I

Recursive Functions

§1.1 The Informal Notion of Algorithm
§1.2 An Example: The Primitive Recursive Functions
§1.3 Extensionality
§1.4 Diagonalization
§1.5 Formal Characterization
§1.6 The Basic Result
§1.7 Church’s Thesis
§1.8 Gödel Numbers, Universality, r-m-n Theorem
§1.9 The Halting Problem
§1.10 Recursiveness

§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 same 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).

1. **$\lambda x \cdot \text{the } x\text{th prime number}.$** (The method of Eratosthenes' sieve is an algorithm here.) (We are assuming Church's lambda notation. To say that $f = \lambda x \cdot \text{the } x\text{th prime number}$ is to say that for all $x$, $f(x) = x\text{th prime number}$.)

2. **$\lambda x \cdot \text{lcm}(x, y).$** (The Euclidean algorithm serves here.)

3. **$\lambda x \cdot \text{the integer } \leq 9 \text{ whose arabic numeral occurs as the } x\text{th digit in the decimal expansion of } \pi = 3.14159 \ldots.$** (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

4. **$\lambda x \cdot \text{n} \cdot \text{ such common algorithms are the substance of elementary school arithmetic.}}$

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 let $L$ 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 inexact, 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 nonontal 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 (non-negative) 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 \( \lambda z[2z] \), can be computed by an arrangement of this kind.† It is possible, however, to make a rather convincing and general argument that the function \( \lambda z[z] \) 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 \textit{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.

§1.2 Primitive recursive functions

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 X1 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 \textit{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