrow \varphi_{z_0}(h(z_0),h(z_0)) = 0.$$

But then φ_{z_0} cannot be g; for, with input $<h(z_0),h(z_0)>$, φ_{z_0} either is undefined or gives erroneous information.⊠

We see that no partial recursive function can satisfy the requirement for g, and our proof is constructive in the sense that for any z, $h(z)$ provides a specific example where φ_z fails. The following corollary to Theorem VII is immediate.

Corollary VII *There is no recursive function f such that*

$$f(x) = \begin{cases} 1, & \text{if } \varphi_x(x) \text{ convergent;} \\ 0, & \text{if } \varphi_x(x) \text{ divergent.} \end{cases}$$

Proof. By proof of the theorem.⊠

Our original question (in the first paragraph of this section) has been called the *halting problem*, where the word "halting" means "having an output." The fact stated as Theorem VII is known as the *recursive unsolvability of the halting problem.* It has precise content in terms of our formal characterization.† The Basic Result of §1.5 gives this fact fundamental

† "There is no Turing machine which can solve the halting problem (for Turing machines). Indeed, given any candidate 'solver' Turing machine, we can exhibit an instance of the halting problem upon which it fails; in fact, there *is* a single Turing machine which will compute, from the description of any given candidate, an appropriate counterinstance."

significance. The halting problem was one of the first "natural" combinatorial problems to be shown recursively unsolvable. Demonstration of the existence of easily described recursively unsolvable problems is one of the more striking achievements of twentieth-century mathematics. Prior to such demonstration (in the 1930's) many mathematicians were unwilling to concede that there could exist easily described combinatorial problems (such as the halting problem) which had no algorithmic solution. We give a more general discussion of such unsolvable problems in Chapter 2.

The results of the early formal work on recursive function theory (prior to 1940) can be summarized, by and large, under (1) the Basic Result (§1.6); (2) existence of universal functions (§1.8); and (3) unsolvability of the halting problem (§1.9). Turing's basic paper [1936] is chiefly concerned with these matters.

We conclude this section with another unsolvability result. It was first obtained by Kleene [1936], and it is a natural consequence of our discussion of diagonalization in §1.4.

Theorem VIII *There is no effective procedure for deciding, given any x, whether or not φ_x is a total function. That is to say, there is no recursive function f such that*

$$f(x) = \begin{cases} 1, & \text{if } \varphi_x \text{ is total;} \\ 0, & \text{if } \varphi_x \text{ is not total.} \end{cases}$$

Proof. We give an informal proof. The transition to a more formal proof is similar to that in Theorem VII (see Exercise 2-7).

Assume such a recursive f exists. Then define g by

$$g(0) = \mu y[f(y) = 1],$$
$$g(x + 1) = \mu y[y > g(x) \text{ and } f(y) = 1].$$

Since $f(y) = 1$ for infinitely many y (Theorem I), g is a total function. By Church's Thesis, g is recursive. Now define h by

$$h = \lambda x[\varphi_{g(x)}(x) + 1].$$

By our assumption on f, h is total. By Church's Thesis, h is recursive. Let $h = \varphi_{z_0}$. By definition of g, $g^{-1}(z_0)$ is uniquely defined; call it y_0. Then $h(y_0) = \varphi_{g(y_0)}(y_0) + 1$ by our definition of h; but $h(y_0) = \varphi_{g(y_0)}(y_0)$ by our definition of y_0. Since h is total, this is a contradiction. ☒

§1.10 RECURSIVENESS

In §1.5, we formally characterized a class of partial functions, known as the *partial recursive functions*. Henceforth, when we say that a partial function is "effective," "computable," "effectively computable," "recur-

sively computable," "mechanically computable," or "algorithmic," we shall mean that it falls within this class. The property of being a member of this class is called *recursiveness.* (Some mathematicians refer to recursiveness as "general recursiveness"; others reserve the phrase "general recursiveness" for total functions and refer to the recursiveness of partial functions as "partial recursiveness." In either case, "general" serves to emphasize that functions from the broad class of §1.5 are in question rather than functions from a narrower subclass (such as the primitive recursive functions).)

In the present section we discuss further aspects of recursiveness. In particular, we consider (1) extension of the concept to nonnumerical inputs and outputs (*codings*); (2) certain structural features of the partial recursive functions and the relation of these features to question *10 in §1.1 (*the mu operator*); and (3) the nature and possible usefulness of our future theory (*final comments*). Goals and applications of the theory will be discussed in greater detail in Chapter 3.

Codings

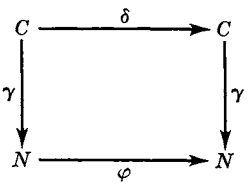
The partial recursive functions are mappings from integers to integers, and their algorithms carry us from notations for integers to notations for integers. The original, unrestricted, informal notion of algorithm concerns procedures with more general kinds of symbolic input and output; e.g., the algorithm for differentiating polynomials. We have already used such unrestricted algorithms as intermediate stages in algorithms defining numerical partial functions (and have then used Church's Thesis to conclude that the numerical partial functions in question were partial recursive functions); e.g., in our definition of a universal function, we used an algorithm (with nonnumerical outputs) that carried us from an integer x to the set of instructions P_x. We now ask: is there any way to include such broader nonnumerical algorithms within our formal theory? Two approaches can be made to this problem.

The first approach is as follows. Given a class of nonnumerical inputs or outputs, choose some fixed one-one mapping from this class into the integers. Henceforth, for theoretical purposes, *identify* each symbolic entity in the nonnumerical class with its corresponding integer "label." Such a standard mapping is called a *coding,* and the integers used as labels are called *code numbers.* The coding is chosen so that (*a*) it is itself given by an informal algorithm in the unrestricted sense; and (*b*) it is *reversible;* i.e., there exists an informal algorithm (in the unrestricted sense) for recognizing code numbers and carrying out the reverse "decoding" mappings from code numbers to nonnumerical entities. Furthermore, it is stipulated that a coding shall be used only when (*c*) an informal algorithm exists for recognizing the expressions that constitute the uncoded, nonnumerical class.

By thus *identifying* expressions (of a nonnumerical class) with integers, we can bring the discussion of algorithms (for such a class) within our

formal theory of algorithms for integers.† We have already seen an example of this in §1.8. Our fixed Gödel numbering for the partial recursive functions is a coding from sets of instructions onto the integers. (Codings are frequently referred to as *Gödel numberings,* and code numbers are then called *Gödel numbers.*‡)

The use of codings raises an immediate question of invariance. Once a coding is chosen, will the formal concept *partial recursive function on code numbers* correspond to the informal notion *algorithmic mapping on the uncoded expressions?* As the latter notion is informal, the answer must be, in part, empirical. Church's Thesis provides an affirmative answer. Let C be the uncoded class (see the diagram below). Let γ be the coding map from C into N, and let us assume, further, that γ takes C *onto* N (modification of the argument, for the case where γ does not map *onto* N, is straightforward.) Let γ^{-1} be the decoding map from N onto C. γ and γ^{-1} are (informally) algorithmic, by our definition of coding. Let φ be any partial recursive function. φ is (both formally and informally) algorithmic. Hence the mapping $\delta = \gamma^{-1}\varphi\gamma$ is an (informally) algorithmic mapping from C into C. Conversely, let δ be any (informally) algorithmic mapping from C into C; then $\varphi = \gamma\delta\gamma^{-1}$ is an (informally) algorithmic partial function, and by Church's Thesis, φ is a partial recursive function. Thus every formal φ has a corresponding informal δ, and every informal δ has a corresponding formal φ.



The second approach to a formal treatment of nonnumerical algorithms is as follows. The *formal characterization* of §1.5 is broadened to include directly, as inputs and outputs, expressions from wider "nonnumerical" classes. The Turing-machine characterization is especially convenient for

† The philosophically minded reader may ask why we take codings to be mappings into *integers* (mathematical objects) rather than into *numerals* for integers (symbolic objects), since algorithms for partial recursive functions must themselves operate on some form of numeral. It is true that use of numerals would be closer to our motivation; however, the distinction is unimportant for our purposes. The use of integers as labels has greater theoretical convenience and is the general practice.

‡ The first use of a coding was made by Gödel, who chose a fixed coding from the formulas of number theory into the integers; he was thus able to study the formulas and proof logic of number theory within number theory itself. An attempt to find self-referential paradoxes via this coding yields the *first incompleteness theorem* of Gödel. Tarski made a similar early construction of this kind.

this purpose.† It requires only that the expressions of the wider classes be expressible as finite strings in a fixed finite alphabet of basic symbols (other than B and 1). The basic operations of Turing machines are extended to include printing and erasure of symbols from this new alphabet. A nonnumerical mapping is then defined to be *recursive* (or *partial recursive*) if a Turing machine exists for carrying it out. After the formal characterization is so broadened, Parts I, II, and III of the Basic Result (§1.6) can themselves be modified and broadened to apply to this broader concept of recursiveness.

The second approach is evidently more direct. We use the first approach, in order to limit our subject matter and to emphasize it as a formal discipline about mappings on integers. For results obtained in this book, it makes no difference which approach is used.

The Mu Operator

The operator μ was defined in the Introduction.

Theorem IX *Let f be a recursive function of $k+1$ variables; then $\lambda x_1 \cdots x_k[\mu y[f(x_1, \ldots, x_k, y) = 1]]$ is a partial recursive function of k variables.*

Proof. Immediate by Church's Thesis; for let

$$\psi = \lambda x_1 \cdots x_k[\mu y[f(x_1, \ldots, x_k, y) = 1]],$$

then to compute $\psi(x_1, \ldots, x_k)$ we need only compute, in succession,

$$f(x_1, \ldots, x_k, 0), f(x_1, \ldots, x_k, 1), f(x_1, \ldots, x_k, 2), \cdots.$$

If and when we find a y such that $f(x_1, \ldots, x_k, y) = 1$, we take it (the first such y) as value. The subcomputations for f always terminate, since f is a recursive function. ☒

Theorem IX is known as the *mu theorem.* In Exercise 2-13 we shall see that Theorem IX does not hold in general when f is replaced by a *partial* recursive function of $k+1$ variables.

There is a sense in which each partial recursive function can be obtained from (total) recursive functions by a single application of μ. Theorem X gives this.

Theorem X *There exist fixed recursive functions p and t of one and three variables, respectively, such that for all z,*

$$\varphi_z = \lambda x[p(\mu y[t(z,x,y) = 1])].$$

† Other extended formal characterizations for this purpose have been studied by Post [1943] and Smullyan [1961] (see also Asser [1960], Curry [1963], and Turing [1936]).

Proof. Define the function s as follows:

$$s(z,x,y,w) = \begin{cases} 1, & \text{if } P_z \text{ with input } x \text{ yields output } y, \text{ in fewer than } w \text{ steps;} \\ 0, & \text{otherwise.} \end{cases}$$

By Church's Thesis, s is recursive. Define p and q by

$$p = \lambda x[\textit{exponent of } 3 \textit{ in prime decomposition of } x + 1],$$
$$q = \lambda x[\textit{exponent of } 2 \textit{ in prime decomposition of } x + 1].$$

Define t by

$$t = \lambda zxy[s(z,x,p(y),q(y))].$$

By Church's Thesis, p, q, and hence t, are recursive. The theorem is now immediate from the definitions of s, p, q, and t.⊠

Theorem X is known as the *Kleene normal-form theorem.* Once Theorem X is established, Theorem IV (the enumeration theorem) follows as a direct consequence. It is possible to show that both p and t are *primitive recursive.* Theorem X can be stated and proved for partial functions of k variables, $k > 1$, by introducing appropriate recursive functions s_k and t_k of $k + 3$ and $k + 2$ variables, respectively.

Corollary X *There exist recursive functions p and t_k such that for all z,*

$$\varphi_z^{(k)} = \lambda x_1 \cdots x_k[p(\mu y[t_k(z,x_1, \ldots ,x_k,y) = 1])].$$

Proof. As just indicated.⊠

A proof of Theorem X can, of course, be based on the Kleene formal characterization. After a somewhat different sequence of definitions, a function $\hat{t}$ can be obtained analogous to our t. The relation

$$T = \{\langle x,y,z\rangle | \hat{t}(x,y,z) = 1\}$$

is called the *Kleene T-predicate,* and the assertion that $\langle x,y,z\rangle \in T$ is commonly abbreviated $T(x,y,z)$. The Kleene T-predicate occurs frequently in the literature.

The question of a stronger form for Theorem X, with p eliminated, naturally arises. Is there a recursive function t^* such that for all z,

$$\varphi_z = \lambda x[\mu y[t^*(z,x,y) = 1]]?$$

In Theorem 2-III, we shall show that no such t^* can exist. In fact, we shall find a partial recursive ψ of one variable such that for no recursive f is it true that $\psi = \lambda x[\mu y[f(x,y) = 1]]$.

The mu theorem emphasizes that our formal characterization permits a computation to take (in effect) the form of an unbounded search for an integer satisfying some given effective condition. The mu theorem is correlated with our failure to give an affirmative answer to question *10 in §1.1. It is a priori possible, of course, that some meaningful positive

answer to question *10 might be deducible from the formal characterization. Unfortunately, this does not turn out to be true. Theorem XI and its proof show that any reasonable affirmative answer would permit a diagonalization leading to contradiction. The proof of Theorem XI is given as Exercise 2-8.

Theorem XI *There is no recursive function f of two variables such that for all x and z: P_z applied to x yields an output $\Leftrightarrow$ P_z applied to x yields an output in fewer than $f(z,x)$ steps.*†

The nonexistence result in Theorem XI is similar to the unsolvability results of Theorems VII and VIII. Like Theorems VII and VIII, it is a consequence of the breadth of the formal notion of recursiveness. Because of such results, some mathematicians have argued that the formal notion of recursiveness is too broad to be a counterpart to their private informal notions of algorithmic computability. Be this as it may, recursiveness does express *one* notion of algorithmic computability and is sufficiently natural to merit further investigation in its own right.

Final Comments

The concept of recursiveness has virtues (breadth and clarity) and defects (unsolvabilities) that are characteristic of the theory now to be developed. Diagonal methods like those in Theorems VII and VIII will play an important role in the theory. It is not inaccurate to say that our theory is, in large part, a "theory of diagonalization."

Our theory is of limited practical usefulness at present. It is concerned with questions of existence or nonexistence of computational methods rather than with questions of efficiency and good design. Questions of the latter kind appear, not in our theory, but in more complex theories based on narrower concepts than recursiveness. Our theory can be viewed as a limiting, asymptotic version of these narrower, more difficult theories. As such, it has some practical value. The inclusive concept of Turing machine, the concept of universal machine (§1.8), and other combinatorial results and methods of our theory have been found useful and suggestive in work on computer programming. Perhaps the most direct practical applications have come from nonexistence (i.e., unsolvability) results, since these results carry over, a fortiori, to narrower theories.

It must be stated, however, that at the present time our theory derives its principal significance from its relevance to pure mathematics. It provides structures which possess considerable intrinsic beauty and naturalness. It gives new, and often deep, insights into other areas. These insights have been especially helpful in mathematical logic, and they have been increasingly useful in more classical areas as well.

† If we use the function s from the proof of Theorem X, Theorem XI can be restated: *there is no recursive f such that for all x, y, z,* $[(\exists w)[s(z,x,y,w) = 1] \Leftrightarrow s(z,x,y,f(z,x)) = 1]$.