CHAPTER 13

Mu-Recursive Functions

In Chapter 9 we introduced computable functions from a mechanical perspective; the transitions of a Turing machine produced the values of a function. The Church-Turing Thesis asserts that every algorithmically computable function can be realized in this manner, but exactly what functions are Turing computable? In this chapter we will provide an answer to this question and, in doing so, obtain further support for the Church-Turing Thesis.

We now consider computable functions from a macroscopic viewpoint. Rather than focusing on elementary Turing machine operations, functions themselves are the fundamental objects of study. We introduce two families of functions, the primitive recursive functions and the μ-recursive functions. The primitive recursive functions are built from a set of intuitively computable functions using the operations of composition and primitive recursion. The μ-recursive functions are obtained by adding unbounded minimalization, a functional representation of sequential search, to the function building operations.

The computability of the primitive and μ-recursive functions is demonstrated by outlining an effective method for producing the values of the functions. The analysis of effective computation is completed by showing the equivalence of the notions of Turing computability and μ-recursivity. This answers the question posed in the opening paragraph—the functions computable by a Turing machine are exactly the μ-recursive functions.

13.1 Primitive Recursive Functions

A family of intuitively computable number-theoretic functions, known as the primitive recursive functions, is obtained from the basic functions

i) the successor function s: $s(x) = x + 1$
ii) the zero function z: $z(x) = 0$
iii) the projection functions $p_i^{(n)}$: $p_i^{(n)}(x_1, \dots, x_n) = x_i$, $1 \le i \le n$

using operations that construct new functions from functions already in the family. The simplicity of the basic functions supports their intuitive computability. The successor function requires only the ability to add one to a natural number. Computing the zero function is even less complex; the value of the function is zero for every argument. The value of the projection function $p_i^{(n)}$ is simply its ith argument.

The primitive recursive functions are constructed from the basic functions by applications of two operations that preserve computability. The first operation is functional composition (Definition 9.4.2). Let f be defined by the composition of the n-variable function h with the k-variable functions $g_1, g_2, \dots, g_n$. If each of the components of the composition is computable, then the value of $f(x_1, \dots, x_k)$ can be obtained from h and $g_1(x_1, \dots, x_k), g_2(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)$. The computability of f follows from the computability of its constituent functions. The second operation for producing new functions is primitive recursion.

Definition 13.1.1

Let g and h be total number-theoretic functions with n and $n+2$ variables, respectively. The $n+1$-variable function f defined by

i) $f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$
ii) $f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))$

is said to be obtained from g and h by **primitive recursion**.

The x_i's are called the *parameters* of a definition by primitive recursion. The variable y is the *recursive variable.*

The operation of primitive recursion provides its own algorithm for computing the value of $f(x_1, \dots, x_n, y)$ whenever g and h are computable. For a fixed set of parameters $x_1, \dots, x_n$, $f(x_1, \dots, x_n, 0)$ is obtained directly from the function g:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n).$$

The value $f(x_1, \dots, x_n, y+1)$ is obtained from the computable function h using

i) the parameters $x_1, \dots, x_n$,
ii) y, the previous value of the recursive variable, and
iii) $f(x_1, \dots, x_n, y)$, the previous value of the function.

For example, $f(x_1, \dots, x_n, y+1)$ is obtained by the sequence of computations

$$\begin{aligned}
f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n)\\
f(x_1, \dots, x_n, 1) &= h(x_1, \dots, x_n, 0, f(x_1, \dots, x_n, 0))\\
f(x_1, \dots, x_n, 2) &= h(x_1, \dots, x_n, 1, f(x_1, \dots, x_n, 1))\\
&\vdots\\
f(x_1, \dots, x_n, y+1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)).
\end{aligned}$$

Since h is computable, this iterative process can be used to determine $f(x_1, \dots, x_n, y+1)$ for any value of the recursive variable y.

Definition 13.1.2

A function is **primitive recursive** if it can be obtained from the successor, zero, and projection functions by a finite number of applications of composition and primitive recursion.

A function defined by composition or primitive recursion from total functions is itself total. This is an immediate consequence of the definitions of the operations and is left as an exercise. Since the basic primitive recursive functions are total and the operations preserve totality, it follows that all primitive recursive functions are total.

Taken together, composition and primitive recursion provide powerful tools for the construction of functions. The following examples show that arbitrary constant functions, addition, multiplication, and factorial are primitive recursive functions.

Example 13.1.1

The constant functions $c_i^{(n)}(x_1, \dots, x_n) = i$ are primitive recursive. Example 9.4.2 defines the constant functions as the composition of the successor, zero, and projection functions.

□

Example 13.1.2

Let add be the function defined by primitive recursion from the functions $g(x) = x$ and $h(x, y, z) = z + 1$. Then

$$\begin{aligned}
add(x, 0) &= g(x) = x\\
add(x, y+1) &= h(x, y, add(x, y)) = add(x, y) + 1.
\end{aligned}$$

The function add computes the sum of two natural numbers. The definition of $add(x, 0)$ indicates that the sum of any number with zero is the number itself. The latter condition defines the sum of x and $y+1$ as the sum of x and y (the result of add for the previous value of the recursive variable) incremented by one.

The preceding definition establishes that addition is primitive recursive. Both g and h, the components of the definition by primitive recursion, are primitive recursive since $g = p_1^{(1)}$ and $h = s \circ p_3^{(3)}$.

The result of the addition of two natural numbers can be obtained from the primitive recursive definition of add by repeatedly applying the condition $add(x, y + 1) = add(x, y) + 1$ to reduce the value of the recursive variable. For example,

$$\begin{aligned} add(2, 4) &= add(2, 3) + 1 \\ &= (add(2, 2) + 1) + 1 \\ &= ((add(2, 1) + 1) + 1) + 1 \\ &= (((add(2, 0) + 1) + 1) + 1) + 1 \\ &= (((2 + 1) + 1) + 1) + 1 \\ &= 6. \end{aligned}$$

When the recursive variable is zero, the function g is used to initiate the evaluation of the expression. □

Example 13.1.3

Let g and h be the primitive functions $g = z$ and $h = add \circ (p_3^{(3)}, p_1^{(3)})$. Multiplication can be defined by primitive recursion from g and h as follows:

$$\begin{aligned} mult(x, 0) &= g(x) = 0 \\ mult(x, y + 1) &= h(x, y, mult(x, y)) = mult(x, y) + x. \end{aligned}$$

The infix expression corresponding to the primitive recursive definition is the identity $x \cdot (y + 1) = x \cdot y + x$, which follows from the distributive property of addition and multiplication. □

Adopting the convention that a zero-variable function is a constant, we can use Definition 13.1.1 to define one-variable functions using primitive recursion and a two-variable function h. The definition of such a function f has the form

i) $f(0) = n_0$, where $n_0 \in \mathbf{N}$
ii) $f(y + 1) = h(y, f(y))$.

Example 13.1.4

The one-variable factorial function defined by

$$fact(y) = \begin{cases} 1 & \text{if } y = 0 \\ \prod_{i=1}^{y} i & \text{otherwise} \end{cases}$$

is primitive recursive. Let $h(x, y) = mult \circ (p_2^{(2)}, s \circ p_1^{(2)}) = y \cdot (x + 1)$. The factorial function is defined using primitive recursion from h by

$$\begin{aligned} fact(0) &= 1 \\ fact(y + 1) &= h(y, fact(y)) = fact(y) \cdot (y + 1). \end{aligned}$$

Note that the definition uses $y + 1$, the value of the recursive variable. This is obtained by applying the successor function to y, the value provided to the function h.

The evaluation of the function $fact$ for the first five input values illustrates how the primitive recursive definition generates the factorial function.

$$\begin{aligned} fact(0) &= 1 \\ fact(1) &= fact(0) \cdot (0 + 1) = 1 \\ fact(2) &= fact(1) \cdot (1 + 1) = 2 \\ fact(3) &= fact(2) \cdot (2 + 1) = 6 \\ fact(4) &= fact(3) \cdot (3 + 1) = 24 \end{aligned}$$

The factorial function is usually denoted $fact(x) = x!$. □

The primitive recursive functions were defined as a family of intuitively computable functions. The Church-Turing Thesis asserts that these functions must also be computable using our Turing machine approach to functional computation. The Theorem 13.1.3 shows that this is indeed the case.

Theorem 13.1.3

Every primitive recursive function is Turing computable.

Proof. Turing machines that compute the basic functions were constructed in Section 9.2. To complete the proof, it suffices to prove that the Turing computable functions are closed under composition and primitive recursion. The former was established in Section 9.4. All that remains is to show that the Turing computable functions are closed under primitive recursion; that is, if f is defined by primitive recursion from Turing computable functions g and h, then f is Turing computable.

Let g and h be Turing computable functions and let f be the function

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y + 1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

defined from g and h by primitive recursion. Since g and h are Turing computable, there are standard Turing machines G and H that compute them. A composite machine F is constructed to compute f. The computation of $f(x_1, x_2, \ldots, x_n, y)$ begins with tape configuration $B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{y}B$.

1. A counter, initially set to 0, is written to the immediate right of the input. The counter is used to record the value of the recursive variable for the current computation.

The parameters are then written to the right of the counter, producing the tape configuration

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{0}B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB.$$

2. The machine G is run on the final n values of the tape, producing

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{0}B\overline{g(x_1,x_2,\ldots,x_n)}B.$$

The computation of G generates $g(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n, 0)$.

3. The tape now has the form

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i}B\overline{f(x_1,x_2,\ldots,x_n,i)}B.$$

If the counter i is equal to y, the computation of $f(x_1, x_2, \ldots, x_n, y)$ is completed by erasing the initial $n+2$ numbers on the tape and translating the result to tape position one.

4. If $i < y$, the tape is configured to compute the next value of f.

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i+1}B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{i}B\overline{f(x_1,x_2,\ldots,x_n,i)}B$$

The machine H is run on the final $n+2$ values on the tape, producing

$$B\overline{x}_1B\overline{x}_2B\ldots B\overline{x}_nB\overline{y}B\overline{i+1}B\overline{h(x_1,x_2,\ldots,x_n,i,f(x_1,x_2,\ldots,x_n,i))}B,$$

where the rightmost value on the tape is $f(x_1, x_2, \ldots, x_n, i+1)$. The computation continues with the comparison in step 3. ∎

13.2 Some Primitive Recursive Functions

A function is primitive recursive if it can be constructed from the zero, successor, and projection functions by a finite number of applications of composition and primitive recursion. Composition permits g and h, the functions used in a primitive recursive definition, to utilize any function that has previously been shown to be primitive recursive.

Primitive recursive definitions are constructed for several common arithmetic functions. Rather than explicitly detailing the functions g and h, a definition by primitive recursion is given in terms of the parameters, the recursive variable, the previous value of the function, and other primitive recursive functions. Note that the definitions of addition and multiplication are identical to the formal definitions given in Examples 13.1.2 and 13.1.3, with the intermediate step omitted.

Because of the compatibility with the operations of composition and primitive recursion, the definitions in Tables 13.1 and 13.2 are given using the functional notation. The standard infix representations of the binary arithmetic functions, given below the function names, are used in the arithmetic expressions throughout the chapter. The notation "+ 1" denotes the successor operator.

TABLE 13.1 Primitive Recursive Arithmetic Functions

Description	Function	Definition
Addition	$add(x, y)$	$add(x, 0) = x$
	$x + y$	$add(x, y+1) = add(x, y) + 1$
Multiplication	$mult(x, y)$	$mult(x, 0) = 0$
	$x \cdot y$	$mult(x, y+1) = mult(x, y) + x$
Predecessor	$pred(y)$	$pred(0) = 0$
		$pred(y+1) = y$
Proper subtraction	$sub(x, y)$	$sub(x, 0) = x$
	$x \dot{-} y$	$sub(x, y+1) = pred(sub(x, y))$
Exponentation	$exp(x, y)$	$exp(x, 0) = 1$
	x^y	$exp(x, y+1) = exp(x, y) \cdot x$

A primitive recursive predicate is a primitive recursive function whose range is the set $\{0, 1\}$. Zero and one are interpreted as false and true, respectively. The first two predicates in Table 13.2, the sign predicates, specify the sign of the argument. The function sg is true when the argument is positive. The complement of sg, denoted $cosg$, is true when the input is zero. Binary predicates that compare the input can be constructed from the arithmetic functions and the sign predicates using composition.

TABLE 13.2 Primitive Recursive Predicates

Description	Predicate	Definition
Sign	$sg(x)$	$sg(0) = 0$
		$sg(y+1) = 1$
Sign complement	$cosg(x)$	$cosg(0) = 1$
		$cosg(y+1) = 0$
Less than	$lt(x, y)$	$sg(y \dot{-} x)$
Greater than	$gt(x, y)$	$sg(x \dot{-} y)$
Equal to	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
Not equal to	$ne(x, y)$	$cosg(eq(x, y))$

Predicates are functions that exhibit the truth or falsity of a proposition. The logical operations negation, conjunction, and disjunction can be constructed using the arithmetic functions and the sign predicates. Let p_1 and p_2 be two primitive recursive predicates. Logical operations on p_1 and p_2 can be defined as follows:

Predicate	Interpretation
$cosg(p_1)$	not p_1
$p_1 \cdot p_2$	p_1 and p_2
$sg(p_1 + p_2)$	p_1 or p_2

Applying *cosg* to the result of a predicate interchanges the values, yielding the negation of the predicate. This technique was used to define the predicate *ne* from the predicate *eq*. Determining the value of a disjunction begins by adding the truth values of the component predicates. Since the sum is 2 when both of the predicates are true, the disjunction is obtained by composing the addition with *sg*. The resulting predicates are primitive recursive since the components of the composition are primitive recursive.

Example 13.2.1

The equality predicates can be used to explicitly specify the value of a function for a finite set of arguments. For example, f is the identity function for all input values other than 0, 1, and 2:

$$f(x) = \begin{cases} 2 & \text{if } x = 0 \\ 5 & \text{if } x = 1 \\ 4 & \text{if } x = 2 \\ x & \text{otherwise} \end{cases} \qquad \begin{aligned} f(x) = {} & eq(x, 0) \cdot 2 \\ & + eq(x, 1) \cdot 5 \\ & + eq(x, 2) \cdot 4 \\ & + gt(x, 2) \cdot x. \end{aligned}$$

The function f is primitive recursive since it can be written as the composition of primitive recursive functions eq, gt, $\cdot$, and $+$. The four predicates in f are exhaustive and mutually exclusive; that is, one and only one of them is true for any natural number. The value of f is determined by the single predicate that holds for the input. □

The technique presented in the previous example, constructing a function from exhaustive and mutually exclusive primitive recursive predicates, is used to establish the following theorem.

Theorem 13.2.1

Let g be a primitive recursive function and f a total function that is identical to g for all but a finite number of input values. Then f is primitive recursive.

Proof. Let g be primitive recursive and let f be defined by

$$f(x) = \begin{cases} y_1 & \text{if } x = n_1 \\ y_2 & \text{if } x = n_2 \\ \vdots & \\ y_k & \text{if } x = n_k \\ g(x) & \text{otherwise.} \end{cases}$$

The equality predicate is used to specify the values of f for input $n_1, \ldots, n_k$. For all other input values, $f(x) = g(x)$. The predicate obtained by the product

$$ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k)$$

is true whenever the value of f is determined by g. Using these predicates, f can be written

$$\begin{aligned} f(x) = {} & eq(x, n_1) \cdot y_1 + eq(x, n_2) \cdot y_2 + \cdots + eq(x, n_k) \cdot y_k \\ & + ne(x, n_1) \cdot ne(x, n_2) \cdot \cdots \cdot ne(x, n_k) \cdot g(x). \end{aligned}$$

Thus f is also primitive recursive. ■

The order of the variables is an essential feature of a definition by primitive recursion. The initial variables are the parameters and the final variable is the recursive variable. Combining composition and the projection functions permits a great deal of flexibility in specifying the number and order of variables in a primitive recursive function. This flexibility is demonstrated by considering alterations to the variables in a two-variable function.

Theorem 13.2.2

Let $g(x, y)$ be a primitive recursive function. Then the functions obtained by

i) (adding dummy variables) $f(x, y, z_1, z_2, \ldots, z_n) = g(x, y)$
ii) (permuting variables) $f(x, y) = g(y, x)$
iii) (identifying variables) $f(x) = g(x, x)$

are primitive recursive.

Proof. Each of the functions is primitive recursive since it can be obtained from g and the projections by composition as follows:

i) $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$
ii) $f = g \circ (p_2^{(2)}, p_1^{(2)})$
iii) $f = g \circ (p_1^{(1)}, p_1^{(1)})$. ■

Dummy variables are used to make functions with different numbers of variables compatible for composition. The definition of the composition $h \circ (g_1, g_2)$ requires that g_1

and g_2 have the same number of variables. Consider the two-variable function f defined by $f(x, y) = (x \cdot y) + x!$. The constituents of the addition are obtained from a multiplication and a factorial operation. The former function has two variables and the latter has one. Adding a dummy variable to the function *fact* produces a two-variable function *fact*$'$ satisfying *fact*$'(x, y) =$ *fact*$(x) = x!$. Finally, we note that $f = add \circ (mult, fact')$ so that f is also primitive recursive.

13.3 Bounded Operators

The sum of a sequence of natural numbers can be obtained by repeated applications of the binary operation of addition. Addition and projection can be combined to construct a function that adds a fixed number of arguments. For example, the primitive recursive function

$$add \circ (p_1^{(4)}, add \circ (p_2^{(4)}, add \circ (p_3^{(4)}, p_4^{(4)})))$$

returns the sum of its four arguments. This approach cannot be used when the number of summands is variable. Consider the function

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

The number of additions is determined by the input variable y. The function f is called the *bounded sum* of g. The variable i is the index of the summation. Computing a bounded sum consists of three actions: the generation of the summands, binary addition, and the comparison of the index with the input y.

We will prove that the bounded sum of a primitive recursive function is primitive recursive. The technique presented can be used to show that repeated applications of any binary primitive recursive operation is also primitive recursive.

Theorem 13.3.1

Let $g(x_1, \ldots, x_n, y)$ be a primitive recursive function. Then the functions

i) (bounded sum) $f(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$

ii) (bounded product) $f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof. The sum

$$\sum_{i=0}^{y} g(x_1, \ldots, x_n, i)$$

is obtained by adding $g(x_1, \ldots, x_n, y)$ to

$$\sum_{i=0}^{y-1} g(x_1, \ldots, x_n, i).$$

Translating this into the language of primitive recursion, we get

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, 0)$$
$$f(x_1, \ldots, x_n, y+1) = f(x_1, \ldots, x_n, y) + g(x_1, \ldots, x_n, y+1). \quad \blacksquare$$

The bounded operations just introduced begin with index zero and terminate when the index reaches the value specified by the argument y. Bounded operations can be generalized by having the range of the index variable determined by two computable functions. The functions l and u are used to determine the lower and upper bounds of the index.

Theorem 13.3.2

Let g be an $n+1$-variable primitive recursive function and let l and u be n-variable primitive recursive functions. Then the functions

i) $f(x_1, \ldots, x_n) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

ii) $f(x_1, \ldots, x_n) = \prod_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i)$

are primitive recursive.

Proof. Since the lower and upper bounds of the summation are determined by the functions l and u, it is possible that the lower bound may be greater than the upper bound. When this occurs, the result of the summation is assigned the default value zero. The predicate

$$gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n))$$

is true in precisely these instances.

If the lower bound is less than or equal to the upper bound, the summation begins with index $l(x_1, \ldots, x_n)$ and terminates when the index reaches $u(x_1, \ldots, x_n)$. Let g' be the primitive recursive function defined by

$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n)).$$

The values of g' are obtained from those of g and $l(x_1, \ldots, x_n)$:

$$g'(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n, l(x_1, \ldots, x_n))$$
$$g'(x_1, \ldots, x_n, 1) = g(x_1, \ldots, x_n, 1 + l(x_1, \ldots, x_n))$$
$$\vdots$$
$$g'(x_1, \ldots, x_n, y) = g(x_1, \ldots, x_n, y + l(x_1, \ldots, x_n)).$$

By Theorem 13.3.1, the function

$$\begin{aligned} f'(x_1, \ldots, x_n, y) &= \sum_{i=0}^{y} g'(x_1, \ldots, x_n, i) \\ &= \sum_{i=l(x_1,\ldots,x_n)}^{y+l(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i) \end{aligned}$$

is primitive recursive. The generalized bounded sum can be obtained by composing f' with the functions u and l:

$$f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))) = \sum_{i=l(x_1,\ldots,x_n)}^{u(x_1,\ldots,x_n)} g(x_1, \ldots, x_n, i).$$

Multiplying this function by the predicate that compares the upper and lower bounds ensures that the bounded sum returns the default value whenever the lower bound exceeds the upper bound. Thus

$$\begin{aligned} f(x_1, \ldots, x_n) = {} & cosg(gt(l(x_1, \ldots, x_n), u(x_1, \ldots, x_n))) \\ & \cdot f'(x_1, \ldots, x_n, (u(x_1, \ldots, x_n) \dot{-} l(x_1, \ldots, x_n))). \end{aligned}$$

Since each of the constituent functions is primitive recursive, it follows that f is also primitive recursive.

A similar argument can be used to show that the generalized bounded product is primitive recursive. When the lower bound is greater than the upper, the bounded product defaults to one. ■

The value returned by a predicate p designates whether the input satisfies the property represented by p. For fixed values $x_1, \ldots, x_n$,

$$\mu z[p(x_1, \ldots, x_n, z)]$$

is defined to be the smallest natural number z such that $p(x_1, \ldots, x_n, z) = 1$. The notation $\mu z[p(x_1, \ldots, x_n, z)]$ is read "the least z satisfying $p(x_1, \ldots, x_n, z)$." This construction is called the *minimalization* of p, and μz is called the μ-operator. The minimalization of an $n+1$-variable predicate defines an n-variable function

$$f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, z)].$$

An intuitive interpretation of minimalization is that it performs a search over the natural numbers. Initially, the variable z is set to zero. The search sequentially examines the natural numbers until a value of z for which $p(x_1, \ldots, x_n, z) = 1$ is encountered.

Unfortunately, the function obtained by the minimalization of a primitive recursive predicate need not be primitive recursive. In fact, such a function may not even be total. Consider the function

$$f(x) = \mu z[eq(x, z \cdot z)].$$

Using the characterization of minimalization as search, f searches for the first z such that $z^2 = x$. If x is a perfect square, then $f(x)$ returns the square root of x. Otherwise, f is undefined.

By restricting the range over which the minimalization occurs, we obtain a bounded minimalization operator. An $n+1$-variable predicate defines an $n+1$-variable function

$$\begin{aligned} f(x_1, \ldots, x_n, y) &= \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)] \\ &= \begin{cases} z & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \le i < z \le y \\ & \quad \text{and } p(x_1, \ldots, x_n, z) = 1 \\ y+1 & \text{otherwise.} \end{cases} \end{aligned}$$

The bounded μ-operator returns the first natural number z less than or equal to y for which $p(x_1, \ldots, x_n, z) = 1$. If no such value exists, the default value of $y+1$ is assigned. Limiting the search to the range of natural numbers between zero and y ensures the totality of the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)].$$

In fact, the bounded minimalization operator defines a primitive recursive function whenever the predicate is primitive recursive.

Theorem 13.3.3

Let $p(x_1, \ldots, x_n, y)$ be a primitive recursive predicate. Then the function

$$f(x_1, \ldots, x_n, y) = \overset{y}{\mu} z[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

Proof. The proof is given for a two-variable predicate $p(x, y)$ and easily generalizes to n-variable predicates. We begin by defining an auxiliary predicate

$$\begin{aligned} g(x, y) &= \begin{cases} 1 & \text{if } p(x, i) = 0 \text{ for } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases} \\ &= \prod_{i=0}^{y} cosg(p(x, i)). \end{aligned}$$

This predicate is primitive recursive since it is a bounded product of the primitive recursive predicate $cosg \circ p$.

The bounded sum of the predicate g produces the bounded μ-operator. To illustrate the use of g in constructing the minimalization operator, consider a two-variable predicate p with argument n whose values are given in the left column:

$$\begin{array}{lll}
p(n,0)=0 & g(n,0)=1 & \sum_{i=0}^{0} g(n,i)=1 \\
p(n,1)=0 & g(n,1)=1 & \sum_{i=0}^{1} g(n,i)=2 \\
p(n,2)=0 & g(n,2)=1 & \sum_{i=0}^{2} g(n,i)=3 \\
p(n,3)=1 & g(n,3)=0 & \sum_{i=0}^{3} g(n,i)=3 \\
p(n,4)=0 & g(n,4)=0 & \sum_{i=0}^{4} g(n,i)=3 \\
p(n,5)=1 & g(n,5)=0 & \sum_{i=0}^{5} g(n,i)=3 \\
\vdots & \vdots & \vdots
\end{array}$$

The value of g is one until the first number z with $p(n, z) = 1$ is encountered. All subsequent values of g are zero. The bounded sum adds the results generated by g. Thus

$$\sum_{i=0}^{y} g(n,i) = \begin{cases} y+1 & \text{if } z > y \\ z & \text{otherwise.} \end{cases}$$

The first condition also includes the possibility that there is no z satisfying $p(n, z) = 1$. In this case the default value is returned regardless of the specified range.

By the preceding argument, we see that the bounded minimalization of a primitive recursive predicate p is given by the function

$$f(x,y) = \overset{y}{\mu} z[p(x,z)] = \sum_{i=0}^{y} g(x,i),$$

and consequently is primitive recursive. ■

Bounded minimalization $f(y) = \overset{y}{\mu} z[p(x, z)]$ can be thought of as a search for the first value of z in the range 0 to y that makes p true. Example 13.3.1 shows that minimalization can also be used to find first value in a subrange or the largest value z in a specified range that satisfies p.

Example 13.3.1

Let $p(x, z)$ be a primitive recursive predicate. Then the functions

i) $f_1(x, y_0, y)$ = the first value in the range $[y_0, y]$ for which $p(x, z)$ is true,
ii) $f_2(x, y)$ = the second value in the range $[0, y]$ for which $p(x, z)$ is true, and
iii) $f_3(x, y)$ = the largest value in the range $[0, y]$ for which $p(x, z)$ is true

are also primitive recursive. For each of these functions, the default is $y + 1$ if there is no value of z that satisfies the specified condition.

To show that f_1 is primitive recursive, the primitive recursive function ge, greater than or equal to, is used to enforce a lower bound on the value of the function. The predicate $p(x, z) \cdot ge(z, y_0)$ is true whenever $p(x, z)$ is true and z is greater than or equal to y_0. The bounded minimalization

$$f_1(x, y_0, y) = \overset{y}{\mu} z[p(x,z) \cdot ge(z, y_0)],$$

returns the first value in the range $[y_0, y]$ for which $p(x, z)$ is true.

The minimalization $\overset{y}{\mu} z'[p(x, z')]$ is the first value in $[0, y]$ for which $p(x, z)$ is true. The second value that makes $p(x, z)$ true is the first value greater than $\overset{y}{\mu} z'[p(x, z')]$ that satisfies p. Using the preceding technique, the function

$$f_2(x,y) = \overset{y}{\mu} z\big[p(x,z) \cdot gt(z, \overset{y}{\mu} z'[p(x,z')])\big]$$

returns the second value in the range $[0, y]$ for which p is true.

A search for the largest value in the range $[0, y]$ must sequentially examine $y, y - 1, y - 2, \ldots, 1, 0$. The bounded minimalization $\overset{y}{\mu} z[p(x, y \dot{-} z)]$ examines the values in the desired order; when $z = 0$, $p(x, y)$ is tested, when $z = 1$, $p(x, y - 1)$ is tested, and so on. The function $f'(x, y) = y \dot{-} \overset{y}{\mu} z[p(x, y \dot{-} z)]$ returns the largest value less than or equal to y that satisfies p. However, the result of f' is $y \dot{-} (y + 1) = 0$ when no such value exists. A comparison is used to produce the proper default value. The first condition in the function

$$f_3(x,y) = eq(y+1, \overset{y}{\mu} z[p(x,z)]) \cdot (y+1) + neq(y+1, \overset{y}{\mu} z[p(x,z)]) \cdot f'(x,y))$$

returns the default $y + 1$ if there is no value in $[0, y]$ that satisfies p. Otherwise, the largest such value is returned. □

Bounded minimalization can be generalized by computing the upper bound of the search with a function u. If u is primitive recursive, so is the resulting function. The proof is similar to that of Theorem 13.3.2 and is left as an exercise.

Theorem 13.3.4

Let p be an $n+1$-variable primitive recursive predicate and let u be an n-variable primitive recursive function. Then the function

$$f(x_1, \ldots, x_n) = \overset{u(x_1,\ldots,x_n)}{\mu z}[p(x_1, \ldots, x_n, z)]$$

is primitive recursive.

13.4 Division Functions

The fundamental operation of integer division, *div*, is not total. The function $div(x, y)$ returns the quotient, the integer part of the division of x by y, when the second argument is nonzero. The function is undefined when y is zero. Since all primitive recursive functions are total, it follows that *div* is not primitive recursive. A primitive recursive division function *quo* is defined by assigning a default value when the denominator is zero:

$$quo(x, y) = \begin{cases} 0 & \text{if } y = 0 \\ div(x, y) & \text{otherwise.} \end{cases}$$

The division function *quo* is constructed using the primitive recursive operation of multiplication. For values of y other than zero, $quo(x, y) = z$ implies that z satisfies $z \cdot y \leq x < (z+1) \cdot y$. That is, $quo(x, y)$ is the smallest natural number z such that $(z+1) \cdot y$ is greater than x. The search for the value of z that satisfies the inequality succeeds before z reaches x since $(x+1) \cdot y$ is greater than x. The function

$$\overset{x}{\mu z}[gt((z+1) \cdot y,\ x)]$$

determines the quotient of x and y whenever the division is defined. The default value is obtained by multiplying the minimalization by $sg(y)$. Thus

$$quo(x, y) = sg(y) \cdot \overset{x}{\mu z}[gt((z+1) \cdot y,\ x)],$$

where the bound is determined by the primitive recursive function $p_1^{(2)}$. The previous definition demonstrates that *quo* is primitive recursive since it has the form prescribed by Theorem 13.3.4.

The quotient function can be used to define a number of division-related functions and predicates including those given in Table 13.3. The function *rem* returns the remainder of the division of x by y whenever the division is defined. Otherwise, $rem(x, 0) = x$. The predicate *divides* defined by

$$divides(x, y) = \begin{cases} 1 & \text{if } x > 0,\ y > 0, \text{ and } y \text{ is a divisor of } x \\ 0 & \text{otherwise} \end{cases}$$

is true whenever y divides x. By convention, zero is not considered to be divisible by any number. The multiplication by $sg(x)$ in the definition of *divides* in Table 13.3 enforces this condition. The default value of the remainder function guarantees that $divides(x, 0) = 0$.

TABLE 13.3 Primitive Recursive Division Functions

Description	Function	Definition
Quotient	$quo(x, y)$	$sg(y) \cdot \overset{x}{\mu z}[gt((z+1) \cdot y,\ x)]$
Remainder	$rem(x, y)$	$x \dot{-} (y \cdot quo(x, y))$
Divides	$divides(x, y)$	$eq(rem(x, y), 0) \cdot sg(x)$
Number of divisors	$ndivisors(x, y)$	$\sum_{i=0}^{x} divides(x, i)$
Prime	$prime(x)$	$eq(ndivisors(x), 2)$

The generalized bounded sum can be used to count the number of divisors of a number. The upper bound of the sum is obtained from the input by the primitive recursive function $p_1^{(1)}$. This bound is satisfactory since no number greater than x is a divisor of x. A prime number is a number whose only divisors are 1 and itself. The predicate *prime* simply checks if the number of divisors is two.

The predicate prime and bounded minimalization can be used to construct a primitive recursive function pn that enumerates the primes. The value of $pn(i)$ is the ith prime. Thus, $pn(0) = 2,\ pn(1) = 3,\ pn(2) = 5,\ pn(3) = 7, \ldots$. The $x+1$st prime is the first prime number greater than $pn(x)$. Bounded minimalization is ideally suited for performing this type of search. To employ the bounded μ-operator, we must determine an upper bound for the minimalization. By Theorem 13.3.4, the bound may be calculated using the input value x.

Lemma 13.4.1

Let $pn(x)$ denote the xth prime. Then $pn(x+1) \leq pn(x)! + 1$.

Proof. Each of the primes $pn(i)$, $i = 0, 1, \ldots, x$, divides $pn(x)!$. Since a prime cannot divide two consecutive numbers, either $pn(x)! + 1$ is prime or its prime decomposition contains a prime other than $pn(0), pn(1), \ldots, pn(x)$. In either case, $pn(x+1) \leq pn(x)! + 1$. ∎

The bound provided by the preceding lemma is computed by the primitive recursive function $fact(x) + 1$. The xth prime function is obtained by primitive recursion as follows:

$$pn(0) = 2$$

$$pn(x+1) = \overset{fact(pn(x))+1}{\mu z}[prime(z) \cdot gt(z, pn(x))].$$

Let us take a moment to reflect on the consequences of the relationship between the family of primitive recursive functions and Turing computability. By Theorem 13.1.3, every primitive recursive function is Turing computable. Designing Turing machines that explicitly compute functions such as *pn* or *ndivisors* would require a large number of states and a complicated transition function. Using the macroscopic approach to computation, these functions are easily shown to be computable. Without the tedium inherent in constructing complicated Turing machines, we have shown that many useful functions and predicates are Turing computable.

13.5 Gödel Numbering and Course-of-Values Recursion

Many common computations involving natural numbers are not number-theoretic functions. Sorting a sequence of numbers returns a sequence, not a single number. However, there are many sorting algorithms that we consider effective procedures. We now introduce primitive recursive constructions that allow us to perform this type of operation. The essential feature is the ability to encode a sequence of numbers in a single value. The coding scheme utilizes the unique decomposition of a natural number into a product of primes. Such codes are called *Gödel numberings* after German logician Kurt Gödel, who developed the technique.

A sequence $x_0, x_1, \ldots, x_{n-1}$ of n natural numbers is encoded by

$$pn(0)^{x_0+1} \cdot pn(1)^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \cdots \cdot pn(n)^{x_n+1}.$$

Since our numbering begins with zero, the elements of a sequence of length n are numbered $0, 1, \ldots, n-1$. Examples of the Gödel numbering of several sequences are

Sequence	Encoding
1, 2	$2^2 3^3 = 108$
0, 1, 3	$2^1 3^2 5^4 = 11{,}250$
0, 1, 0, 1	$2^1 3^2 5^1 7^2 = 4{,}410$

An encoded sequence of length n is a product of powers of the first n primes. The choice of the exponent $x_i + 1$ guarantees that $pn(i)$ occurs in the encoding even when x_i is zero.

The definition of a function that encodes a fixed number of inputs can be obtained directly from the definition of the Gödel numbering. We let

$$gn_n(x_0, \ldots, x_n) = pn(0)^{x_0+1} \cdot \cdots \cdot pn(n)^{x_n+1} = \prod_{i=0}^{n} pn(i)^{x_i+1}$$

be the $n+1$-variable function that encodes a sequence $x_0, x_1, \ldots, x_n$. The function gn_{n-1} can be used to encode the components of an ordered n-tuple. The Gödel number associated with the ordered pair $[x_0, x_1]$ is $gn_1(x_0, x_1)$.

A decoding function is constructed to retrieve the components of an encoded sequence. The function

$$dec(i, x) = \overset{x}{\mu} z[cosg(divides(x, pn(i)^{z+1}))] \dot{-} 1$$

returns the ith element of the sequence encoded in the Gödel number x. The bounded μ-operator is used to find the power of $pn(i)$ in the prime decomposition of x. The minimalization returns the first value of z for which $pn(i)^{z+1}$ does not divide x. The ith element in an encoded sequence is one less than the power of $pn(i)$ in the encoding. The decoding function $dec(x, i)$ returns zero for every prime $pn(i)$ that does not occur in the prime decomposition of x.

When a computation requires n previously computed values, the Gödel encoding function gn_{n-1} can be used to encode the values. The encoded values can be retrieved when they are needed by the computation.

Example 13.5.1

The Fibonacci numbers are defined as the sequence 0, 1, 1, 2, 3, 5, 8, 13, . . . , where an element in the sequence is the sum of its two predecessors. The function

$$f(0) = 0$$
$$f(1) = 1$$
$$f(y+1) = f(y) + f(y-1) \text{ for } y > 1$$

generates the Fibonacci numbers. This is not a definition by primitive recursion since the computation of $f(y+1)$ utilizes both $f(y)$ and $f(y-1)$. To show that the Fibonacci numbers are generated by a primitive recursive function, the Gödel numbering function gn_1 is used to store the two values as a single number. An auxiliary function h encodes the ordered pair with first component $f(y-1)$ and second component $f(y)$:

$$h(0) = gn_1(0, 1) = 2^1 3^2 = 18$$
$$h(y+1) = gn_1(dec(1, h(y)),\ dec(0, h(y)) + dec(1, h(y))).$$

The initial value of h is the encoded pair $[f(0), f(1)]$. The calculation of $h(y+1)$ begins by producing the components of the subsequent ordered pair

$$[dec(1, h(y)),\ dec(0, h(y)) + dec(1, h(y))] = [f(y),\ f(y-1) + f(y)].$$

Encoding the pair with gn_1 completes the evaluation of $h(y+1)$. This process constructs the sequence of Gödel numbers of the pairs $[f(0), f(1)], [f(1), f(2)], [f(2), f(3)], \ldots$. The primitive recursive function $f(y) = dec(0, h(y))$ extracts the Fibonacci numbers from the first components of the ordered pairs. □

The Gödel numbering functions gn_i encode a fixed number of arguments. A Gödel numbering function can be constructed in which the number of elements to be encoded is computed from the arguments of the function. The approach is similar to that taken in constructing the bounded sum and product operations. The values of a one-variable primitive recursive function f with input $0, 1, \ldots, n$ define a sequence $f(0), f(1), \ldots, f(n)$ of length $n+1$. Using the bounded product, the Gödel numbering function

$$gn_f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} pn(i)^{f(i)+1}$$

encodes the first $y+1$ values of f. The relationship between a function f and its encoding function gn_f is established in Theorem 13.5.1.

Theorem 13.5.1

Let f be an $n+1$-variable function and gn_f the encoding function defined from f. Then f is primitive recursive if, and only if, gn_f is primitive recursive.

Proof. If $f(x_1, \ldots, x_n, y)$ is primitive recursive, then the bounded product

$$gn_f(x_1, \ldots, x_n, y) = \prod_{i=0}^{y} pn(i)^{f(x_1,\ldots,x_n,i)+1}$$

computes the Gödel encoding function. On the other hand, the decoding function can be used to recover the values of f from the Gödel number generated by gn_f:

$$f(x_1, \ldots, x_n, y) = dec(y, gn_f(x_1, \ldots, x_n, y)).$$

Thus f is primitive recursive whenever gn_f is. ■

The primitive recursive functions have been introduced because of their intuitive computability. In a definition by primitive recursion, the computation is permitted to use the result of the function with the previous value of the recursive variable. Consider the function defined by

$$\begin{aligned}
f(0) &= 1\\
f(1) &= f(0)\cdot 1 = 1\\
f(2) &= f(0)\cdot 2 + f(1)\cdot 1 = 3\\
f(3) &= f(0)\cdot 3 + f(1)\cdot 2 + f(2)\cdot 1 = 8\\
f(4) &= f(0)\cdot 4 + f(1)\cdot 3 + f(2)\cdot 2 + f(3)\cdot 1 = 21\\
&\vdots
\end{aligned}$$

The function f can be written as

$$\begin{aligned}
f(0) &= 1\\
f(y+1) &= \sum_{i=0}^{y} f(i)\cdot(y+1-i).
\end{aligned}$$

The definition, as formulated, is not primitive recursive since the computation of $f(y+1)$ utilizes all of the previously computed values. The function, however, is intuitively computable; the definition itself outlines an algorithm by which any value can be calculated.

When the result of a function with recursive variable $y+1$ is defined in terms of $f(0), f(1), \ldots, f(y)$, the function f is said to be defined by course-of-values recursion. Determining the result of a function defined by course-of-values recursion appears to utilize a different number of inputs for each value of the recursive variable. In the preceding example, $f(2)$ requires only $f(0)$ and $f(1)$, while $f(4)$ requires $f(0), f(1), f(2)$, and $f(3)$. No single function can be used to compute both $f(2)$ and $f(4)$ directly from the preceding values since a function is required to have a fixed number of arguments.

Regardless of the value of the recursive variable $y+1$, the preceding results can be encoded in the Gödel number $gn_f(y)$. This observation provides the framework for a formal definition of course-of-values recursion.

Definition 13.5.2

Let g and h be $n+2$-variable total number-theoretic functions, respectively. The $n+1$-variable function f defined by

i) $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$
ii) $f(x_1, \ldots, x_n, y+1) = h(x_1, \ldots, x_n, y, gn_f(x_1, \ldots, x_n, y))$

is said to be obtained from g and h by **course-of-values recursion**.

Theorem 13.5.3

Let f be an $n+1$-variable function defined by course-of-values recursion from primitive recursive functions g and h. Then f is primitive recursive.

Proof. We begin by defining gn_f by primitive recursion directly from the primitive recursive functions g and h.

$$\begin{aligned}
gn_f(x_1, \ldots, x_n, 0) &= 2^{f(x_1,\ldots,x_n,0)+1}\\
&= 2^{g(x_1,\ldots,x_n)+1}\\
gn_f(x_1, \ldots, x_n, y+1) &= gn_f(x_1, \ldots, x_n, y)\cdot pn(y+1)^{f(x_1,\ldots,x_n,y+1)+1}\\
&= gn_f(x_1, \ldots, x_n, y)\cdot pn(y+1)^{h(x_1,\ldots,x_n,y,gn_f(x_1,\ldots,x_n,y))+1}
\end{aligned}$$

The evaluation of $gn_f(x_1, \ldots, x_n, y+1)$ uses only

i) the parameters $x_0, \ldots, x_n$,
ii) y, the previous value of the recursive variable,
iii) $gn_f(x_1, \ldots, x_n, y)$, the previous value of gn_f, and
iv) the primitive recursive functions h, pn, $\cdot$, $+$, and exponentiation.

Thus, the function gn_f is primitive recursive. By Theorem 13.5.1, it follows that f is also primitive recursive. ■

In mechanical terms, the Gödel numbering gives computation the equivalent of unlimited memory. A single Gödel number is capable of storing any number of preliminary results. The Gödel numbering encodes the values $f(x_0, \ldots, x_n, 0)$, $f(x_0, \ldots, x_n, 1), \ldots,$ $f(x_0, \ldots, x_n, y)$ that are required for the computation of $f(x_0, \ldots, x_n, y+1)$. The decoding function provides the connection between the memory and the computation. Whenever a stored value is needed by the computation, the decoding function makes it available.

Example 13.5.2

Let h be the primitive recursive function

$$h(x, y) = \sum_{i=0}^{x} dec(i, y) \cdot (x + 1 - i).$$

The function f, which was defined earlier to introduce course-of-values computation, can be defined by course-of-values recursion from h.

$$f(0) = 1$$

$$f(y+1) = h(y, gn_f(y)) = \sum_{i=0}^{y} dec(i, gn_f(y)) \cdot (y + 1 - i)$$

$$= \sum_{i=0}^{y} f(i) \cdot (y + 1 - i) \qquad \square$$

13.6 Computable Partial Functions

The primitive recursive functions were defined as a family of intuitively computable functions. We have established that all primitive recursive functions are total. Conversely, are all computable total functions primitive recursive? Moreover, should we restrict our analysis of computability to total functions? In this section we will present arguments for a negative response to both of these questions.

We will use a diagonalization argument to establish the existence of a total computable function that is not primitive recursive. The first step is to show that the syntactic structure of the primitive recursive functions allows them to be effectively enumerated. The ability to list the primitive recursive functions permits the construction of a computable function that differs from every function in the list.

Theorem 13.6.1

The set of primitive recursive functions is a proper subset of the set of effectively computable total number-theoretic functions.

Proof. The primitive recursive functions can be represented as strings over the alphabet $\Sigma = \{s, p, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), \circ, :, \langle, \rangle\}$. The basic functions s, z, and $p_i^{(j)}$

are represented by $\langle s \rangle$, $\langle z \rangle$, and $\langle pi(j) \rangle$. The composition $h \circ (g_1, \ldots, g_n)$ is encoded $\langle \langle h \rangle \circ (\langle g_1 \rangle, \ldots, \langle g_n \rangle) \rangle$, where $\langle h \rangle$ and $\langle g_i \rangle$ are the representations of the constituent functions. A function defined by primitive recursion from functions g and h is represented by $\langle \langle g \rangle : \langle h \rangle \rangle$.

The strings in Σ^* can be generated by length: first the null string, followed by strings of length one, length two, and so on. A straightforward mechanical process can be designed to determine whether a string represents a correctly formed primitive recursive function. The enumeration of the primitive recursive functions is accomplished by repeatedly generating a string and determining if it is a syntactically correct representation of a function. The first correctly formed string is denoted f_0, the next f_1, and so on. In the same manner, we can enumerate the one-variable primitive recursive functions. This is accomplished by deleting all n-variable functions, $n > 1$, from the previously generated list. This sequence is denoted $f_0^{(1)}, f_1^{(1)}, f_2^{(1)}, \ldots$.

The total one-variable function

$$g(i) = f_i^{(1)}(i) + 1$$

is effectively computable. The effective enumeration of the one-variable primitive recursive functions establishes the computability of g. The value $g(i)$ is obtained by

i) determining the ith one-variable primitive recursive function $f_i^{(1)}$,
ii) computing $f_i^{(1)}(i)$, and
iii) adding one to $f_i^{(1)}(i)$.

Since each of these steps is effective, we conclude that g is computable. By the familiar diagonalization argument,

$$g(i) \neq f_i^{(1)}(i)$$

for any i. Consequently, g is total and computable but not primitive recursive. ■

Theorem 13.6.1 used diagonalization to demonstrate the existence of computable functions that are not primitive recursive. This can also be accomplished directly by constructing a computable function that is not primitive recursive. The two-variable number-theoretic function, known as *Ackermann's function*, defined by

i) $A(0, y) = y + 1$
ii) $A(x+1, 0) = A(x, 1)$
iii) $A(x+1, y+1) = A(x, A(x+1, y))$

is one such function. The values of A are defined recursively with the basis given in condition (i). A proof by induction on x establishes that A is uniquely defined for every pair of input values (Exercise 22). The computations in Example 13.6.1 illustrate the computability of Ackermann's function.

Example 13.6.1

The values $A(1, 1)$ and $A(3, 0)$ are constructed from the definition of Ackermann's function. The column on the right gives the justification for the substitution.

$$\begin{array}{lll} \text{a)}\ A(1,1) & = A(0, A(1,0)) & \text{(iii)} \\ & = A(0, A(0,1)) & \text{(ii)} \\ & = A(0,2) & \text{(i)} \\ & = 3 & \end{array}$$

$$\begin{array}{lll} \text{b)}\ A(2,1) & = A(1, A(2,0)) & \text{(iii)} \\ & = A(1, A(1,1)) & \text{(ii)} \\ & = A(1,3) & \text{(a)} \\ & = A(0, A(1,2)) & \text{(iii)} \\ & = A(0, A(0, A(1,1))) & \text{(iii)} \\ & = A(0, A(0,3)) & \text{(a)} \\ & = A(0,4) & \text{(i)} \\ & = 5 & \text{(i)} \end{array}$$

□

The values of Ackermann's function exhibit a remarkable rate of growth. By fixing the first variable, Ackermann's function generates the one-variable functions

$$A(1, y) = y + 2$$
$$A(2, y) = 2y + 3$$
$$A(3, y) = 2^{y+3} - 3$$
$$A(4, y) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}} - 3.$$

The number of 2's in the exponential chain in $A(4, y)$ is y. For example, $A(4, 0) = 16 - 3$, $A(4, 1) = 2^{16} - 3$, and $A(4, 2) = 2^{2^{16}} - 3$. The first variable of Ackermann's function determines the rate of growth of the function values. We state, without proof, the following theorem that compares the rate of growth of Ackermann's function with that of the primitive recursive functions.

Theorem 13.6.2

For every one-variable primitive recursive function f, there is some $i \in \mathbf{N}$ such that $f(i) < A(i, i)$.

Clearly, the one-variable function $A(i, i)$ obtained by identifying the variables of A is not primitive recursive. It follows that Ackermann's function is not primitive recursive. If it were, then $A(i, i)$, which can be obtained by the composition $A \circ (p_1^{(1)}, p_1^{(1)})$, would also be primitive recursive.

Is it possible to increase the set of primitive recursive functions, possibly by adding some new basic functions or additional operations, to include all total computable functions? Unfortunately, the answer is no. Regardless of the set of total functions that we consider computable, the diagonalization argument in the proof of Theorem 13.6.1 can be used to show that there is no effective enumeration of all total computable functions. Therefore, we must conclude that the computable functions cannot be effectively generated or that there are computable nontotal functions. If we accept the latter proposition, the contradiction from the diagonalization disappears. The reason we can claim that g is not one of the f_i's is that $g(i) \neq f_i^{(1)}(i)$. If $f_i^{(1)}(i)\uparrow$, then $g(i) = f_i^{(i)}(i) + 1$ is also undefined. If we wish to be able to effectively enumerate the computable functions, it is necessary to include partial functions in the enumeration.

We now consider the computability of partial functions. Since composition and primitive recursion preserve totality, an additional operation is needed to construct partial functions from the basic functions. Minimalization has been informally described as a search procedure. Placing a bound on the range of the natural numbers to be examined ensures that the bounded minimalization operation produces total functions. *Unbounded minimalization* is obtained by performing the search without an upper limit on the set of natural numbers to be considered. The function

$$f(x) = \mu z[eq(x, z \cdot z)]$$

defined by unbounded minimalization returns the square root of x whenever x is a perfect square. Otherwise, the search for the first natural number satisfying the predicate continues ad infinitum. Although eq is a total function, the resulting function f is not. For example, $f(3)\uparrow$. A function defined by unbounded minimalization is undefined for input x whenever the search fails to return a value.

The introduction of partial functions forces us to reexamine the operations of composition and primitive recursion. The possibility of undefined values was considered in the definition of composition. The function $h \circ (g_1, \ldots, g_n)$ is undefined for input $x_1, \ldots, x_k$ if either

i) $g_i(x_1, \ldots, x_k)\uparrow$ for some $1 \leq i \leq n$; or
ii) $g_i(x_1, \ldots, x_k)\downarrow$ for all $1 \leq i \leq n$ and $h(g_1(x_1, \ldots, x_k), \ldots g_n(x_1, \ldots, x_k))\uparrow$.

An undefined value propagates from any of the g_i's to the composite function.

The operation of primitive recursion required both of the defining functions g and h to be total. This restriction is relaxed to permit definitions by primitive recursion using partial functions. Let f be defined by primitive recursion from partial functions g and h.

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

Determining the value of a function defined by primitive recursion is an iterative process. The function f is defined for recursive variable y only if the following conditions are satisfied:

i) $f(x_1, \ldots, x_n, 0)\downarrow$ if $g(x_1, \ldots, x_n)\downarrow$
ii) $f(x_1, \ldots, x_n, y+1)\downarrow$ if $f(x_1, \ldots, x_n, i)\downarrow$ for $0 \le i \le y$ and $h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))\downarrow$.

An undefined value for the recursive variable causes f to be undefined for all the subsequent values of the recursive variable.

With the conventions established for definitions with partial functions, a family of computable partial functions can be defined using the operations composition, primitive recursion, and unbounded minimalization.

Definition 13.6.3

The family of **μ-recursive functions** is defined as follows:

i) The successor, zero, and projection functions are μ-recursive.
ii) If h is an n-variable μ-recursive function and $g_1, \ldots, g_n$ are k-variable μ-recursive functions, then $f = h \circ (g_1, \ldots, g_n)$ is μ-recursive.
iii) If g and h are n and $n+2$-variable μ-recursive functions, then the function f defined from g and h by primitive recursion is μ-recursive.
iv) If $p(x_1, \ldots, x_n, y)$ is a total μ-recursive predicate, then $f = \mu z[p(x_1, \ldots, x_n, z)]$ is μ-recursive.
v) A function is μ-recursive only if it can be obtained from condition (i) by a finite number of applications of the rules in (ii), (iii), and (iv).

Conditions (i), (ii), and (iii) imply that all primitive recursive functions are μ-recursive. Notice that unbounded minimalization is not defined for all predicates, but only for total μ-recursive predicates.

The notion of Turing computability encompasses partial functions in a natural way. A Turing machine computes a partial number-theoretic function f if

i) the computation terminates with result $f(x_1, \ldots, x_n)$ whenever $f(x_1, \ldots, x_n)\downarrow$, and
ii) the computation does not terminate whenever $f(x_1, \ldots, x_n)\uparrow$.

The Turing machine computes the value of the function whenever possible. Otherwise, the computation continues indefinitely.

We will now establish the relationship between the μ-recursive and Turing computable functions. The first step is to show that every μ-recursive function is Turing computable. This is not a surprising result; it simply extends Theorem 13.1.3 to partial functions.

Theorem 13.6.4

Every μ-recursive function is Turing computable.

Proof. Since the basic functions are known to be Turing computable, the proof consists of showing that the Turing computable partial functions are closed under operations of composition, primitive recursion, and unbounded minimalization. The techniques developed in Theorems 9.4.3 and 13.1.3 demonstrate the closure of Turing computable total functions under composition and primitive recursion, respectively. These machines also establish the closure for partial functions. An undefined value in one of the constituent computations causes the entire computation to continue indefinitely.

The proof is completed by showing that the unbounded minimalization of a Turing computable total predicate is Turing computable. Let $f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, y)]$ where $p(x_1, \ldots, x_n, y)$ is a total Turing computable predicate. A Turing machine to compute f can be constructed from P, the machine that computes the predicate p. The initial configuration of the tape is $B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB$.

1. The representation of the number zero is added to the right of the input. The search specified by the minimalization operator begins with the tape configuration
$$B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{0}B.$$
The number to the right of the input, call it j, is the index for the minimalization operator.
2. A working copy of the parameters and j is made, producing the tape configuration
$$B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{j}B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{j}B.$$
3. The machine P is run with the input consisting of the copy of the parameters and j, producing
$$B\overline{x}_1B\overline{x}_2B \ldots B\overline{x}_nB\overline{j}B\overline{p(x_1, x_2, \ldots, x_n, j)}B.$$
4. If $p(x_1, x_2, \ldots, x_n, j) = 1$, the value of the minimalization of p is j. Otherwise, the $p(x_1, x_2, \ldots, x_n, j)$ is erased, j is incremented, and the computation continues with step 2.

A computation terminates at step 4 when the first j for which $p(x_1, \ldots, x_n, j) = 1$ is encountered. If no such value exists, the computation loops indefinitely, indicating that the function f is undefined. ■

13.7 Turing Computability and Mu-Recursive Functions

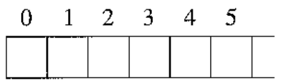
It has already been established that every μ-recursive function can be computed by a Turing machine. We now turn our attention to the opposite inclusion, that every Turing computable function is μ-recursive. To show this, a number-theoretic function is designed to simulate the computations of a Turing machine. The construction of the simulating function requires moving from the domain of machines to the domain of natural numbers. The process of translating machine computations to functions is known as the *arithmetization* of Turing machines.

The arithmetization begins by assigning a number to a Turing machine configuration. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_n)$ be a standard Turing machine that computes a one-variable number-theoretic function f. We will construct a μ-recursive function to numerically simulate the computations of M. The construction easily generalizes to functions of more than one variable.

A configuration of the Turing machine M consists of the state, the position of the tape head, and the segment of the tape from the left boundary to the rightmost nonblank symbol. Each of these components must be represented by a natural number. We will denote the states and tape alphabet by

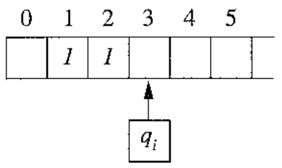
$$Q = \{q_0, q_1, \ldots, q_n\}$$
$$\Gamma = \{B = a_0, 1 = a_1, a_2, \ldots, a_k\}$$

and the numbering will be obtained from the subscripts. Using this numbering, the tape symbols B and 1 are assigned zero and one, respectively. The location of the tape head can be encoded using the numbering of the tape positions.



The symbols on the tape to the rightmost nonblank square form a string over Σ^*. Encoding the tape uses the numeric representation of the elements of the tape alphabet. The string $a_{i_0}a_{i_1}\ldots a_{i_n}$ is encoded by the Gödel number associated with the sequence $i_0, i_1, \ldots, i_n$. The number representing the nonblank tape segment is called the *tape number*.

The tape number of the nonblank segment of the machine configuration



is $2^1 3^2 5^2 = 450$. Explicitly encoding the blank in position three produces $2^1 3^2 5^2 7^1 = 3150$, another tape number representing the tape. Any number of blanks to the right of the rightmost nonblank square may be included in the tape number.

Representing the blank by the number zero permits the correct decoding of any tape position regardless of the segment of the tape encoded in the tape number. If $dec(i, z) = 0$ and $pn(i)$ divides z, then the blank is specifically encoded in the tape number z. On the other hand, if $dec(i, z) = 0$ and $pn(i)$ does not divide z, then position i is to the right of the encoded segment of the tape. Since the tape number encodes the entire nonblank segment of the tape, it follows that position i must be blank.

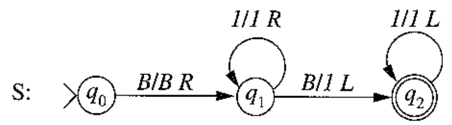
A Turing machine configuration is defined by the state number, tape head position, and tape number. The configuration number incorporates these values into the single number

$$gn_2(\text{state number, tape head position, tape number}),$$

where gn_2 is the Gödel numbering function that encodes ordered triples.

Example 13.7.1

The Turing machine S computes the successor function.



The configuration numbers are given for each configuration produced by the computation of the successor of 1. Recall that the tape symbols B and 1 are assigned the numbers zero and one, respectively.

	State	Position	Tape Number	Configuration Number
q_0B11B	0	0	$2^1 3^2 5^2 = 450$	$gn_2(0, 0, 450)$
$\vdash Bq_111B$	1	1	$2^1 3^2 5^2 = 450$	$gn_2(1, 1, 450)$
$\vdash B1q_11B$	1	2	$2^1 3^2 5^2 = 450$	$gn_2(1, 2, 450)$
$\vdash B11q_1B$	1	3	$2^1 3^2 5^2 7^1 = 3150$	$gn_2(1, 3, 3150)$
$\vdash B1q_211B$	2	2	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 2, 242550)$
$\vdash Bq_2111B$	2	1	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 1, 242550)$
$\vdash q_2B111B$	2	0	$2^1 3^2 5^2 7^2 11^1 = 242550$	$gn_2(2, 0, 242550)$

□

A transition of a standard Turing machine need not alter the tape or the state, but it must move the tape head. The change in the tape head position and the uniqueness of the Gödel numbering ensure that no two consecutive configuration numbers of a computation are identical.

A function tr_M is constructed to trace the computations of a Turing machine M. Tracing a computation means generating the sequence of configuration numbers that correspond to the machine configurations produced by the computation. The value of $tr_M(x, i)$ is the number of the configuration after i transitions when M is run with input x. Since the initial configuration of M is $q_0B\overline{x}B$,

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

The value of $tr_M(x, y+1)$ is obtained by manipulating the configuration number $tr_M(x, y)$ to construct the encoding of the subsequent machine configuration.

The state and symbol in the position scanned by the tape head determine the transition to be applied by the machine M. The primitive recursive functions

$$\begin{aligned} cs(z) &= dec(0, z) \\ ctp(z) &= dec(1, z) \\ cts(z) &= dec(ctp(z), dec(2, z)) \end{aligned}$$

return the state number, tape head position, and the number of the symbol scanned by the tape head from a configuration number z. The position of the tape head is obtained by a direct decoding of the configuration number. The numeric representation of the scanned symbol is encoded as the $ctp(z)$th element of the tape number. The c's in cs, ctp, and cts stand for the components of the current configuration: current state, current tape position, and current tape symbol.

A transition specifies the alterations to the machine configuration and, hence, the configuration number. A transition of M is written

$$\delta(q_i, b) = [q_j, c, d],$$

where $q_i, q_j \in Q$; $b, c \in \Gamma$; and $d \in \{R, L\}$. Functions are defined to simulate the effects of a transition of M. We begin by listing the transitions of M:

$$\begin{aligned} &\delta(q_{i_0}, b_0) = [q_{j_0}, c_0, d_0] \\ &\delta(q_{i_1}, b_1) = [q_{j_1}, c_1, d_1] \\ &\qquad\vdots \\ &\delta(q_{i_m}, b_m) = [q_{j_m}, c_m, d_m]. \end{aligned}$$

The determinism of the machine ensures that the arguments of the transitions are distinct.

The "new state" function

$$ns(z) = \begin{cases} j_0 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ j_1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \vdots & \qquad\vdots \\ j_m & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ cs(z) & \text{otherwise} \end{cases}$$

returns the number of the state entered by a transition from a configuration with configuration number z. The conditions on the right indicate the appropriate transition. Letting $n(b)$ denote the number of the tape symbol b, the first condition can be interpreted, "If the number of the current state is i_0 (state q_{i_0}) and the current tape symbol is b_0 (number $n(b_0)$), then the new state number has number j_0 (state q_{j_0})." This is a direct translation of the initial transition into the numeric representation. Each transition of M defines one condition in ns.

The final condition indicates that the new state is the same as the current state if there is no transition that matches the state and input symbol, that is, if M halts. The conditions define a set of exhaustive and mutually exclusive primitive recursive predicates. Thus, $ns(z)$ is primitive recursive. A function nts that computes the number of the new tape symbol can be defined in a completely analogous manner.

A function that computes the new tape head position alters the number of the current position as specified by the direction in the transition. The transitions designate the directions as L (left) or R (right). A movement to the left subtracts one from the current position number and a movement to the right adds one. To numerically represent the direction we use the notation

$$n(d) = \begin{cases} 0 & \text{if } d = L \\ 2 & \text{if } d = R. \end{cases}$$

The new tape position is computed by

$$ntp(z) = \begin{cases} ctp(z) + n(d_0) \dot{-} 1 & \text{if } cs(z) = i_0 \text{ and } cts(z) = n(b_0) \\ ctp(z) + n(d_1) \dot{-} 1 & \text{if } cs(z) = i_1 \text{ and } cts(z) = n(b_1) \\ \vdots & \qquad\vdots \\ ctp(z) + n(d_m) \dot{-} 1 & \text{if } cs(z) = i_m \text{ and } cts(z) = n(b_m) \\ ctp(z) & \text{otherwise.} \end{cases}$$

The addition of $n(d_i) \dot{-} 1$ to the current position number increments the value by one when the transition moves the tape head to the right. Similarly, one is subtracted on a move to the left.

We have almost completed the construction of the components of the trace function. Given a machine configuration, the functions ns and ntp compute the state number and tape head position of the new configuration. All that remains is to compute the new tape number.

A transition replaces the tape symbol occupying the position scanned by the tape head. In our functional approach, the location of the tape head is obtained from the configuration number z by the function ctp. The tape symbol to be written at position $ctp(z)$ is represented numerically by $nts(z)$. The new tape number is obtained by changing the power of $pn(ctp(z))$ in the current tape number. Before the transition, the decomposition of z contains $pn(ctp(z))^{cts(z)+1}$, encoding the value of the current tape symbol at position $ctp(z)$. After the transition, position $ctp(z)$ contains the symbol represented by $nts(z)$. The primitive recursive function

$$ntn(z) = quo(ctn(z), pn(ctp(z))^{cts(z)+1}) \cdot pn(ctp(z))^{nts(z)+1}$$

makes the desired substitution. The division removes the factor that encodes the current symbol at position $ctp(z)$ from the tape number $ctn(z)$. The result is then multiplied by $pn(ctp(z))^{nts(z)+1}$, encoding the new tape symbol.

The trace function tr_M is defined by primitive recursion from the functions that simulate the effects of a transition of M on the components of the configuration. As noted previously, M is in state q_0, the tape head is at position zero, and the tape has *1*'s in positions one to $x+1$ at the start of a computation with input x. This machine configuration is encoded in $tr_M(x, 0)$:

$$tr_M(x, 0) = gn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} pn(i)^2).$$

The subsequent machine configurations are obtained using the new state, new tape position, and new tape number functions with the previous configuration as input:

$$tr_M(x, y+1) = gn_2(ns(tr_M(x, y)),\ ntp(tr_M(x, y)),\ ntn(tr_M(x, y))).$$

Since each of the functions in tr_M has been shown to be primitive recursive, we conclude that the tr_M is not only μ-recursive but also primitive recursive. The trace function, however, is not the culmination of our functional simulation of a Turing machine; it does not return the result of a computation but rather a sequence of configuration numbers.

The result of the computation of the Turing machine M that computes the number-theoretic function f with input x may be obtained from the function tr_M. We first note that the computation of M may never terminate; $f(x)$ may be undefined. The question of termination can be determined from the values of tr_M. If M specifies a transition for configuration $tr_M(x, i)$, then $tr_M(x, i) \neq tr_M(x, i+1)$ since the movement of the head changes the Gödel number. On the other hand, if M halts after transition i, then $tr_M(x, i) = tr_M(x, i+1)$ since the functions *nts*, *ntp*, and *ntn* return the preceding value when the configuration number represents a halting configuration. Consequently, the machine halts after the zth transition, where z is the first number that satisfies $tr_M(x, z) = tr_M(x, z+1)$.

Since no bound can be placed on the number of transitions that occur before an arbitrary Turing machine computation terminates, unbounded minimalization is required to determine this value. The μ-recursive function

$$term(x) = \mu z[eq(tr_M(x, z),\ tr_M(x, z+1))]$$

computes the number of the transition after which the computation of M with input x terminates. When a computation terminates, the halting configuration of the machine is encoded in the value $tr_M(x, term(x))$. Upon termination, the tape has the form $B\overline{f(x)}B$. The terminal tape number, ttn, is obtained from the terminal configuration number by

$$ttn(x) = dec(2,\ tr_M(x, term(x))).$$

The result of the computation is obtained by counting the number of *1*'s on the tape or, equivalently, determining the number of primes that are raised to the power of 2 in the terminal tape number. The latter computation is performed by the bounded sum

$$sim_M(x) = \left(\sum_{i=0}^{y} eq(1, dec(i, ttn(x))\right) \dot{-} 1,$$

where y is the length of the tape segment encoded in the terminal tape number. The bound y is computed by the primitive recursive function $gdln(ttn(x))$ (Exercise 17). One is subtracted from the bounded sum since the tape contains the unary representation of $f(x)$.

Whenever f is defined for input x, the computation of M and the simulation of M both compute the $f(x)$. If $f(x)$ is undefined, the unbounded minimalization fails to return a value and $sim_M(x)$ is undefined. The construction of sim_M completes the proof of the following theorem.

Theorem 13.7.1

Every Turing computable function is μ-recursive.

Theorems 13.6.4 and 13.7.1 establish the equivalence of the microscopic and macroscopic approaches to computation.

Corollary 13.7.2

A function is Turing computable if, and only if, it is μ-recursive.

13.8 The Church-Turing Thesis Revisited

In its functional form, the Church-Turing Thesis associates the effective computation of functions with Turing computability. Utilizing Theorem 13.7.2, the Church-Turing Thesis can be restated in terms of μ-recursive functions.

The Church-Turing Thesis (Revisited) A number-theoretic function is computable if, and only if, it is μ-recursive.

As before, no proof can be put forward for the Church-Turing Thesis. It is accepted by the community of mathematicians and computer scientists because of the accumulation of evidence supporting the claim. Accepting the Church-Turing Thesis is tantamount to bestowing the title "most general computing device" on the Turing machine. The thesis implies that any number-theoretic function that can be effectively computed by any machine or technique can also be computed by a Turing machine. This contention extends to nonnumeric computation as well.

We begin by observing that the computation of any digital computer can be interpreted as a numeric computation. Character strings are often used to communicate with the computer, but this is only a convenience to facilitate the input of the data and the interpretation of the output. The input is immediately translated to a string over $\{0, 1\}$ using either the ASCII or EBCDIC encoding schemes. After the translation, the input string can be considered the binary representation of a natural number. The computation progresses, generating another sequence of *0*'s and *1*'s, again a binary natural number. The output is then translated back to character data because of our inability to interpret and appreciate the output in its internal representation.

Following this example, we can design effective procedures that transform a string computation to a number-theoretic computation. The Gödel encoding can be used to translate strings to numbers. Let $\Sigma = \{a_0, a_1, \ldots, a_n\}$ be an alphabet and f be a function from Σ^* to Σ^*. The generation of a Gödel number from a string begins by assigning a unique number to each element in the alphabet. For simplicity we will define the numbering of the elements of Σ by their subscripts. The encoding of a string $a_{i_0}a_{i_1}\ldots a_{i_n}$ is generated by the bounded product

$$pn(0)^{i_0+1} \cdot pn(1)^{i_1+1} \cdot \cdots \cdot pn(n)^{i_n+1} = \prod_{j=0}^{y} pn(j)^{i_j+1},$$

where y is the length of the string to be encoded.

The decoding function retrieves the exponent of each prime in the prime decomposition of the Gödel number. A string can be reconstructed using the decoding function and the numbering of the alphabet. If x is the encoding of a string $a_{i_0}a_{i_1}\ldots a_{i_n}$ over Σ, then $dec(j, x) = i_j$. The original string can be obtained by concatenating the results of the decoding. Once the elements of the alphabet have been identified with natural numbers, the encoding and decoding are primitive recursive and therefore Turing computable.

The transformation of a string function f to a numeric function is obtained using character to number encoding and number to character decoding:

$$\begin{tikzcd}
u \in \Sigma^* \arrow[r, dashed, "f"] \arrow[d, "\text{encoding}"'] & f(u) \in \Sigma^* \\
x \in \mathbf{N} \arrow[r, "f'"] & f'(x) \in \mathbf{N} \arrow[u, "\text{decoding}"']
\end{tikzcd}$$

With the help of the Church-Turing Thesis, we will argue that a string function f is algorithmically computable if, and only if, the associated numeric function f' is Turing computable. We begin by noting that there is an effective procedure to obtain the values of f whenever f' is Turing computable. An algorithm to compute f consists of three steps:

i) encoding the input string u to a number x,
ii) computing $f'(x)$, and
iii) decoding $f'(x)$ to produce $f(u)$,

each of which can be performed by a Turing machine.

Now assume that there is an effective procedure to compute f. Using the reversibility of the encoding and decoding functions, we will outline an effective procedure to compute f'.

$$\begin{tikzcd}
u \in \Sigma^* \arrow[r, "f"] & f(u) \in \Sigma^* \arrow[d, "\text{encoding}"] \\
x \in \mathbf{N} \arrow[r, dashed, "f'"] \arrow[u, "\text{decoding}"] & f'(x) \in \mathbf{N}
\end{tikzcd}$$

The value $f'(x)$ can be generated by transforming the input x into a string u, computing $f(u)$, and then transforming $f(u)$ to obtain $f'(x)$. Since there is an effective procedure to compute f', the Church-Turing Thesis allows us to conclude that f' is Turing computable.

The preceding argument shows that the implications of the Church-Turing Thesis and universality of Turing machine computation are not limited to numeric computation or decision problems. A string function is computable only if it can be realized by a suitably defined Turing machine combined with a Turing computable encoding and decoding. Example 13.8.1 exhibits the correspondence between string and numeric functions.

Example 13.8.1

Let Σ be the alphabet $\{a, b\}$. Consider the function $f : \Sigma^* \to \Sigma^*$ that interchanges the a's and the b's in the input string. A number-theoretic function f' is constructed which, when combined with the functions that encode and decode strings over Σ, computes f. The elements of the alphabet are numbered by the function n: $n(a) = 0$ and $n(b) = 1$. A string $u = u_0u_1\ldots u_n$ is encoded as the number

$$pn(0)^{n(u_0)+1} \cdot pn(1)^{n(u_1)+1} \cdot \cdots \cdot pn(n)^{n(u_n)+1}.$$

The power of $pn(i)$ in the encoding is one or two depending upon whether the ith element of the string is a or b, respectively.

Let x be the encoding of a string u over Σ. Recall that $gdln(x)$ returns the length of the sequence encoded by x. The bounded product

$$f'(x) = \prod_{i=0}^{gdln(x)} \big(eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i) + eq(dec(i, x), 1) \cdot pn(i)\big)$$

generates the encoding of a string of the same length as the string u. When $eq(dec(i, x), 0) = 1$, the ith symbol in u is a. This is represented by $pn(i)^1$ in the encoding of u. The product

$$eq(dec(i, x), 0) \cdot pn(i) \cdot pn(i)$$

contributes the factor $pn(i)^2$ to $f'(x)$. Similarly, the power of $pn(i)$ in $f'(x)$ is one whenever the ith element of u is b. Thus f' constructs a number whose prime decomposition can be obtained from that of x by interchanging the exponents 1 and 2. The translation of $f'(x)$ to a string generates $f(u)$. □

Exercises

1. Let $g(x) = x^2$ and $h(x, y, z) = x + y + z$, and let $f(x, y)$ be the function defined from g and f by primitive recursion. Compute the values $f(1, 0)$, $f(1, 1)$, $f(1, 2)$ and $f(5, 0)$, $f(5, 1)$, $f(5, 2)$.
2. Using only the basic functions, composition, and primitive recursion, show that the following functions are primitive recursive. When using primitive recursion, give the functions g and h.
   a) $c_2^{(3)}$
   b) *pred*
   c) $f(x) = 2x + 2$
3. The functions below were defined by primitive recursion in Table 13.1. Explicitly, give the functions g and h that constitute the definition by primitive recursion.
   a) *sg*
   b) *sub*
   c) *exp*
4. a) Prove that a function f defined by the composition of total functions h and $g_1, \ldots, g_n$ is total.
   b) Prove that a function f defined by primitive recursion from total functions g and h is total.
   c) Conclude that all primitive recursive functions are total.
5. Let $g = id$, $h = p_1^{(3)} + p_3^{(3)}$, and let f be defined from g and h by primitive recursion.
   a) Compute the values $f(3, 0)$, $f(3, 1)$, and $f(3, 2)$.
   b) Give a closed-form (nonrecursive) definition of the function f.
6. Let $g(x, y, z)$ be a primitive recursive function. Show that each of the following functions is primitive recursive.
   a) $f(x, y) = g(x, y, x)$
   b) $f(x, y, z, w) = g(x, y, x)$
   c) $f(x) = g(1, 2, x)$
7. Let f be the function

$$f(x) = \begin{cases} x & \text{if } x > 2 \\ 0 & \text{otherwise.} \end{cases}$$

   a) Give the state diagram of a Turing machine that computes f.
   b) Show that f is primitive recursive.

8. Show that the following functions are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2. Do not use the bounded operations.
   a) $max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$
   b) $min(x, y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$
   c) $min_3(x, y, z) = \begin{cases} x & \text{if } x \leq y \text{ and } x \leq z \\ y & \text{if } y \leq x \text{ and } y \leq z \\ z & \text{if } z \leq x \text{ and } z \leq y \end{cases}$
   d) $even(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
   e) $half(x) = div(x, 2)$
   *f) $sqrt(x) = \lfloor\sqrt{x}\rfloor$
9. Show that the following predicates are primitive recursive. You may use the functions and predicates from Tables 13.1 and 13.2 and Exercise 8. Do not use the bounded operators.
   a) $le(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$
   b) $ge(x, y) = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
   c) $btw(x, y, z) = \begin{cases} 1 & \text{if } y < x < z \\ 0 & \text{otherwise} \end{cases}$
   d) $prsq(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
10. Let t be a two-variable primitive recursive function and define f as follows:

$$f(x, 0) = t(x, 0)$$
$$f(x, y + 1) = f(x, y) + t(x, y + 1)$$

    Explicitly give the functions g and h that define f by primitive recursion.
11. Let g and h be primitive recursive functions. Use bounded operators to show that the following functions are primitive recursive. You may use any functions and predicates that have been shown to be primitive recursive.
    a) $f(x, y) = \begin{cases} 1 & \text{if } g(i) < g(x) \text{ for all } 0 \leq i \leq y \\ 0 & \text{otherwise} \end{cases}$

b) $f(x, y) = \begin{cases} 1 & \text{if } g(i) = x \text{ for some } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases}$

c) $f(y) = \begin{cases} 1 & \text{if } g(i) = h(j) \text{ for some } 0 \le i, j \le y \\ 0 & \text{otherwise} \end{cases}$

d) $f(y) = \begin{cases} 1 & \text{if } g(i) < g(i+1) \text{ for all } 0 \le i \le y \\ 0 & \text{otherwise} \end{cases}$

e) $nt(x, y) =$ the number of times $g(i) = x$ in the range $0 \le i \le y$

f) $thrd(x, y) = \begin{cases} 0 & \text{if } g(i) \text{ does not assume the value } x \text{ at least} \\ & \text{three times in the range } 0 \le i \le y \\ j & \text{if } j \text{ is the third integer in the range } 0 \le i \le y \\ & \text{for which } g(i) = x \end{cases}$

g) $lrg(x, y) =$ the largest value in the range $0 \le i \le y$ for which $g(i) = x$

12. Show that the following functions are primitive recursive.

a) $gcd(x, y) =$ the greatest common divisor of x and y

b) $lcm(x, y) =$ the least common multiple of x and y

c) $pw2(x) = \begin{cases} 1 & \text{if } x = 2^n \text{ for some } n \\ 0 & \text{otherwise} \end{cases}$

d) $twopr(x) = \begin{cases} 1 & \text{if } x \text{ is the product of exactly two primes} \\ 0 & \text{otherwise} \end{cases}$

* **13.** Let g be a one-variable primitive recursive function. Prove that the function

$$f(x) = \min_{i=0}^{x}(g(i)) = min\{g(0), \ldots, g(x)\}$$

is primitive recursive.

14. Prove that the function

$$f(x_1, \ldots, x_n) = \overset{u(x_1,\ldots,x_n)}{\mu z}[p(x_1, \ldots, x_n, z)]$$

is primitive recursive whenever p and u are primitive recursive.

15. Compute the Gödel number for the following sequence:

a) 3, 0

b) 0, 0, 1

c) 1, 0, 1, 2

d) 0, 1, 1, 2, 0

16. Determine the sequences encoded by the following Gödel numbers:

a) 18,000

b) 131,072

c) 2,286,900

d) 510,510

17. Prove that the following functions are primitive recursive:

a) $gdn(x) = \begin{cases} 1 & \text{if } x \text{ is the Gödel number of some sequence} \\ 0 & \text{otherwise} \end{cases}$

b) $gdln(x) = \begin{cases} n & \text{if } x \text{ is the Gödel number of a sequence of length } n \\ 0 & \text{otherwise} \end{cases}$

c) $g(x, y) = \begin{cases} 1 & \text{if } x \text{ is a Gödel number and } y \text{ occurs in the sequence encoded in } x \\ 0 & \text{otherwise} \end{cases}$

18. Construct a primitive recursive function whose input is an encoded ordered pair and whose output is the encoding of an ordered pair in which the positions of the elements have been swapped. For example, if the input is the encoding of $[x, y]$, then the output is the encoding of $[y, x]$.

19. Let f be the function defined by

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ 2 & \text{if } x = 1 \\ 3 & \text{if } x = 2 \\ f(x-3) + f(x-1) & \text{otherwise.} \end{cases}$$

Give the values $f(4)$, $f(5)$, and $f(6)$. Prove that f is primitive recursive.

* **20.** Let g_1 and g_2 be one-variable primitive recursive functions. Also let h_1 and h_2 be four-variable primitive recursive functions. The two functions f_1 and f_2 defined by

$$\begin{aligned} f_1(x, 0) &= g_1(x) \\ f_2(x, 0) &= g_2(x) \\ f_1(x, y+1) &= h_1(x, y, f_1(x, y), f_2(x, y)) \\ f_2(x, y+1) &= h_2(x, y, f_1(x, y), f_2(x, y)) \end{aligned}$$

are said to be constructed by *simultaneous recursion* from g_1, g_2, h_1, and h_2. The values $f_1(x, y+1)$ and $f_2(x, y+1)$ are defined in terms of the previous values of both of the functions. Prove that f_1 and f_2 are primitive recursive.

21. Let f be the function defined by

$$f(0) = 1$$
$$f(y+1) = \sum_{i=0}^{y} f(i)^y.$$

a) Compute $f(1)$, $f(2)$, and $f(3)$.

b) Use course-of-values recursion to show that f is primitive recursive.

22. Let A be Ackermann's function (see Section 13.6).

a) Compute $A(2, 2)$.

b) Prove that $A(x, y)$ has a unique value for every $x, y \in \mathbf{N}$.

c) Prove that $A(1, y) = y + 2$.

d) Prove that $A(2, y) = 2y + 3$.

23. Prove that the following functions are μ-recursive. The functions g and h are assumed to be primitive recursive.

a) $cube(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect cube} \\ \uparrow & \text{otherwise} \end{cases}$

b) $root(c_0, c_1, c_2) =$ the smallest natural number root of the quadratic polynomial $c_2 \cdot x_2 + c_1 \cdot x + c_0$

c) $r(x) = \begin{cases} 1 & \text{if } g(i) = g(i+x) \text{ for some } i \geq 0 \\ \uparrow & \text{otherwise} \end{cases}$

d) $l(x) = \begin{cases} \uparrow & \text{if } g(i) - h(i) < x \text{ for all } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$

e) $f(x) = \begin{cases} 1 & \text{if } g(i) + h(j) = x \text{ for some } i, j \in \mathbf{N} \\ \uparrow & \text{otherwise} \end{cases}$

f) $f(x) = \begin{cases} 1 & \text{if } g(y) = h(z) \text{ for some } y > x,\ z > x \\ \uparrow & \text{otherwise.} \end{cases}$

***24.** The unbounded μ-operator can be defined for partial predicates as follows:

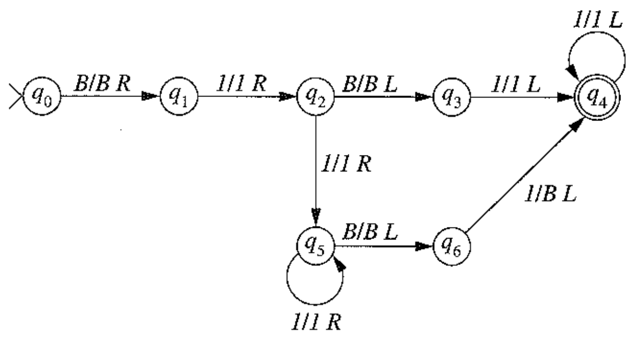
$$\mu z[p(x_1, \ldots, x_n, z)] = \begin{cases} j & \text{if } p(x_1, \ldots, x_n, i) = 0 \text{ for } 0 \leq i < j \\ & \quad \text{and } p(x_1, \ldots, x_n, j) = 1 \\ \uparrow & \text{otherwise.} \end{cases}$$

That is, the value is undefined if $p(x_1, \ldots, x_n, i) \uparrow$ for some i occurring before the first value j for which $p(x_1, \ldots, x_n, j) = 1$. Prove that the family of functions obtained by replacing the unbounded minimalization operator in Definition 13.6.3 with the preceding μ-operator is the family of Turing computable functions.

25. Construct the functions ns, ntp, and nts for the Turing machine S given in Example 13.7.1.

26. Let M be the machine

M:



a) What unary number-theoretic function does M compute?

b) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{0}$.

c) Give the tape numbers for each configuration that occurs in the computation of M with input $\overline{2}$.

27. Let f be the function defined by

$$f(x) = \begin{cases} x+1 & \text{if } x \text{ even} \\ x-1 & \text{otherwise.} \end{cases}$$

a) Give the state diagram of a Turing machine M that computes f.

b) Trace the computation of your machine for input 1 ($B11B$). Give the tape number for each configuration in the computation. Give the value of $tr_{\mathrm{M}}(1, i)$ for each step in the computation.

c) Show that f is primitive recursive. You may use the functions from the text that have been shown to be primitive recursive in Sections 13.1, 13.2, and 13.4.

***28.** Let M be a Turing machine and tr_{M} the trace function of M.

a) Show that the function

$$prt(x, y) = \begin{cases} 1 & \text{if the } y\text{th transition of M with input } x \text{ prints} \\ & \text{a blank} \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

b) Show that the function

$$lprt(x) = \begin{cases} 1 & \text{if the final transition of M} \\ & \text{with input } x \text{ that prints a 1} \\ \uparrow & \text{otherwise} \end{cases}$$

is μ-recursive.

c) In light of undecidability of the printing problem (Exercise 12.7), explain why *lprt* cannot be primitive recursive.

29. Give an example of a function that is not μ-recursive. *Hint*: Consider a language that is not recursively enumerable.

30. Let f be the function from $\{a, b\}^*$ to $\{a, b\}^*$ defined by $f(u) = u^R$. Construct the primitive recursive function f' that, along with the encoding and decoding functions, computes f.

31. A number-theoretic function is said to be *macro-computable* if it can be computed by a Turing machine defined using only the machines S and D that compute the successor and predecessor functions and the macros from Section 9.3. Prove that every μ-recursive function is macro-computable. To do this you must show that

i) The successor, zero, and projection functions are macro-computable.

ii) The macro-computable functions are closed under composition, primitive recursion, and unbounded minimimalization.

32. Prove that the programming language TM defined in Section 9.6 computes the entire set of μ-recursive functions.

Bibliographic Notes

The functional and mechanical development of computability flourished in the 1930s. Gödel [1931] defined a method of computation now referred to as Herbrand-Gödel computability. The properties of Herbrand-Gödel computability and μ-recursive functions were developed extensively by Kleene. The equivalence of μ-recursive functions and Turing computability was established in Kleene [1936]. Post machines [Post, 1936] provide an alternative mechanical approach to numeric computation. The classic book by Kleene [1952] presents computability, the Church-Turing Thesis, and recursive functions. A further examination of recursive function theory can be found in Hermes [1965], Péter [1967], and Rogers [1967]. Hennie [1977] develops computability from the notion of an abstract family of algorithms.

Ackermann's function was introduced in Ackermann [1928]. An excellent exposition of the features of Ackermann's function can be found in Hennie [1977].