6

Recursive Functions

The intuitive notion of an effectively computable *function is the notion of a function for which there are definite, explicit rules, following which one could in principle compute its value for any given arguments. This chapter studies an extensive class of effectively computable functions, the* recursively computable, *or simply* recursive, *functions. According to* Church's thesis, *these are in fact* all *the effectively computable functions. Evidence for Church's thesis will be developed in this chapter by accumulating examples of effectively computable functions that turn out to be recursive. The subclass of* primitive recursive *functions is introduced in section 6.1, and the full class of* recursive *functions in section 6.2. The next chapter contains further examples. The discussion of recursive computability in this chapter and the next is entirely independent of the discussion of Turing and abacus computability in the preceding three chapters, but in the chapter after next the three notions of computability will be proved equivalent to each other.*

6.1 Primitive Recursive Functions

Intuitively, the notion of an *effectively computable* function f from natural numbers to natural numbers is the notion of a function for which there is a finite list of instructions that in principle make it possible to determine the value $f(x_1, \ldots, x_n)$ for any arguments $x_1, \ldots, x_n$. The instructions must be so definite and explicit that they require no external sources of information and no ingenuity to execute. But the determination of the value given the arguments need only be possible in principle, disregarding practical considerations of time, expense, and the like: the notion of effective computability is an idealized one.

For purposes of computation, the natural numbers that are the arguments and values of the function must be presented in some system of numerals or other, though the class of functions that is effectively computable will not be affected by the choice of system of numerals. (This is because conversion from one system of numerals to another is itself an effective process that can be carried out according to definite, explicit rules.) Of course, in practice some systems of numerals are easier to work with than others, but that is irrelevant to the idealized notion of effective computability.

For present purposes we adopt a variant of the primeval monadic or tally notation, in which a positive integer n is represented by n strokes. The variation is needed because we want to consider not just positive integers (excluding zero) but the natural numbers

(including zero). We adopt the system in which the number zero is represented by the cipher 0, and a natural number $n > 0$ is represented by the cipher 0 followed by a sequence of n little raised strokes or *accents*. Thus the numeral for one is $0'$, the numeral for two is $0''$, and so on.

Two functions that are extremely easy to compute in this notation are the *zero* function, whose value $z(x)$ is the same, namely zero, for any argument x, and the *successor* function $s(x)$, whose value for any number x is the next larger number. In our special notation we write:

$$\begin{array}{llll} z(0) = 0 & z(0') = 0 & z(0'') = 0 & \cdots \\ s(0) = 0' & s(0') = 0'' & s(0'') = 0''' & \cdots \end{array}$$

To compute the zero function, given any any argument, we simply ignore the argument and write down the symbol 0. To compute the successor function in our special notation, given a number written in that notation, we just add one more accent at the right.

Some other functions it is easy to compute (in *any* notation) are the *identity functions*. We have earlier encountered also the identity function of one argument, id or more fully id^1_1, which assigns to each natural number as argument that same number as value:

$$\mathrm{id}^1_1(x) = x.$$

There are two identity functions of two arguments: id^2_1 and id^2_2. For any pair of natural numbers as arguments, these pick out the first and second, respectively, as values:

$$\mathrm{id}^2_1(x, y) = x \qquad \mathrm{id}^2_2(x, y) = y.$$

In general, for each positive integer n, there are n identity functions of n arguments, which pick out the first, second, ..., and nth of the arguments:

$$\mathrm{id}^n_i(x_1, \ldots, x_i, \ldots, x_n) = x_i.$$

Identity functions are also called *projection functions*. [In terms of analytic geometry, $\mathrm{id}^2_1(x, y)$ and $\mathrm{id}^2_2(x, y)$ are the projections x and y of the point (x, y) to the X-axis and to the Y-axis respectively.]

The foregoing functions—zero, successor, and the various identity functions—are together called the *basic* functions. They can be, so to speak, computed in one step, at least on one way of counting steps.

The stock of effectively computable functions can be enlarged by applying certain processes for defining new functions from old. A first sort of operation, composition, is familiar and straightforward. If f is a function of m arguments and each of $g_1, \ldots, g_m$ is a function of n arguments, then the function obtained by *composition* from $f, g_1, \ldots, g_m$ is the function h where we have

$$\boxed{h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))} \quad \text{(Cn)}$$

One might indicate this in shorthand:

$$h = \mathrm{Cn}[f, g_1, \ldots, g_m].$$

Composition is also called *substitution*.

Clearly, if the functions g_i are all effectively computable and the function f is effectively computable, then so is the function h. The number of steps needed to compute $h(x_1, \ldots, x_n)$ will be the sum of the number of steps needed to compute $y_1 = g_1(x_1, \ldots, x_n)$, the number needed to compute $y_2 = g_2(x_1, \ldots, x_n)$, and so on, plus at the end the number of steps needed to compute $f(y_1, \ldots, y_m)$.

6.1 Example (Constant functions). For any natural number n, let the *constant* function const_n be defined by $\mathrm{const}_n(x) = n$ for all x. Then for each n, const_n can be obtained from the basic functions by finitely many applications of composition. For, const_0 is just the zero function z, and $\mathrm{Cn}[s, z]$ is the function h with $h(x) = s(z(x)) = s(0) = 0' = 1 = \mathrm{const}_1(x)$ for all x, so $\mathrm{const}_1 = \mathrm{Cn}[s, z]$. (Actually, such notations as $\mathrm{Cn}[s, z]$ are genuine function symbols, belonging to the same grammatical category as h, and we could have simply written $\mathrm{Cn}[s, z](x) = s(z(x))$ here rather than the more longwinded 'if $h = \mathrm{Cn}[s, z]$, then $h(x) = z(x)'$'.) Similarly $\mathrm{const}_2 = \mathrm{Cn}[s, \mathrm{const}_1]$, and generally $\mathrm{const}_{n+1} = \mathrm{Cn}[s, \mathrm{const}_n]$.

The examples of effectively computable functions we have had so far are admittedly not very exciting. More interesting examples are obtainable using a different process for defining new functions from old, a process that can be used to define addition in terms of successor, multiplication in terms of addition, exponentiation in terms of multiplication, and so on. By way of introduction, consider addition. The rules for computing this function in our special notation can be stated very concisely in two equations as follows:

$$x + 0 = x \qquad x + y' = (x + y)'.$$

To see how these equations enable us to compute sums consider adding $2 = 0''$ and $3 = 0'''$. The equations tell us:

$$\begin{array}{lll}
0'' + 0''' = (0'' + 0'')' & \text{by 2nd equation} & \text{with } x = 0'' \text{ and } y = 0'' \\
0'' + 0'' = (0'' + 0')' & \text{by 2nd equation} & \text{with } x = 0'' \text{ and } y = 0' \\
0'' + 0' = (0'' + 0)' & \text{by 2nd equation} & \text{with } x = 0'' \text{ and } y = 0 \\
0'' + 0 = 0'' & \text{by 1st equation} & \text{with } x = 0''.
\end{array}$$

Combining, we have the following:

$$\begin{aligned}
0'' + 0''' &= (0'' + 0'')' \\
&= (0'' + 0')'' \\
&= (0'' + 0)''' \\
&= 0''''.
\end{aligned}$$

So the sum is $0'''' = 5$. Thus we use the second equation to reduce the problem of computing $x + y$ to that of computing $x + z$ for smaller and smaller z, until we arrive at $z = 0$, when the first equation tells us directly how to compute $x + 0$.

Similarly, for multiplication we have the rules or equations

$$x \cdot 0 = 0 \qquad x \cdot y' = x + (x \cdot y)$$

which enable us to reduce the computation of a product to the computation of sums, which we know how to compute:

$$\begin{aligned} 0'' \cdot 0''' &= 0'' + (0'' \cdot 0'') \\ &= 0'' + (0'' + (0'' \cdot 0')) \\ &= 0'' + (0'' + (0'' + (0'' \cdot 0))) \\ &= 0'' + (0'' + (0'' + 0)) \\ &= 0'' + (0'' + 0'') \end{aligned}$$

after which we would carry out the computation of the sum in the last line in the way indicated above, and obtain $0'''''$.

Now addition and multiplication are just the first two of a series of arithmetic operations, all of which are effectively computable. The next item in the series is exponentiation. Just as multiplication is repeated addition, so exponentiation is repeated multiplication. To compute x^y, that is, to raise x to the power y, multiply together y xs as follows:

$$x \cdot x \cdot x \cdot \ldots \cdot x \qquad \text{(a row of } y \text{ } x\text{s)}.$$

Conventionally, a product of *no* factors is taken to be 1, so we have the equation

$$x^0 = 0'.$$

For higher powers we have

$$\begin{aligned} x^1 &= x \\ x^2 &= x \cdot x \\ &\vdots \\ x^y &= x \cdot x \cdot \ldots \cdot x && \text{(a row of } y \text{ } x\text{s)} \\ x^{y+1} &= x \cdot x \cdot \ldots \cdot x \cdot x = x \cdot x^y && \text{(a row of } y+1 \text{ } x\text{s)}. \end{aligned}$$

So we have the equation

$$x^{y'} = x \cdot x^y.$$

Again we have two equations, and these enable us to reduce the computation of a power to the computation of products, which we know how to do.

Evidently the next item in the series, *super-exponentiation*, would be defined as follows:

$$x^{x^{x^{\cdot^{\cdot^{\cdot}}}}} \qquad \text{(a stack of } y \text{ } x\text{s)}.$$

The alternative notation $x \uparrow y$ may be used for exponentiation to avoid piling up of superscripts. In this notation the definition would be written as follows:

$$x \uparrow x \uparrow x \uparrow \ldots \uparrow x \qquad \text{(a row of } y \text{ } x\text{s)}.$$

Actually, we need to indicate the grouping here. It is to the right, like this:

$$x \uparrow (x \uparrow (x \uparrow \ldots \uparrow x \ldots))$$

and not to the left, like this:

$$(\ldots((x \uparrow x) \uparrow x) \uparrow \ldots) \uparrow x.$$

For it makes a difference: $3 \uparrow (3 \uparrow 3) = 3 \uparrow (27) = 7\,625\,597\,484\,987$; while $(3 \uparrow 3) \uparrow 3 = 27 \uparrow 3 = 19\,683$. Writing $x \Uparrow y$ for the super-exponential, the equations would be

$$x \Uparrow 0 = 0' \qquad x \Uparrow y' = x \uparrow (x \Uparrow y).$$

The next item in the series, *super-duper-exponentiation*, is analogously defined, and so on.

The process for defining new functions from old at work in these cases is called (*primitive*) *recursion*. As our official format for this process we take the following:

$$\boxed{h(x, 0) = f(x),\ \ h(x, y') = g(x, y, h(x, y))} \ \text{(Pr)}.$$

Where the boxed equations—called the *recursion equations* for the function h—hold, h is said to be definable by (primitive) recursion from the functions f and g. In shorthand,

$$h = \text{Pr}[f, g].$$

Functions obtainable from the basic functions by composition and recursion are called *primitive recursive.*

All such functions are effectively computable. For if f and g are effectively computable functions, then h is an effectively computable function. The number of steps needed to compute $h(x, y)$ will be the sum of the number of steps needed to compute $z_0 = f(x) = h(x, 0)$, the number needed to compute $z_1 = g(x, 0, z_0) = h(x, 1)$, the number needed to compute $z_2 = g(x, 1, z_1) = h(x, 2)$, and so on up to $z_y = g(x, y - 1, z_{y-1}) = h(x, y)$.

The definitions of sum, product, and power we gave above are approximately in our official boxed format. [The main difference is that the boxed format allows one, in computing $h(x, y')$, to apply a function taking x, y, and $h(x, y)$ as arguments. In the examples of sum, product, and power, we never needed to use y as an argument.] By fussing over the definitions we gave above, we can put them exactly into the format (Pr), thus showing addition and multiplication to be primitive recursive.

6.2 Example (The addition or sum function). We start with the definition given by the equations we had above,

$$x + 0 = x \qquad x + y' = (x + y)'.$$

As a step toward reducing this to the boxed format (Pr) for recursion, we replace the ordinary plus sign, written between the arguments, by a sign written out front:

$$\text{sum}(x, 0) = x \qquad \text{sum}(x, y') = \text{sum}(x, y)'.$$

To put these equations in the boxed format (Pr), we must find functions f and g for which we have

$$f(x) = x \qquad g(x, y, —) = s(—)$$

for all natural numbers x, y, and —. Such functions lie ready to hand: $f = \mathrm{id}_1^1$, $g = \mathrm{Cn}[s, \mathrm{id}_3^3]$. In the boxed format we have

$$\mathrm{sum}(x, 0) = \mathrm{id}_1^1(x) \qquad \mathrm{sum}(x, s(y)) = \mathrm{Cn}\big[s, \mathrm{id}_3^3\big](x, y, \mathrm{sum}(x, y))$$

and in shorthand we have

$$\mathrm{sum} = \mathrm{Pr}\big[\mathrm{id}_1^1, \mathrm{Cn}\big[s, \mathrm{id}_3^3\big]\big].$$

6.3 Example (The multiplication or product function). We claim prod = Pr[z, Cn[sum, id_1^3, id_3^3]]. To verify this claim we relate it to the boxed formats (Cn) and (Pr). In terms of (Pr) the claim is that the equations

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = g(x, y, \mathrm{prod}(x, y))$$

hold for all natural numbers x and y, where [setting $h = g$, $f = \mathrm{sum}$, $g_1 = \mathrm{id}_1^3$, $g_2 = \mathrm{id}_3^3$ in the boxed (Cn) format] we have

$$\begin{aligned} g(x_1, x_2, x_3) &= \mathrm{Cn}\big[\mathrm{sum}, \mathrm{id}_1^3, \mathrm{id}_3^3\big](x_1, x_2, x_3) \\ &= \mathrm{sum}\big(\mathrm{id}_1^3(x_1, x_2, x_3), \mathrm{id}_3^3(x_1, x_2, x_3)\big) \\ &= x_1 + x_3 \end{aligned}$$

for all natural numbers x_1, x_2, x_3. Overall, then, the claim is that the equations

$$\mathrm{prod}(x, 0) = z(x) \qquad \mathrm{prod}(x, s(y)) = x + \mathrm{prod}(x, y)$$

hold for all natural numbers x and y, which is true:

$$x \cdot 0 = 0 \qquad x \cdot y' = x + x \cdot y.$$

Our rigid format for recursion serves for functions of two arguments such as sum and product, but we are sometimes going to wish to use such a scheme to define functions of a single argument, and functions of more than two arguments. Where there are three or more arguments $x_1, \ldots, x_n, y$ instead of just the two x, y that appear in (Pr), the modification is achieved by viewing each of the five occurrences of x in the boxed format as shorthand for $x_1, \ldots, x_n$. Thus with $n = 2$ the format is

$$\begin{aligned} h(x_1, x_2, 0) &= f(x_1, x_2) \\ h(x_1, x_2, s(y)) &= g(x_1, x_2, y, h(x_1, x_2, y)). \end{aligned}$$

6.4 Example (The factorial function). The factorial $x!$ for positive x is the product $1 \cdot 2 \cdot 3 \cdot \cdots \cdot x$ of all the positive integers up to and including x, and by convention $0! = 1$. Thus we have

$$\begin{aligned} 0! &= 1 \\ y'! &= y! \cdot y'. \end{aligned}$$

To show this function is recursive we would seem to need a version of the format for recursion with $n = 0$. Actually, however, we can simply define a two-argument function with a *dummy* argument, and then get rid of the dummy argument afterwards by composing with an identity function. For example, in the case of the factorial function we can define

$$\begin{aligned} \text{dummyfac}(x, 0) &= \text{const}_1(x) \\ \text{dummyfac}(x, y') &= \text{dummyfac}(x, y) \cdot y' \end{aligned}$$

so that $\text{dummyfac}(x, y) = y!$ regardless of the value of x, and then define $\text{fac}(y) = \text{dummyfac}(y, y)$. More formally,

$$\text{fac} = \text{Cn}\big[\text{Pr}\big[\text{const}_1, \text{Cn}\big[\text{prod}, \text{id}_3^3, \text{Cn}\big[\text{s}, \text{id}_2^3\big]\big]\big], \text{id}, \text{id}\big].$$

(We leave to the reader the verification of this fact, as well as the conversions of informal-style definitions into formal-style definitions in subsequent examples.)

The example of the factorial function can be generalized.

6.5 Proposition. Let f be a primitive recursive function. Then the functions

$$g(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y) = \sum_{i=0}^{y} f(x, i)$$

$$h(x, y) = f(x, 0) \cdot f(x, 1) \cdot \cdots \cdot f(x, y) = \prod_{i=0}^{y} f(x, i)$$

are primitive recursive.

Proof: We have for the g the recursion equations

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

and similarly for h.

Readers may wish, in the further examples to follow, to try to find definitions of their own before reading ours; and for this reason we give the description of the functions first, and our definitions of them (in informal style) afterwards.

6.6 Example. The exponential or power function.

6.7 Example (The (modified) predecessor function). Define pred (x) to be the predecessor $x - 1$ of x for $x > 0$, and let $\text{pred}(0) = 0$ by convention. Then the function pred is primitive recursive.

6.8 Example (The (modified) difference function). Define $x \dot{-} y$ to be the difference $x - y$ if $x \geq y$, and let $x \dot{-} y = 0$ by convention otherwise. Then the function $\dot{-}$ is primitive recursive.

6.9 Example (The signum functions). Define $\text{sg}(0) = 0$, and $\text{sg}(x) = 1$ if $x > 0$, and define $\overline{\text{sg}}(0) = 1$ and $\overline{\text{sg}}(x) = 0$ if $x > 0$. Then sg and $\overline{\text{sg}}$ are primitive recursive.

Proofs

Example 6.6. $x \uparrow 0 = 1$, $x \uparrow s(y) = x \cdot (x \uparrow y)$, or more formally,

$$\exp = \mathrm{Pr}\big[\mathrm{Cn}[s, z], \mathrm{Cn}\big[\mathrm{prod}, \mathrm{id}_1^3, \mathrm{id}_3^3\big]\big].$$

Example 6.7. $\mathrm{pred}(0) = 0$, $\mathrm{pred}(y') = y$.

Example 6.8. $x \dot{-} 0 = x$, $x \dot{-} y' = \mathrm{pred}(x \dot{-} y)$.

Example 6.9. $\mathrm{sg}(y) = 1 \dot{-} (1 \dot{-} y)$, $\overline{\mathrm{sg}}(y) = 1 \dot{-} y$.

6.2 Minimization

We now introduce one further process for defining new functions from old, which can take us beyond primitive recursive functions, and indeed can take us beyond total functions to partial functions. Intuitively, we consider a *partial* function f to be *effectively computable* if a list of definite, explicit instructions can be given, following which one will, in the case they are applied to any x in the domain of f, arrive after a finite number of steps at the value $f(x)$, but following which one will, in the case they are applied to any x not in the domain of f, go on forever without arriving at any result. This notion applies also to two- and many-place functions.

Now the new process we want to consider is this. Given a function f of $n+1$ arguments, the operation of *minimization* yields a total or partial function h of n arguments as follows:

$$\mathrm{Mn}[f](x_1, \ldots, x_n) = \begin{cases} y & \text{if } f(x_1, \ldots, x_n, y) = 0, \text{ and for all } t < y \\ & \quad f(x_1, \ldots, x_n, t) \text{ is defined and} \neq 0 \\ \text{undefined} & \text{if there is no such } y. \end{cases}$$

If $h = \mathrm{Mn}[f]$ and f is an effectively computable total or partial function, then h also will be such a function. For writing x for $x_1, \ldots, x_n$, we compute $h(x)$ by successively computing $f(x, 0)$, $f(x, 1)$, $f(x, 2)$, and so on, stopping if and when we reach a y with $f(x, y) = 0$. If x is in the domain of h, there will be such a y, and the number of steps needed to compute $h(x)$ will be the sum of the number of steps needed to compute $f(x, 0)$, the number of steps needed to compute $f(x, 1)$, and so on, up through the number of steps needed to compute $f(x, y) = 0$. If x is not in the domain of h, this may be for either of two reasons. On the one hand, it may be that all of $f(x, 0)$, $f(x, 1)$, $f(x, 2)$, ... are defined, but they are all nonzero. On the other hand, it may be that for some i, all of $f(x, 0)$, $f(x, 1)$, ..., $f(x, i-1)$ are defined and nonzero, but $f(x, i)$ is undefined. In either case, the attempt to compute $h(x)$ will involve one in a process that goes on forever without producing a result.

In case f is a total function, we do not have to worry about the second of the two ways in which $\mathrm{Mn}[f]$ may fail to be defined, and the above definition boils down to the following simpler form.

$$\mathrm{Mn}[f](x_1, \ldots, x_n) = \begin{cases} \text{the smallest } y \text{ for which} & \\ \quad f(x_1, \ldots, x_n, y) = 0 & \text{if such a } y \text{ exists} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The total function f is called *regular* if for every $x_1, \ldots, x_n$ there is a y such that $f(x_1, \ldots, x_n, y) = 0$. In case f is a regular function, Mn[f] will be a total function. In fact, if f is a total function, Mn[f] will be total if and only if f is regular.

For example, the product function is regular, since for every x, $x \cdot 0 = 0$; and Mn[prod] is simply the zero function. But the sum function is not regular, since $x + y = 0$ only in case $x = y = 0$; and Mn[sum] is the function that is defined only for 0, for which it takes the value 0, and undefined for all $x > 0$.

The functions that can be obtained from the basic functions z, s, id_i^n by the processes Cn, Pr, and Mn are called the *recursive* (total or partial) *functions*. (In the literature, 'recursive function' is often used to mean more specifically 'recursive *total* function', and 'partial recursive function' is then used to mean 'recursive total or partial function'.) As we have observed along the way, recursive functions are all effectively computable.

The hypothesis that, conversely, all effectively computable total functions are recursive is known as *Church's thesis* (the hypothesis that all effectively computable partial functions are recursive being known as the *extended* version of Church's thesis). The interest of Church's thesis derives largely from the following fact. Later chapters will show that some particular functions of great interest in logic and mathematics are nonrecursive. In order to infer from such a theoretical result the conclusion that such functions are not effectively computable (from which may be inferred the practical advice that logicians and mathematicians would be wasting their time looking for a set of instructions to compute the function), we need assurance that Church's thesis is correct.

At present Church's thesis is, for us, simply an hypothesis. It has been made somewhat plausible to the extent that we have shown a significant number of effectively computable functions to be recursive, but one can hardly on the basis of just these few examples be assured of its correctness. More evidence of the correctness of the thesis will accumulate as we consider more examples in the next two chapters.

Before turning to examples, it may be well to mention that the thesis that every effectively computable total function is *primitive* recursive would simply be erroneous. Examples of recursive total functions that are not primitive recursive are described in the next chapter.

Problems

6.1 Let f be a two-place recursive total function. Show that the following functions are also recursive:
- **(a)** $g(x, y) = f(y, x)$
- **(b)** $h(x) = f(x, x)$
- **(c)** $k_{17}(x) = f(17, x)$ and $k^{17}(x) = f(x, 17)$.

6.2 Let $J_0(a, b)$ be the function coding pairs of positive integers by positive integers that was called J in Example 1.2, and from now on use the name J for the corresponding function coding pairs of natural numbers by natural numbers, so that $J(a, b) = J_0(a + 1, b + 1) - 1$. Show that J is primitive recursive.

6.3 Show that the following functions are primitive recursive:
- **(a)** the *absolute difference* $|x - y|$, defined to be $x - y$ if $y < x$, and $y - x$ otherwise.
- **(b)** the *order characteristic*, $\chi_{\leq}(x, y)$, defined to be 1 if $x \leq y$, and 0 otherwise.
- **(c)** the *maximum* $\max(x, y)$, defined to be the larger of x and y.

6.4 Show that the following functions are primitive recursive:
- **(a)** $c(x, y, z) = 1$ if $yz = x$, and 0 otherwise.
- **(b)** $d(x, y, z) = 1$ if $J(y, z) = x$, and 0 otherwise.

6.5 Define $K(n)$ and $L(n)$ as the first and second entries of the pair coded (under the coding J of the preceding problems) by the number n, so that $J(K(n), L(n)) = n$. Show that the functions K and L are primitive recursive.

6.6 An alternative coding of pairs of numbers by numbers was considered in Example 1.2, based on the fact that every natural number n can be written in one and only one way as 1 less than a power of 2 times an odd number, $n = 2^{k(n)}(2l(n) \dot{-} 1) \dot{-} 1$. Show that the functions k and l are primitive recursive.

6.7 Devise some reasonable way of assigning code numbers to recursive functions.

6.8 Given a reasonable way of coding recursive functions by natural numbers, let $d(x) = 1$ if the one-place recursive function with code number x is defined and has value 0 for argument x, and $d(x) = 0$ otherwise. Show that this function is not recursive.

6.9 Let $h(x, y) = 1$ if the one-place recursive function with code number x is defined for argument y, and $h(x, y) = 0$ otherwise. Show that this function is not recursive.