1. Computability

Recursion theory is, at least in its initial stages, the theory of computability. In particular, the first task of recursion theory is to give a rigorous mathematical definition of *computable*.

A *computation* is a process by which we proceed from some initially given objects by means of a fixed set of rules to obtain some final results. The initially given objects are called *inputs*; the fixed set of rules is called an *algorithm*; and the final results are called *outputs*.

We shall always suppose that there is at most one output; for a computation with k outputs can be thought of as k different computations with one output each. On the other hand, we shall allow any finite number of inputs (including zero).

We shall suppose that each algorithm has a fixed number k of inputs. We do not, ever, require that the algorithm give an output when applied to every k–tuple of inputs. In particular, for some k–tuples of inputs the algorithm may go on computing forever without giving an output.

An algorithm with k inputs *computes* a function F defined as follows. A k–tuple of inputs $x_1,\ldots,x_k$ is in the domain of F iff the algorithm has an output when applied to the inputs $x_1,\ldots,x_k$; in this case, $F(x_1,\ldots,x_k)$ is that output. A function is *computable* if there is an algorithm which computes it.

As noted, an algorithm is set of rules for proceeding from the inputs to the output. The algorithm must specify precisely and unambiguously what action is to be taken at each step; and this action must be sufficiently mechanical that it can be done by a suitable computer.

It seems very hard to make these ideas precise. We shall therefore proceed in a different way. We shall give a rigorous definition of a class of functions. It will be clear from the definition that every function in the class is computable. After some study of the class, we shall give arguments to show that every computable function is in the class. If we accept these arguments, we have our rigorous definition of computable.

2. Functions and Relations

We must first decide what inputs and outputs to allow. For the moment, we will take our inputs and outputs to be natural numbers, i.e., non–negative integers. We agree that <u>number</u> means natural number unless otherwise indicated. Lower case Latin letters represent numbers.

We now describe the functions to which the notion of computability applies. Let ω be the set of numbers. For each k, ω^k is the set of k–tuples of numbers. Thus ω^1 is ω, and ω^0 has just one member, the empty tuple. When it is not necessary to specify k, we write $\vec{x}$ for $x_1,\dots,x_k$.

A <u>k–ary</u> <u>function</u> is a mapping of a subset of ω^k into ω. We agree that a <u>function</u> is always a k–ary function for some k. We use capital Latin letters (usually F, G, and H) for functions.

A k–ary function is <u>total</u> if its domain is all of ω^k. A 0–ary total function is clearly determined by its value at the empty tuple. We identify it with this value, so that a 0–ary total function is just a number. A 1–ary total function is called a <u>real</u>. (This terminology comes from set theory, where reals are often identified with real numbers. It will lead to no confusion, since we never deal with real numbers.)

A common type of algorithm has as output a yes or no answer to some question about the inputs. Since we want our outputs to be numbers, we identify the answer yes with the number 0 and the answer no with the number 1. We now describe the objects computed by such algorithms.

A <u>k–ary</u> <u>relation</u> is a subset of ω^k. We use capital Latin letters (generally P, Q, and R) for relations. If R is a relation, we usually write $R(\vec{x})$ for $\vec{x} \in R$. If R is 2–ary, we may also write $x\ R\ y$ for $R(x,y)$.