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.