Computability and Logic

Fifth Edition

GEORGE S. BOOLOS

JOHN P. BURGESS
Princeton University

RICHARD C. JEFFREY

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York
www.cambridge.org
Information on this title: www.cambridge.org/9780521877527

First published in print format 2007

ISBN-13 978-0-511-36668-0 eBook (EBL)
ISBN-10 0-511-36668-X eBook (EBL)

ISBN-13 978-0-521-87752-7 hardback
ISBN-10 0-521-87752-0 hardback

ISBN-13 978-0-521-70146-4 paperback
ISBN-10 0-521-70146-5 paperback

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:

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

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

$$x^0 = 0'.$$

For higher powers we have

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

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:

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

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.

7

Recursive Sets and Relations

In the preceding chapter we introduced the classes of primitive recursive and recursive functions. In this chapter we introduce the related notions of primitive recursive *and* recursive sets *and* relations, *which help provide many more examples of primitive recursive and recursive functions. The basic notions are developed in section 7.1. Section 7.2 introduces the related notion of a* semirecursive *set or relation. The optional section 7.3 presents examples of recursive total functions that are not primitive recursive.*

7.1 Recursive Relations

A set of, say, natural numbers is *effectively decidable* if there is an effective procedure that, applied to a natural number, in a finite amount of time gives the correct answer to the question whether it belongs to the set. Thus, representing the answer 'yes' by 1 and the answer 'no' by 0, a set is effectively decidable if and only if its *characteristic function* is effectively computable, where the characteristic function is the function that takes the value 1 for numbers in the set, and the value 0 for numbers not in the set. A set is called *recursively decidable*, or simply *recursive* for short, if its characteristic function is recursive, and is called *primitive recursive* if its characteristic function is primitive recursive. Since recursive functions are effectively computable, recursive sets are effectively decidable. Church's thesis, according to which all effectively computable functions are recursive, implies that all effectively decidable sets are recursive.

These notions can be generalized to relations. Officially, a two-place *relation* R among natural numbers will be simply a set of ordered pairs of natural numbers, and we write Rxy—or $R(x, y)$ if punctuation seems needed for the sake of readability—interchangeably with $(x, y) \in R$ to indicate that the relation R holds of x and y, which is to say, that the pair (x, y) belongs to R. Similarly, a k-place relation is a set of ordered k-tuples. [In case $k = 1$, a one-place relation on natural numbers ought to be a set of 1-tuples (sequences of length one) of numbers, but we will take it simply to be a set of numbers, not distinguishing in this context between n and (n). We thus write Sx or $S(x)$ interchangeably with $x \in S$.] The *characteristic function* of a k-place relation is the k-argument function that takes the value 1 for a k-tuple if the relation holds of that k-tuple, and the value 0 if it does not; and a relation is *effectively*

decidable if its characteristic function is effectively computable, and is (*primitive*) *recursive* if its characteristic function is (primitive) recursive.

7.1 Example (Identity and order). The identity relation, which holds if and only if $x = y$, is primitive recursive, since a little thought shows its characteristic function is $1 - (\mathrm{sg}(x \dot{-} y) + \mathrm{sg}(y \dot{-} x))$. The strict less-than order relation, which holds if and only if $x < y$, is primitive recursive, since its characteristic function is $\mathrm{sg}(y \dot{-} x)$.

We are now ready to indicate an important process for obtaining new recursive functions from old. What follows is actually a pair of propositions, one about primitive recursive functions, the other about recursive functions (according as one reads the proposition with or without the bracketed word 'primitive'). The same proof works for both propositions.

7.2 Proposition (Definition by cases). Suppose that f is the function defined in the following form:

$$f(x, y) = \begin{cases} g_1(x, y) & \text{if } C_1(x, y) \\ \vdots & \vdots \\ g_n(x, y) & \text{if } C_n(x, y) \end{cases}$$

where $C_1, \ldots, C_n$ are (primitive) recursive relations that are mutually exclusive, meaning that for no x, y do more than one of them hold, and collectively exhaustive, meaning that for any x, y at least one of them holds, and where $g_1, \ldots, g_n$ are (primitive) recursive total functions. Then f is (primitive) recursive.

Proof: Let c_i be the characteristic function of C_i. By recursion, define $h_i(x, y, 0) = 0$, $h_i(x, y, z') = g_i(x, y)$. Let $k_i(x, y) = h_i(x, y, c_i(x, y))$, so $k_i(x, y) = 0$ unless $C_i(x, y)$ holds, in which case $k_i(x, y) = g_i(x, y)$. It follows that $f(x, y) = k_1(x, y) + \ldots + k_n(x, y)$, and f is (primitive) recursive since it is obtainable by primitive recursion and composition from the g_i and the c_i, which are (primitive) recursive by assumption, together with the addition (and identity) functions.

7.3 Example (The maximum and minimum functions). As an example of definition by cases, consider $\max(x, y) =$ the larger of the numbers x, y. This can be defined as follows:

$$\max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{if } x < y \end{cases}$$

or in the official format of the proposition above with $g_1 = \mathrm{id}_1^2$ and $g_2 = \mathrm{id}_2^2$. Similarly, function $\min(x, y) =$ the smaller of x, y is also primitive recursive.

These particular functions, max and min, can also be shown to be primitive recursive in a more direct way (as you were asked to do in the problems at the end of the preceding chapter), but in more complicated examples, definition by cases makes it far easier to establish the (primitive) recursiveness of important functions. This is mainly because there are a variety of processes for defining new relations from old that can be shown to produce new (primitive) recursive relations when applied to (primitive) recursive relations. Let us list the most important of these.

Given a relation $R(y_1, \ldots, y_m)$ and total functions $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$, the relation defined by *substitution* of the f_i in R is the relation $R^*(x_1, \ldots, x_n)$ that holds of $x_1, \ldots, x_n$ if and only if R holds of $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$, or in symbols,

$$R^*(x_1, \ldots, x_n) \leftrightarrow R(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)).$$

If the relation R^* is thus obtained by substituting functions f_i in the relation R, then the characteristic function c^* of R^* is obtainable by composition from the f_i and the characteristic function c of R:

$$c^*(x_1, \ldots, x_n) = c(f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)).$$

Therefore, *the result of substituting recursive total functions in a recursive relation is itself a recursive relation*. (Note that it is important here that the functions be *total*.)

An illustration may make this important notion of substitution clearer. For a given function f, the *graph* relation of f is the relation defined by

$$G(x_1, \ldots, x_n, y) \leftrightarrow f(x_1, \ldots, x_n) = y.$$

Let $f^*(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n)$. Then f^* is recursive if f is, since

$$f^* = \mathrm{Cn}\big[f, \mathrm{id}_1^{n+1}, \ldots, \mathrm{id}_n^{n+1}\big].$$

Now $f(x_1, \ldots, x_n) = y$ if and only if

$$f^*(x_1, \ldots, x_n, y) = \mathrm{id}_{n+1}^{n+1}(x_1, \ldots, x_n, y).$$

Indeed, the latter condition is essentially just a long-winded way of writing the former condition. But this shows that if f is a recursive total function, then the graph relation $f(x_1, \ldots, x_n) = y$ is obtainable from the identity relation $u = v$ by substituting the recursive total functions f^* and id_{n+1}^{n+1}. Thus *the graph relation of a recursive total function is a recursive relation*. More compactly, if less strictly accurately, we can summarize by saying that the graph relation $f(x) = y$ is obtained by substituting the recursive total function f in the identity relation. (This compact, slightly inaccurate manner of speaking, which will be used in future, suppresses mention of the role of the identity functions in the foregoing argument.)

Besides substitution, there are several *logical* operations for defining new relations from old. To begin with the most basic of these, given a relation R, its *negation* or *denial* is the relation S that holds if and only if R does not:

$$S(x_1, \ldots, x_n) \leftrightarrow \sim R(x_1, \ldots, x_n).$$

Given two relations R_1 and R_2, their *conjunction* is the relation S that holds if and only if R_1 holds *and* R_2 holds:

$$S(x_1, \ldots, x_n) \leftrightarrow R_1(x_1, \ldots, x_n) \;\&\; R_2(x_1, \ldots, x_n)$$

while their *disjunction* is the relation S that holds if and only if R_1 holds *or* R_2 holds (or both do):

$$S(x_1, \ldots, x_n) \leftrightarrow R_1(x_1, \ldots, x_n) \vee R_2(x_1, \ldots, x_n).$$

Conjunction and disjunctions of more than two relations are similarly defined. Note that when, in accord with our official definition, relations are considered as sets of k-tuples, the negation is simply the complement, the conjunction the intersection, and the disjunction the union.

Given a relation $R(x_1, \ldots, x_n, u)$, by the relation obtained from R through *bounded universal quantification* we mean the relation S that holds of $x_1, \ldots, x_n, u$ if and only if for all $v < u$, the relation R holds of $x_1, \ldots, x_n, v$. We write

$$S(x_1, \ldots, x_n, u) \leftrightarrow \forall v < u \; R(x_1, \ldots, x_n, v)$$

or more fully:

$$S(x_1, \ldots, x_n, u) \leftrightarrow \forall v(v < u \rightarrow R(x_1, \ldots, x_n, v)).$$

By the relation obtained from R through *bounded existential quantification* we mean the relation S that holds of $x_1, \ldots, x_n, u$ if and only if for some $v < u$, the relation R holds of $x_1, \ldots, x_n, v$. We write

$$S(x_1, \ldots, x_n, u) \leftrightarrow \exists v < u \, R(x_1, \ldots, x_n, v)$$

or more fully:

$$S(x_1, \ldots, x_n, u) \leftrightarrow \exists v(v < u \,\&\, R(x_1, \ldots, x_n, v)).$$

The bounded quantifiers $\forall v \leq u$ and $\exists v \leq u$ are similarly defined.

The following theorem and its corollary are stated for recursive relations (and recursive total functions), but hold equally for primitive recursive relations (and primitive recursive functions), by the same proofs, though it would be tedious for writers and readers alike to include a bracketed '(primitive)' everywhere in the statement and proof of the result.

7.4 Theorem (Closure properties of recursive relations).

(a) A relation obtained by substituting recursive total functions in a recursive relation is recursive.
(b) The graph relation of any recursive total function is recursive.
(c) If a relation is recursive, so is its negation.
(d) If two relations are recursive, then so is their conjunction.
(e) If two relations are recursive, then so is their disjunction.
(f) If a relation is recursive, then so is the relation obtained from it by bounded universal quantification.
(g) If a relation is recursive, then so is the relation obtained from it by bounded existential quantification.

Proof:

(a), (b): These have already been proved.

(c): In the remaining items, we write simply x for $x_1, \ldots, x_n$. The characteristic function c^* of the negation or complement of R is obtainable from the characteristic function c of R by $c^*(x) = 1 \dot{-} c(x)$.

(d), (e): The characteristic function c^* of the conjunction or intersection of R_1 and R_2 is obtainable from the characteristic functions c_1 and c_2 of R_1 and R_2 by $c^*(x) = \min(c_1(x), c_2(x))$, and the characteristic function $c\dagger$ of the disjunction or union is similarly obtainable using max in place of min.

(f), (g): From the characteristic function $c(x, y)$ of the relation $R(x, y)$ the characteristic functions u and e of the relations $\forall v \leq y\ R(x_1, \ldots, x_n, v)$ and $\exists v \leq y\ R(x_1, \ldots, x_n, v)$ are obtainable as follows:

$$u(x, y) = \prod_{i=0}^{y} c(x, i) \qquad e(x, y) = \mathrm{sg}\left(\sum_{i=0}^{y} c(x, i)\right)$$

where the summation $\left(\sum\right)$ and product $\left(\prod\right)$ notation is as in Proposition 6.5. For the product will be 0 if any factor is 0, and will be 1 if and only if all factors are 1; while the sum will be positive if any summand is positive. For the strict bounds $\forall v < y$ and $\exists v < y$ we need only replace y by $y \dot{-} 1$.

7.5 Example (Primality). Recall that a natural number x is prime if $x > 1$ and there do not exist any u, v both $< x$ such that $x = u \cdot v$. The set P of primes is primitive recursive, since we have

$$P(x) \leftrightarrow 1 < x\ \&\ \forall u < x\ \forall v < x(u \cdot v \neq x).$$

Here the relation $1 < x$ is the result of substituting const_1 and id into the relation $y < x$, which we know to be primitive recursive from Example 7.1, and so this relation is primitive recursive by clause (a) of the theorem. The relation $u \cdot v = x$ is the graph of a primitive recursive function, namely, the product function; hence this relation is primitive recursive by clause (b) of the theorem. So P is obtained by negation, bounded universal quantification, and conjunction from primitive recursive relations, and is primitive recursive by clauses (c), (d), and (f) of the theorem.

7.6 Corollary (Bounded minimization and maximization). Given a (primitive) recursive relation R, let

$$\mathrm{Min}[R](x_1, \ldots, x_n, w) = \begin{cases} \text{the smallest } y \leq w \text{ for which} \\ \quad R(x_1, \ldots, x_n, y) & \text{if such a } y \text{ exists} \\ w + 1 & \text{otherwise} \end{cases}$$

and

$$\mathrm{Max}[R](x_1, \ldots, x_n, w) = \begin{cases} \text{the largest } y \leq w \text{ for which} \\ \quad R(x_1, \ldots, x_n, y) & \text{if such a } y \text{ exists} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\mathrm{Min}[R]$ and $\mathrm{Max}[R]$ are (primitive) recursive total functions.

Proof: We give the proof for Min. Write x for $x_1, \ldots, x_n$. Consider the (primitive) recursive relation $\forall t \leq y \sim R(x, t)$, and let c be its characteristic function. If there is a smallest $y \leq w$ such that $R(x, y)$, then abbreviating $c(x, i)$ to $c(i)$ we have

$$c(0) = c(1) = \cdots = c(y-1) = 1 \qquad c(y) = c(y+1) = \cdots = c(w) = 0.$$

So c takes the value 1 for the y numbers $i < y$, and the value 0 thereafter. If there is no such y, then

$$c(0) = c(1) = \cdots = c(w) = 1.$$

So c takes the value 1 for all $w + 1$ numbers $i \leq w$. In either case

$$\mathrm{Min}[R](x, w) = \sum_{i=0}^{w} c(x, i)$$

and is therefore (primitive) recursive. The proof for Max is similar, and is left to the reader.

7.7 Example (Quotients and remainders). Given natural numbers x and y with $y > 0$, there are unique natural numbers q and r such that $x = q \cdot y + r$ and $r < y$. They are called the *quotient* and *remainder* on division of x by y. Let $\mathrm{quo}(x, y)$ be the quotient on dividing x by y if $y > 0$, and set $\mathrm{quo}(x, 0) = 0$ by convention. Let $\mathrm{rem}(x, y)$ be the remainder on dividing x by y if $y > 0$, and set $\mathrm{rem}(x, 0) = x$ by convention. Then quo is primitive recursive, as an application of bounded maximization, since $q \leq x$ and q is the largest number such that $q \cdot y \leq x$.

$$\mathrm{quo}(x, y) = \begin{cases} \text{the largest } z \leq x \text{ such that } y \cdot z \leq x & \text{if} \quad y \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

We apply the preceding corollary (in its version for primitive recursive functions and relations). If we let $Rxyz$ be the relation $y \cdot z \leq x$, then $\mathrm{quo}(x, y) = \mathrm{Max}[R](x, y, x)$, and therefore quo is primitive recursive. Also rem is primitive recursive, since $\mathrm{rem}(x, y) = x \dot{-} (\mathrm{quo}(x, y) \cdot y)$. Another notation for $\mathrm{rem}(x, y)$ is $x \bmod y$.

7.8 Corollary. Suppose that f is a regular primitive function and that there is a primitive recursive function g such that the least y with $f(x_1, \ldots, x_n, y) = 0$ is always less than $g(x_1, \ldots, x_n)$. Then $\mathrm{Mn}[f]$ is not merely recursive but primitive recursive.

Proof: Let $R(x_1, \ldots, x_n, y)$ hold if and only if $f(x_1, \ldots, x_n, y) = 0$. Then

$$\mathrm{Mn}[f](x_1, \ldots, x_n) = \mathrm{Min}[R](x_1, \ldots, x_n, g(x_1, \ldots, x_n)).$$

7.9 Proposition. Let R be an $(n + 1)$-place recursive relation. Define a total or partial function r by

$$r(x_1, \ldots, x_n) = \text{the least } y \text{ such that } R(x_1, \ldots, x_n, y).$$

Then r is recursive.

Proof: The function r is just $\mathrm{Mn}[c]$, where c is the characteristic function of $\sim R$.

Note that if r is a function and R its graph relation, then $r(x)$ is the *only* y such that $R(x, y)$, and therefore *a fortiori* the *least* such y (as well as the *greatest* such y).

So the foregoing proposition tells us that if the graph relation of a function is recursive, the function is recursive. We have not set this down as a numbered corollary because we are going to be getting a stronger result at the beginning of the next section.

7.10 Example (The next prime). Let $f(x) =$ the least y such that $x < y$ and y is prime. The relation

$$x < y \ \& \ y \text{ is prime}$$

is primitive recursive, using Example 7.5. Hence the function f is recursive by the preceding proposition. There is a theorem in Euclid's *Elements* that tells us that for any given number x there exists a prime $y > x$, from which we know that our function f is total. But actually, the proof in Euclid shows that there is a prime $y > x$ with $y \leq x! + 1$. Since the factorial function is primitive recursive, the Corollary 7.8 applies to show that f is actually *primitive* recursive.

7.11 Example (Logarithms). Subtraction, the inverse operation to addition, can take us beyond the natural numbers to negative integers; but we have seen there is a reasonable modified version $\dot{-}$ that stays within the natural numbers, and that it is primitive recursive. Division, the inverse operation to multiplication, can take us beyond the integers to fractional rational numbers; but again we have seen there is a reasonable modified version quo that is primitive recursive. Because the power or exponential function is not commutative, that is, because in general $x^y \neq y^x$, there are *two* inverse operations: the yth root of x is the z such that $z^y = x$, while the base-x logarithm of y is the z such that $x^z = y$. Both can take us beyond the rational numbers to irrational real numbers or even imaginary and complex numbers. But again there is a reasonable modified version, or several reasonable modified versions. Here is one for the logarithms

$$\mathrm{lo}(x, y) = \begin{cases} \text{the greatest } z \leq x \text{ such that } y^z \text{ divides } x & \text{if } x, y > 1 \\ 0 & \text{otherwise} \end{cases}$$

where 'divides x' means 'divides x without remainder'. Clearly if $x, y > 1$ and y^z divides x, z must be (quite a bit) less than x. So we can agrue as in the proof of 7.7 to show that lo is a primitive recursive function. Here is another reasonable modified logarithm function:

$$\mathrm{lg}(x, y) = \begin{cases} \text{the greatest } z \text{ such that } y^z \leq x & \text{if } x, y > 1 \\ 0 & \text{otherwise.} \end{cases}$$

The proof that lg is primitive recursive is left to the reader.

The next series of examples pertain to the coding of finite sequences of natural numbers by single natural numbers. The coding we adopt is based on the fact that each positive integer can be written in one and only one way as a product of powers of larger and larger primes. Specifically:

$$(a_0, a_1, \ldots, a_{n-1}) \text{ is coded by } 2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

where $\pi(n)$ is the nth prime (counting 2 as the 0th). (When we first broached the topic of coding finite sequences by single numbers in section 1.2, we used a slightly different coding. That was because we were then coding finite sequences of positive integers, but now want to code finite sequences of natural numbers.) We state the examples first and invite the reader to try them before we give our own proofs.

7.12 Example (The nth prime). Let $\pi(n)$ be the nth prime, counting 2 as the 0th, so $\pi(0) = 2$, $\pi(1) = 3$, $\pi(2) = 5$, $\pi(3) = 7$, and so on. This function is primitive recursive.

7.13 Example (Length). There is a primitive recursive function lh such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then the value $\mathrm{lh}(s)$ is the *length* of that sequence.

7.14 Example (Entries). There is a primitive recursive function ent such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then for each $i < n$ the value of $\mathrm{ent}(s, i)$ is the ith *entry* in that sequence (counting a_0 as the 0th).

Proofs

Example 7.12. $\pi(0) = 2$, $\pi(x') = f(\pi(x))$, where f is the next prime function of Example 7.10. The form of the definition is similar to that of the factorial function: see Example 6.4 for how to reduce definitions of this form to the official format for recursion.

Example 7.13. $\mathrm{lh}(s) = \mathrm{lo}(s, 2)$ will do, where lo is as in Example 7.11. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

this function yields n.

Example 7.14. $\mathrm{ent}(s, i) = \mathrm{lo}(s, \pi(i + 1))$ will do. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

and i, this function yields a_i.

There are some further examples pertaining to coding, but these will not be needed till a much later chapter, and even then only in a section that is optional reading, so we defer them to the optional final section of this chapter. Instead we turn to another auxiliary notion.

7.2 Semirecursive Relations

Intuitively, a set is (*positively*) *effectively semidecidable* if there is an effective procedure that, applied to any number, will if the number is in the set in a finite amount of time give the answer 'yes', but will if the number is not in the set never give an answer. For instance, the domain of an effectively computable partial function f is always effectively semidecidable: the procedure for determining whether n is in the domain of f is simply to try to compute the value $f(n)$; if and when we succeed, we know that n is in the domain; but if n is not in the domain, we never succeed.

The notion of effective semidecidability extends in the obvious way to relations. When applying the procedure, after any number t of steps of computation, we can tell whether we have obtained the answer 'yes' already, or have so far obtained no

answer. Thus if S is a semidecidable set we have

$$S(x) \leftrightarrow \exists t\ R(x, t)$$

where R is the *effectively decidable* relation 'by t steps of computation we obtain the answer "yes"'. Conversely, if R is an effectively decidable relation of any kind, and S is the relation obtained from R by (unbounded) existential quantification, then S is effectively semidecidable: we can attempt to determine whether n is in S by checking whether $R(n, 0)$ holds, and if not, whether $R(n, 1)$ holds, and if not, whether $R(n, 2)$ holds, and so on. If n is in S, we must eventually find a t such that $R(n, t)$, and will thus obtain the answer 'yes'; but if n is not in S, we go on forever without obtaining an answer.

Thus we may characterize the effectively semidecidable sets as those obtained from two-place effectively decidable relations by existential quantification, and more generally, the n-place effectively semidecidable relations as those obtained from $(n + 1)$-place effectively decidable relations by existential quantification. We define an n-place relation S on natural numbers to be (*positively*) *recursively semidecidable*, or simply *semirecursive*, if it is obtainable from an $(n + 1)$-place recursive relation R by existential quantification, thus:

$$S(x_1, \ldots, x_n) \leftrightarrow \exists y\ R(x_1, \ldots, x_n, y).$$

A y such that R holds of the x_i and y may be called a 'witness' to the relation S holding of the x_i (provided we understand that when the witness is a number rather than a person, a witness only testifies to what is true). Semirecursive relations are effectively semidecidable, and Church's thesis would imply that, conversely, effectively semidecidable relations are semirecursive.

These notions should become clearer as we work out their most basic properties, an exercise that provides an opportunity to review the basic properties of recursive relations. The closure properties of recursive relations established in Theorem 7.4 can be used to establish a similar but not identical list of properties of semirecursive relations.

7.15 Corollary (Closure properties of semirecursive relations).

(a) Any recursive relation is semirecursive.
(b) A relation obtained by substituting recursive total functions in a semirecursive relation is semirecursive.
(c) If two relations are semirecursive, then so is their conjunction.
(d) If two relations are semirecursive, then so is their disjunction.
(e) If a relation is semirecursive, then so is the relation obtained from it by bounded universal quantification.
(f) If a relation is semirecursive, then so is the relation obtained from it by existential quantification.

Proof: We write simply x for $x_1, \ldots, x_n$.

(a): If Rx is a recursive relation, then the relation S given by $Sxy \leftrightarrow (Rx\ \&\ y = y)$ is also recursive, and we have $R(x) \leftrightarrow \exists y\ Sxy$.

(b): If Rx is a semirecursive relation, say $Rx \leftrightarrow \exists y Sxy$ where S is recursive, and if $R^*x \leftrightarrow Rf(x)$, where f is a recursive total function, then the relation S^* given by $S^*xy \leftrightarrow Sf(x)y$ is also recursive, and we have $R^*x \leftrightarrow \exists y\ S^*xy$ and R^* is semirecursive.

(c): If R_1x and R_2x are semirecursive relations, say $R_ix \leftrightarrow \exists y\ S_ixy$ where S_1 and S_2 are recursive, then the relation S given by $Sxw \leftrightarrow \exists y_1 < w\ \exists y_2 < w(S_1xy_1\ \&\ S_2xy_2)$ is also recursive, and we have $(R_1x\ \&\ R_2y) \leftrightarrow \exists w\ Sxw$. We are using here the fact that for any two numbers y_1 and y_2, there is a number w greater than both of them.

(d): If R_i and S_i are as in (c), then the relation S given by $Sxy \leftrightarrow (S_1xy \vee S_2xy)$ is also recursive, and we have $(R_1y \vee R_2x) \leftrightarrow \exists y\ Sxy$.

(e): If Rx is a semirecursive relation, say $Rx \leftrightarrow \exists y\ Sxy$ where S is recursive, and if $R^*x \leftrightarrow \forall u < x Ru$, then the relation S^* given by $S^*xw \leftrightarrow \forall u < x\ \exists y < w\ Suy$ is also recursive, and we have $R^*x \leftrightarrow \exists w\ S^*xw$. We are using here the fact that for any finite number of numbers $y_0, y_1, \ldots, y_x$ there is a number w greater than all of them.

(f): If Rxy is a semirecursive relation, say $Rxy \leftrightarrow \exists z\ Sxyz$ where S is recursive, and if $R^*x \leftrightarrow \exists y\ Rxy$, then the relation S^* given by $S^*xw \leftrightarrow \exists y < w\ \exists z < w\ Sxyz$ is also recursive, and we have $R^*x \leftrightarrow \exists w\ S^*xw$.

The potential for semirecursive relations to yield new recursive relations and functions is suggested by the following propositions. Intuitively, if we have a procedure that will eventually tell us when a number is in a set (but will tell us nothing if it is not), and *also* have a procedure that will eventually tell us when a number is not in a set (but will tell us nothing if it is), then by combining them we can get a procedure that will tell us *whether or not* a number is in the set: apply both given procedures (say by doing a step of the one, then a step of the other, alternately), and eventually one or the other must give us an answer. In jargon, if a set and its complement are both effectively semidecidable, the set is decidable. The next proposition is the formal counterpart of this observation.

7.16 Proposition (Complementation principle, or Kleene's theorem). If a set and its complement are both semirecursive, then the set (and hence also its complement) is recursive.

Proof: If Rx and $\sim Rx$ are both semirecursive, say $Rx \leftrightarrow \exists y\ S^+xy$ and $\sim Rx \leftrightarrow \exists y\ S^-xy$, then the relation S^* given by $S^*xy \leftrightarrow (S^+xy \vee S^-xy)$ is recursive, and if f is the function defined by letting $f(x)$ be the least y such that S^*xy, then f is a recursive total function. But then we have $Rx \leftrightarrow S^+xf(x)$, showing that R is obtainable by substituting a recursive total function in a recursive relation, and is therefore recursive.

7.17 Proposition (First graph principle). If the graph relation of a total or partial function f is semirecursive, then f is a recursive total or partial function.

Proof: Suppose $f(x) = y \leftrightarrow \exists z\, Sxyz$, where S is recursive. We first introduce two auxiliary functions:

$$g(x) = \begin{cases} \text{the least } w \text{ such that} \\ \quad \exists y < w\ \exists z < w\ Sxyz & \text{if such a } w \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$h(x, w) = \begin{cases} \text{the least } y < w \text{ such that} \\ \quad \exists z < w\ Sxyz & \text{if such a } y \text{ exists} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here the relations involved are *recursive*, and not just semirecursive, since they are obtained from S by *bounded*, not unbounded, existential quantification. So g and h are recursive. And a little thought shows that $f(x) = h(x, g(x))$, so f is recursive also.

The converse of the foregoing proposition is also true—the graph relation of a recursive partial function is semirecursive, and hence a total or partial function is recursive if and only if its graph relation is recursive or semirecursive—but we are not at this point in a position to prove it.

An *unavoidable* appeal to Church's thesis is made whenever one passes from a theorem about what is or isn't recursively computable to a conclusion about what is or isn't effectively computable. On the other hand, an *avoidable* or *lazy* appeal to Church's thesis is made whenever, in the proof of a technical theorem, we skip the verification that certain obviously effectively computable functions are recursively computable. Church's thesis is mentioned in connection with omissions of verifications only when writing for comparatively inexperienced readers, who cannot reasonably be expected to be able to fill in the gap for themselves; when writing for the more experienced reader one simply says "proof left to reader" as in similar cases elsewhere in mathematics. The reader who works through the following optional section and/or the optional Chapter 8 and/or the optional sections of Chapter 15 will be well on the way to becoming "experienced" enough to fill in virtually any such gap.

7.3* Further Examples

The list of recursive functions is capable of indefinite extension using the machinery developed so far. We begin with the examples pertaining to coding that were alluded to earlier.

7.18 Example (First and last). There are primitive recursive functions fst and lst such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then fst($s$) and lst($s$) are the first and last entries in that sequence.

7.19 Example (Extension). There is a primitive recursive function ext such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$, then for any b, ext(s, b) codes the *extended* sequence $(a_0, a_1, \ldots, a_{n-1}, b)$.

7.20 Example (Concatenation). There is a primitive recursive function conc such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$ and t codes a sequence $(b_0, b_1, \ldots, b_{m-1})$, then conc (s, t) codes the *concatenation* $(a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{m-1})$ of the two sequences.

Proofs

Example 7.18. $\mathrm{fst}(s) = \mathrm{ent}(s, 0)$ and $\mathrm{lst}(s) = \mathrm{ent}(s, \mathrm{lh}(s) \dot{-} 1)$ will do.

Example 7.19. $\mathrm{ext}(s, b) = 2 \cdot s \cdot \pi(\mathrm{lh}(s) + 1)^b$ will do. Applied to

$$2^n 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}}$$

this function yields

$$2^{n+1} 3^{a_0} 5^{a_1} \cdots \pi(n)^{a_{n-1}} \pi(n+1)^b.$$

Example 7.20. A head-on approach here does not work, and we must proceed a little indirectly, first introducing an auxiliary function such that

$$g(s, t, i) = \text{ the code for } (a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{i-1}).$$

We can then obtain the function we really want as $\mathrm{conc}(s, t) = g(s, t, \mathrm{lh}(t))$. The auxiliary g is obtained by recursion as follows:

$$\begin{aligned} g(s, t, 0) &= s \\ g(s, t, i') &= \mathrm{ext}(g(s, t, i), \mathrm{ent}(t, i)). \end{aligned}$$

Two more we leave entirely to the reader.

7.21 Example (Truncation). There is a primitive recursive function tr such that if s codes a sequence $(a_0, a_1, \ldots, a_{n-1})$ and $m \leq n$, then $\mathrm{tr}(s, m)$ codes the *truncated* sequence $(a_0, a_1, \ldots, a_{m-1})$.

7.22 Example (Substitution). There is a primitive recursive function sub such that if s codes a sequence $(a_1, \ldots, a_k)$, and c and d are any natural numbers, then $\mathrm{sub}(s, c, d)$ codes the sequence that results upon taking s and substituting for any entry that is equal to c the number d instead.

We now turn to examples, promised in the preceding chapter, of recursive total functions that are not primitive recursive.

7.23 Example (The Ackermann function). Let $\ll 0 \gg$ be the operation of addition, $\ll 1 \gg$ the operation of multiplication, $\ll 2 \gg$ the operation of exponentiation, $\ll 3 \gg$ the operation of super-exponentiation, and so on, and let $\alpha(x, y, z) = x \ll y \gg z$ and $\gamma(x) = \alpha(x, x, x)$. Thus

$$\begin{aligned} \gamma(0) &= 0 + 0 = 0 \\ \gamma(1) &= 1 \cdot 1 = 1 \\ \gamma(2) &= 2^2 = 4 \\ \gamma(3) &= 3^{3^3} = 7\,625\,597\,484\,987 \end{aligned}$$

after which the values of $\gamma(x)$ begin to grow very rapidly. A related function δ is determined as follows:

$$\begin{aligned} \beta_0(0) &= 2 & \beta_0(y') &= (\beta_0(y))' \\ \beta_{x'}(0) &= 2 & \beta_{x'}(y') &= \beta_x(\beta_{x'}(y)) \\ \beta(x, y) &= \beta_x(y) \\ \delta(x) &= \beta(x, x). \end{aligned}$$

Clearly each of $\beta_0, \beta_1, \beta_2, \ldots$ is recursive. The proof that β and hence δ are also recursive is outlined in a problem at the end of the chapter. (The proof for α and γ would be similar.) The proof that γ and hence α is not primitive recursive in effect proceeds by showing that one needs to apply recursion at least once to get a function that grows as fast as the addition function, at least twice to get one that grows as fast as the multiplication function, and so on; so that no finite number of applications of recursion (and composition, starting with the zero, successor, and identity functions) can give a function that grows as fast as γ. (The proof for β and δ would be similar.) While it would take us too far afield to give the whole proof here, working through the first couple of cases can give insight into the nature of recursion. We present the first case next and outline the second in the problems at the end of the chapter.

7.24 Proposition. It is impossible to obtain the sum or addition function from the basic functions (zero, successor, and identity) by composition, without using recursion.

Proof: To prove this negative result we claim something positive, that if f belongs to the class of functions that can be obtained from the basic functions using only composition, then there is a positive integer a such that for all $x_1, \ldots, x_n$ we have $f(x_1, \ldots, x_n) < x + a$, where x is the largest of $x_1, \ldots, x_n$. No such a can exist for the addition function, since $(a + 1) + (a + 1) > (a + 1) + a$, so it will follow that the addition function is not in the class in question—provided we can prove our claim. The claim is certainly true for the zero function (with $a = 1$), and for the successor function (with $a = 2$), and for each identity function (with $a = 1$ again). Since every function in the class we are interested in is built up step by step from these functions using composition, it will be enough to show if the claim holds for given functions, it holds for the function obtained from them by composition.

So consider a composition

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

Suppose we know

$$g_i(x_1, \ldots, x_n) < x + a_j \qquad \text{where } x \text{ is the largest of the } x_j$$

and suppose we know

$$f(y_1, \ldots, y_m) < y + b \qquad \text{where } y \text{ is the largest of the } y_i.$$

We want to show there is a c such that

$$h(x_1, \ldots, x_n) < x + c \qquad \text{where } x \text{ is the largest of the } x_j.$$

Let a be the largest of $a_1, \ldots, a_m$. Then where x is the largest of the x_j, we have

$$g_i(x_1, \ldots, x_n) < x + a$$

so if $y_i = g_i(x_1, \ldots, x_n)$, then where y is the largest of the y_i, we have $y < x + a$. And so

$$h(x_1, \ldots, x_n) = f(y_1, \ldots, y_m) < (x + a) + b = x + (a + b)$$

and we may take $c = a + b$.

Problems

7.1 Let R be a two-place primitive recursive, recursive, or semirecursive relation. Show that the following relations are also primitive recursive, recursive, or semirecursive, accordingly:

(a) the *converse* of R, given by $S(x, y) \leftrightarrow R(y, x)$

(b) the *diagonal* of R, given by $D(x) \leftrightarrow R(x, x)$

(c) for any natural number m, the *vertical and horizontal sections* of R at m, given by

$$R_m(y) \leftrightarrow R(m, y) \quad \text{and} \quad R^m(x) \leftrightarrow R(x, m).$$

7.2 Prove that the function lg of Example 7.11 is, as there asserted, primitive recursive.

7.3 For natural numbers, write $u \mid v$ to mean that u divides v without remainder, that is, there is a w such that $u \cdot w = v$. [Thus $u \mid 0$ holds for all u, but $0 \mid v$ holds only for $v = 0$.] We say z is the *greatest common divisor* of x and y, and write $z = \gcd(x, y)$, if $z \mid x$ and $z \mid y$ and whenever $w \mid x$ and $w \mid y$, then $w \leq z$ [except that, by convention, we let $\gcd(0, 0) = 0$]. We say z is the *least common multiple* of x and y, and write $z = \mathrm{lcm}(x, y)$, if $x \mid z$ and $y \mid z$ and whenever $x \mid w$ and $y \mid w$, then $z \leq w$. Show that the functions gcd and lcm are primitive recursive.

7.4 For natural numbers, we say x and y are *relatively prime* if $\gcd(x, y) = 1$, where gcd is as in the preceding problem. The *Euler ϕ-function* $\phi(n)$ is defined as the number of $m < n$ such that $\gcd(m, n) = 1$. Show that ϕ is primitive recursive. More generally, let Rxy be a (primitive) recursive relation, and let $r(x) =$ the number of $y < x$ such that Rxy. Show that r is (primitive) recursive.

7.5 Let A be an infinite recursive set, and for each n, let $a(n)$ be the nth element of A in increasing order (counting the least element as the 0th). Show that the function a is recursive.

7.6 Let f be a (primitive) recursive total function, and let A be the set of all n such that the value $f(n)$ is 'new' in the sense of being different from $f(m)$ for all $m < n$. Show that A is (primitive) recursive.

7.7 Let f be a recursive total function whose range is infinite. Show that there is a one-to-one recursive total function g whose range is the same as that of f.

7.8 Let us define a real number ξ to be *primitive recursive* if the function $f(x) =$ the digit in the $(x + 1)$st place in the decimal expansion of ξ is primitive recursive. [Thus if $\xi = \sqrt{2} = 1.4142\ldots$, then $f(0) = 4$, $f(1) = 1$, $f(2) = 4$, $f(3) = 2$, and so on.] Show that $\sqrt{2}$ is a primitive recursive real number.

7.9 Let $f(n)$ be the nth entry in the infinite sequence 1, 1, 2, 3, 5, 8, 13, 21, ... of *Fibonacci numbers.* Then f is determined by the conditions $f(0) = f(1) = 1$, and $f(n+2) = f(n) + f(n+1)$. Show that f is a primitive recursive function.

7.10 Show that the truncation function of Example 7.21 is primitive recursive.

7.11 Show that the substitution function of Example 7.22 is primitive recursive.

The remaining problems pertain to Example 7.23 *in the optional section* 7.3. *If you are not at home with the method of proof by mathematical induction, you should probably defer these problems until after that method has been discussed in a later chapter.*

7.12 If f and g are n- (and $n+2$)-place primitive recursive functions obtainable from the initial functions (zero, successor, identity) by composition, without use of recursion, we have shown in Proposition 7.24 that there are numbers a and b such that for all $x_1, \ldots, x_n$, y, and z we have

$$f(x_1, \ldots, x_n) < x + a, \qquad \text{where } x \text{ is the largest of } x_1, \ldots, x_n$$

$$g(x_1, \ldots, x_n, y, z) < x + b, \qquad \text{where } x \text{ is the largest of } x_1, \ldots, x_n, y, \text{ and } z.$$

Show now that if $h = \mathrm{Pr}[f, g]$, then there is a number c such that for all $x_1, \ldots, x_n$ and y we have

$$h(x_1, \ldots, x_n, y) < cx + c, \qquad \text{where } x \text{ is the largest of } x_1, \ldots, x_n \text{ and } y.$$

7.13 Show that if f and $g_1, \ldots, g_m$ are functions with the property ascribed to the function h in the preceding problem, and if $j = \mathrm{Cn}[f, g_1, \ldots, g_m]$, then j also has that property.

7.14 Show that the multiplication or product function is not obtainable from the initial functions by composition without using recursion at least twice.

7.15 Let β be the function considered in Example 7.23. Consider a natural number s that codes a sequence $(s_0, \ldots, s_m)$ whose every entry s_i is itself a code for a sequence $(b_{i,0}, \ldots, b_{i,ni})$. Call such an s a β-code if the following conditions are met:

if $i < m$, then $b_{i,0} = 2$
if $j < n_0$, then $b_{0,j+1} = b_{0,j}$
if $i < m$ and $j < n_{i+1}$, then $c = b_{i+1,j} \leq n_i$ and $b_{i+1,j+1} = b_{i,c}$.

Call such an s a β-code *covering* (p, q) if $p \leq m$ and $q \leq n_p$.

(a) Show that if s is a β-code covering (p, q), then $b_{p,q} = \beta(p, q)$.

(b) Show that for every p it is the case that for every q there exists a β-code covering (p, q).

7.16 Continuing the preceding problem, show that the relation *Rspqx*, which we define to hold if and only if s is a β-code covering (p, q) and $b_{p,q} = x$, is a primitive recursive relation.

7.17 Continuing the preceding problem, show that β is a recursive (total) function.