Computability, Complexity, and Languages

By Martin D. Davis et al

First Edition

CONTENTS

CHAPTER 1

PRELIMINARIES

1 Sets and n-tuples

We shall often be dealing with *sets* of objects of some definite kind. Thinking of a collection iof entities as a *set* simply amounts to a decision to regard the whole collection as a single object. We shall use the word *class* as synonymous with *set*. In particular we write N for the set of *natural numbers* $0, 1, 2, 3 \cdots$.

It is useful to speak of the *empty set*, written $\varnothing$, which has no members. The equation $R = S$, where R and S are sets, means that R and S are *identical as sets*, that is, that they have exactly the same members. We write $R \subseteq S$ and speak of R as a *subset* of S to mean that every element of R is also an element of S. We write $R \subset S$ to indicate that $R \subseteq S$ but $R \neq S$. In this case R is called a *proper subset* of S. If R and S are set, we write $R \cup S$ for the *union* of R and S, which is the collection of all objects which are members of either R or S or both. $R \cap S$, the *intersection* of R and S, is the set of all objects that belong to both R and S. $R - S$, the set of all objects that belong to R and do not belong to S, is the *difference* between R and S. Often we will be working in contexts where all sets being considered are subsets of some fixed set D (sometimes called a *domain* or a *universe*). In such a case we write $\bar{S}$ for $D - S$, and call $\bar{S}$ the *complement* of S. We write

$$\{a_1, a_2, \cdots, a_n\}$$

for the set consisting of the n objects $a_1, a_2, \cdots, a_n$. Sets that can be written in this form as well as the empty set are called *finite*. Sets that are not finite are called *infinite*. Since two sets are equal if and only if they have the same members. That is, the order in which we may choose to write the members of a set is irrelevant. Where order is important, we speak instead of an n-tuple or a *list*. A 2-tuple is called an *ordered pair*, and a 3-tuple is called an *ordered triple*. Unlike the case for sets of one object, we *do not distinguish between the object a and the 1-tuple* (a). The crucial property of n-tuples is

$$(a_1, a_2, \cdots, a_n) = (b_1, b_2, \cdots, b_n)$$

if and only if

$$a_1 = b_1, \quad a_2 = b_2, \quad \ldots, \quad \textit{and} \quad a_n = b_n.$$

If $S_1, S_2, \cdots, S_n$ are given sets, then we write $S_1 \times S_2 \times \cdots \times S_n$ for the set of all n-tuples such that $a_1 \in S_1, a_2 \in S_2, \cdots, a_n \in S_n$. $S_1 \times S_2 \times \cdots \times S_n$ is sometimes called the *Cartesian product* of $S_1, S_2, \cdots, S_n$.

2 Functions

For f a function, one writes $f(a) = b$ to mean that $(a, b) \in f$; the definition of function ensures that for each a there can be at most one such b. The set of all a such that $(a, b) \in f$ for some b is called the *domain* of f. The set of all $f(a)$ for a in the domain of f is called the *range* of f.

Functions f are often specified by *algorithms* that provide procedures for obtaining $f(a)$ from a. However, it is quite possible to possess an algorithm that specifies a function without being able to tell which elements belong to its domain. This makes the notion of a so-called *partial function* play a central role in computability theory. A *partial function on a set S* is simply a function whose domain is a subset of S. If f is a partial function on S and $a \in S$, then we write $f(a) \downarrow$ and say that $f(a)$ is *defined* to indicate that a is in the domain of f; if a is not in the domain of f, we write $f(a) \uparrow$ and say that $f(a)$ is *undefined*. If a partial function on S has the domain S, then it is called *total*. Finally, we should mention that the empty set $\varnothing$ is itself a function. Considered as a partial function on some set S, *it is nowhere defined*.

A partial function f on a set S^n is called an *n-ary partial function on S*, or a function of n variables on S. We use *unary* and *binary* for 1-ary and 2-ary, respectively.

A function f is *one-one* if, for all x, y in the doamin of f, $f(x) = f(y)$ implies $x = y$. If the range of f is the set S, then we say that f is an *onto* function with respect to S, or simply that f is *onto S*.

We will sometimes refer to the idea of *closure*. If S is a set and f is a partial function on S, then S is *closed under f* if the range of f is a subset of S.

3 Alphabets and Strings

An *alphabet* is simply some finite nonempty set A of objects called *symbols*. An n-tuple of symbols of A is called a *word* or a *string* on A. The set of all words on the alphabet A is written A^*. Any subset of A^* is called a *language on A* or a *language with alphabet A*. We do *not* distinguish between a symbol $a \in A$ and the word of length 1 consisting of that symbol.

4 Predicates

By a *predicate* or a *Boolean-valued function* on a set S we mean a *total* function P on S such that for each $a \in S$, either

$$P(a) = \text{TRUE} \quad \text{or} \quad P(a) = \text{FALSE},$$

where TRUE and FALSE are a pair of distinct objects called *truth values*. We often say *$P(a)$ is true* for $P(a)$ =TRUE, and *$P(a)$ is false* for $P(a)$ =FALSE. Given a predicate P on a set S, there is a corresponding subset R of S, namely, the set of all elements $a \in S$ for which $P(a) = 1$. The predicate P is called the *characteristic function* of the set R.

5 Quantifiers

In this section we will be concerned exclusively with predicates on N^m (or what is the same thing, m-ary predicates on N) for different values of m. Thus, let $P(t, x_1, \cdots, x_n)$ be an $(n+1)$-ary predicate. Consider the predicate $Q(y, x_1, \cdots, x_n)$ defined by

$$\begin{aligned} Q(y, x_1, \ldots, x_n) \Leftrightarrow & P(0, x_1, \ldots, x_n) \vee P(1, x_1, \ldots, x_n) \\ & \vee \cdots \vee P(y, x_1, \ldots, x_n). \end{aligned}$$

Thus the predicate $Q(y, x_1, \cdots, x_n)$ is true just in case there is value of $t \leq y$ such that $P(t, x_1, \cdots, x_n)$ is true. We write this predicate Q as

$$(\exists t)_{\leq y} P(t, x_1, \ldots, x_n).$$

The expression "$(\exists t)_{\leq y}$" is called a *bounded existential quantifier.* Similarly, we write $(\forall t)_{\leq y} P(t, x_1, \ldots, x_n)$ for the predicate

$$P(0, x_1, \ldots, x_n) \& P(1, x_1, \ldots, x_n) \& \cdots \& P(y, x_1, \ldots, x_n).$$

The predicate is true just in case $P(t, x_1. \cdots, x_n)$ is true for *all* $t \leq y$. The expression "$(\forall t)_{\leq y}$" is called a *bounded universal quantifier.*

6 Proof by Contradiction

Recall that a number is called a *prime* if it has *exactly two distinct divisors*, itself and 1. Consider the following assertion:

$$n^2 - n + 41 \text{ is prime for all } n \in N.$$

This assertion is in fact *false.*

In a *proof by contradiction*, one begins by supposing that the assertion we wish to prove is false. In a proof by contradiction we look for a pair of statements developed in the course of the proof which *contradict* one another.

Theorem 6.1

Let $x \in \{a, b\}^*$ such that $xa = ax$. Then $x = a^{[n]}$ for some $n \in N$.

7 Mathematical Induction

Mathematical induction furnishes an important technique for proving statements of the form $(\forall n)P(n)$, where P is a predicate on N. One proceeds by proving a pair of auxiliary statements, namely, $P(0)$ and

$$(\forall n)(\textit{if } P(n) \textit{ then } P(n+1)). \tag{1.1}$$

Why is this helpful? Because sometimes it is much easier to prove (1.1) than to prove $(\forall n)P(n)$ in some other way. In proving this second auxiliary proposition one typically considers some fixed but arbitrary value k of n and shows that if we assume $P(k)$ we can prove $P(k+1)$. $P(k)$ is then called the *induction hypothesis*.

There are some paradoxical things about proofs by mathematical induction. One is assuming $P(k)$ for some *particular* k in order to show that $P(k+1)$ follows.

It is also paradoxical that in using induction (we shall often omit the word *mathematical*), it is sometimes easier to prove statements by first making them "stronger." We wish to prove $(\forall n)P(n)$. Instead we decide to prove the *stronger* assertion $(\forall n)(P(n)\&Q(n))$ (which of course implies the original statement). The technique of deliberately strengthening what is to be proved for the purpose of making proofs by induction easier is called *induction loading*.

Theorem 7.1

For all $n \in N$ we have $\sum_{i=0}^{n}(2i+1) = (n+1)^2$.

Another form of mathematical induction that is often very useful is called *course-of-values induction* or sometimes *complete induction*.

Theorem 7.2

There is no string $x \in \{a,b\}^*$ such that $ax = xb$.

CHAPTER 2

PROGRAMS AND COMPUTABLE FUNCTIONS

1 A Programming Language

In particular, the letters

$$X_1 \; X_2 \; X_3 \; \cdots$$

will be called the *input variables* of $\mathscr{L}$, the letter Y will be called the *output variable* of $\mathscr{L}$, and the letters

$$Z_1 \; Z_2 \; Z_3 \; \cdots$$

will be called the *local variables* of $\mathscr{L}$.

In $\mathscr{L}$ we will be able to write "instructions" of various sorts; a "program" of $\mathscr{L}$ will then consist of a *list* (i.e., a finite sequence) of instructions.

Table 2.1

Insturction	Interpretation
$V \leftarrow V + 1$	Increase by 1 the value of the variable V.
$V \leftarrow V - 1$	If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of V.
IF $V \neq 0$ GOTO L	If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list

We give in Table 2.1 a complete list of our instructions. In this list V stands for any variable and L stands for any label.

These instructions will be called the *increment*, *decrement*, and *conditional branch* instructions, respectively.

We will use the special convention that *the output variable Y and the local variables Z_i initially have the value* 0.

2 Some Examples of Programs

Our first example is the program

$$
\begin{array}{ll}
[A] & X \leftarrow X - 1 \\
 & Y \leftarrow Y + 1 \\
 & \text{IF } X \neq 0 \text{ GOTO } A
\end{array}
$$

If the initial value x of X is not 0, the effect of this program is to copy x into Y and to decrement the value of X down to 0. We will say that this program *computes* the function

$$
f(x) = \begin{cases} 1 & \text{if} \quad x = 0 \\ x & \text{otherwise.} \end{cases}
$$

Although the preceding program is a perfectly well-defined program of our language $\mathscr{L}$, we may think of it as having arisen in an attempt to write a program that copies the value of X into Y, and therefore containing a "bug" because it does not handle 0 correctly. The following slightly more complicated example remedies this situation.

$$
\begin{array}{ll}
[A] & \text{IF } X \neq 0 \text{ GOTO } B \\
 & Z \leftarrow Z + 1 \\
 & \text{IF } Z \neq 0 \text{ GOTO } E \\
[B] & X \leftarrow X - 1 \\
 & Y \leftarrow Y + 1 \\
 & Z \leftarrow Z + 1 \\
 & \text{IF } Z \neq 0 \text{ GOTO } A
\end{array}
$$

At first glance Z's role in the computation may not be obvious. It is used simply to allow us to code an *unconditional branch*. That is, the program segment

$$
\begin{array}{l}
Z \leftarrow Z + 1 \\
\text{IF } Z \neq 0 \text{ GOTO } L
\end{array} \tag{2.1}
$$

has the effect (ignoring the effect on the value of Z) of an instruction

$$\text{GOTO } L$$

such as is available in most programming languages. Now GOTO L is not an instruction in our language $\mathscr{L}$, but since we will frequently have use for such an instruction, we can use it as an abbreviation for the program segment (3.1). Such an abbreviating pseudoinstruction will be called a *macro* and the program or program segment which it abbreviates will be called it *macro expansion*.

For our final example, we take the program

$$
\begin{array}{ll}
 & Y \leftarrow X_1 \\
 & Z \leftarrow X_2 \\
[C] & \text{IF } Z \neq 0 \text{ GOTO } A \\
 & \text{GOTO } E \\
[A] & \text{IF } Y \neq 0 \text{ GOTO } B \\
 & \text{GOTO } A \\
[B] & Y \leftarrow Y - 1 \\
 & Z \leftarrow Z - 1 \\
 & \text{GOTO } C
\end{array}
$$

What happens if we begin with a value of X_1 less than the value of X_2? At this point the computation enters the "loop":

$$\begin{array}{ll} [A] & \text{IF } Y \neq 0 \text{ GOTO } B \\ & \text{GOTO } A \end{array}$$

Since $y = 0$, there is no way out of this loop and the computation will continue "forever." Thus, if we begin with $X_1 = m$, $X_2 = n$, where $m < n$, the computation will never terminate. In this case (and in similar cases) we will say that the program computes the *partial function*

$$g(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{if} \quad x_1 \geq x_2 \\ \uparrow & \text{if} \quad x_1 < x_2. \end{cases}$$

3 Syntax

The symbols

$$X_1 \; X_2 \; X_3 \; \cdots$$

are called *input variables*,

$$Z_1 \; Z_2 \; Z_3 \; \cdots$$

are called *local variables*, and Y is called the *output variable* of $\mathscr{L}$. The symbols

$$A_1, \; B_1 \; C_1 \; D_1 \; E_1 \; A_2 \; B_2 \; \cdots$$

are called *labels* of $\mathscr{L}$. A *statement* is one of the following:

$$\begin{array}{l} V \leftarrow V + 1 \\ V \leftarrow V - 1 \\ V \leftarrow V \\ \text{IF } V \neq 0 \text{ GOTO } L \end{array}$$

where V may be any variable and L may be any label.

Next, an *instruction* is either a statement (in which case it is also called an *unlabeled* instruction) or $[L]$ followed by a statement (in which case the instruction is said to have L as its label or to be labeled L). A *program* is a list (i.e., a finite sequence) of instructions. The length of this list is called the *Length* of the progra. It is useful to include the *empty program* of length 0, which of course contains no instructions.

A *state of a program* $\mathscr{P}$ is a list of equations of the form $V = m$, where V is a variable and m is a number, including an equation for each variable that occurs in $\mathscr{P}$ and including no two equations with the same variable. As an example, let $\mathscr{P}$ be the program which contains the variables $X \; Y \; Z$. (The definition of *state* does not require that the state can actually be "attained" from some initial state.) The list

$$X = 3, \quad Z = 3$$

is *not* a state of $\mathscr{P}$ since no equation in Y occurs. Likewise, the list

$$X = 3, \quad X = 4, \quad Y = 2, \quad Z = 2$$

is *not* a state of $\mathscr{P}$: there are two equations in X.

Let σ be a state of $\mathscr{P}$ and let V be a variable that occurs in σ. The *value of* V *at* σ is then the (unique) number q such that the equation $V = q$ is one of the equations making up σ.

Suppose we have a program $\mathscr{P}$ and a state σ of $\mathscr{P}$. In order to say what happens "next," we also need to know which instruction of $\mathscr{P}$ is about to be executed. We therefore define a *snapshot* or *instantaneous description* of a program $\mathscr{P}$ of length n to be a pair (i, σ) where $1 \leq i \leq n+1$, and σ is a state of $\mathscr{P}$.

If $s = (i, \sigma)$ is a snapshot of $\mathscr{P}$ and V is a variable of $\mathscr{P}$, then the *value* of V at s just means the value of V at σ.

A snapshot (i, σ) of a program $\mathscr{P}$ of length n is called *terminal* if $i = n+1$. If (i, σ) is a nonterminal snapshot of $\mathscr{P}$, we define the *successor* of (i, σ) to be the snapshot (j, τ) defined as follows:

Case 1. The ith instruction of $\mathscr{P}$ is $V \leftarrow V + 1$ and σ contains the equation $V = m$. Then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m + 1$ (i.e., the value of V at τ is $m + 1$).

Case 2. The ith instruction of $\mathscr{P}$ is $V \leftarrow V - 1$ and σ contains the equation $V = m$. Then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m - 1$ if $m \neq 0$; if $m = 0$, $\tau = \sigma$.

Case 3. The ith instruction of $\mathscr{P}$ is $V \leftarrow V$. Then $\tau = \sigma$ and $j = i + 1$.

Case 4. The ith instruction of $\mathscr{P}$ is IF $V \neq 0$ GOTO L. Then $\tau = \sigma$, and there are two subcases:

Case 4a. σ contains the equation $V = 0$. Then $j = i + 1$.

Case 4b. σ contains the equation $V = m$ where $m \neq 0$. Then, if there is an instruction of $\mathscr{P}$ labeled L, j is the *least number* such that the jth instruction of $\mathscr{P}$ is labeled L. Otherwise, $j = n + 1$.

A *computation* of a program $\mathscr{P}$ is defined to be a sequence (i.e., a list) $s_1, s_2, \ldots, s_k$ of snapshots of $\mathscr{P}$ such that s_{i+1} is the successor of s_i for $i = 1, 2, \cdots, k-1$ and s_k is terminal.

Note that we have not forbidden a program to contain more than one instruction having the same label. However, our definition of successor of a snapshot, in effect, interprets a branch instruction as always referring to the *first* statement in the program having the label in question.

4 Computable Functions

One would expect a program that computes a function of m variables to contain the input variables $X_1, X_2, \ldots, X_m$, and the output variable Y, and to have all other variables (if any) in the program to be local.

Thus, let $\mathscr{P}$ be any program in the language $\mathscr{L}$ and let $r_1, \ldots, r_m$ be m given numbers. We form the state σ of $\mathscr{P}$ which consists of the equations

$$X_1 = r_1, \quad X_2 = r_2, \quad \ldots, \quad X_m = r_m, \quad Y = 0$$

together with the equations $V = 0$ for each variable V in $\mathscr{P}$ other than $X_1, \ldots, X_m, Y$. We will call this the *initial state*, and the snapshot $(1, \sigma)$, the *initial snapshot*.

Case 1. *There is a computation* $s_1, s_2, \dots, s_k$ of $\mathscr{P}$ *beginning with the initial snapshot.* Then we write $\psi_{\mathscr{P}}^{(m)}(r_1, \dots, r_m)$ for the value of the variable Y at the (terminal) snapshot s_k.

Case 2. *There is no such computation*; i.e., there is an *infinite* sequence $s_1, s_2, s_3, \dots$ beginning with the initial snapshot where each s_{i+1} is the successor of s_i. In this case $\psi_{\mathscr{P}}^{(m)}(r_1, \dots, r_m)$ is undefined.

For any program $\mathscr{P}$ and any positive integer m, the function $\psi_{\mathscr{P}}^{(m)}(r_1, \dots, r_m)$ is said to be *computed* by $\mathscr{P}$. A given partial function g (of one or more variables) is said to be *partially computable* if it is computed by some program.

A given function g of m variables is called *total* if $g(r_1, \dots, r_m)$ is defined for *all* $r_1, \dots, r_m$. A function is said to be *computable* of it is both partially computable and total.

Partially computable functions are also called *partial recursive*, and computable functions, i.e., functions that are both total and partial recursive, are called *recursive*.

5 More about Macros

We now see how to augment our language to include macros of the form

$$\text{IF } P(V_1, \dots, V_n) \text{ GOTO } L$$

where $P(x_1, \dots, x_n)$ is a computable predicate. Here we are making use of the convention that

$$\text{TRUE} = 1, \quad \text{FALSE} = 0.$$

Hence predicates are just total functions whose values are always either 0 or 1. And therefore, it makes perfect sense to say that some given *predicate* is or is not computable.

CHAPTER 3

PRIMITIVE RECURSIVE FUNCTIONS

1 Composition

We want to combine computable functions in such a way that the output of one becomes an input to another. In the simplest case we combine functions f and g to obtain the function

$$h(x) = f(g(x)).$$

More generally, for functions of several variables:

Definition 1.1

Let f be a function of k variables and let $g_1, \ldots, g_k$ be functions of n variables. Let

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

Then h is said to be obtained from f and $g_1, \ldots, g_k$ by *composition*.

Theorem 1.2

If h is obtained from the (partially) computable functions $f, g_1, \ldots, g_k$ by composition, then h is (partially) computable.
The word *partially* is placed in parentheses in order to assert the correctness of the statement with the word included or omitted in both places.

2 Recursion

Suppose k is some fixed number and

$$\begin{aligned} h(0) &= k, \\ h(t+1) &= g(t, h(t)), \end{aligned} \tag{3.1}$$

where g is some given *total* function of two variables. Then h is said to be obtained from g by *primitive recursion*, or simply *recursion*.

Theorem 2.1

Let h be obtained from g as in (3.1), and let g be computable. Then h is also computable.

A slightly more complicated kind of recursion is involved when we have

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n, t+1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n). \end{aligned} \tag{3.2}$$

Here the function h of $n+1$ variables is said to be obtained by *primitive recursion*, or simply *recursion*, from the total functions f (of n variables) and g (of $n+2$ variables). Again we have

Theorem 2.2

Let h be obtained from f and g as in (3.2) and let f, g be computable. Then h is also computable.

3 PRC Classes

Now we need some functions on which to get started. These will be

$$\begin{aligned} s(x) &= x+1, \\ n(x) &= 0, \end{aligned}$$

and the *projection functions*

$$u_i^n(x_1, \ldots, x_n) = x_i, \quad 1 \le i \le n.$$

The functions s, n, and u_i^n are called the *initial functions*.

Definition 3.1

A class of total functions $\mathscr{C}$ is called a *PRC class* if

1. the initial functions belong to $\mathscr{C}$.
2. a function obtained from functions belonging to $\mathscr{C}$ by either composition or recursion also belongs to $\mathscr{C}$.

Then we have

Theorem 3.2

The class of computable functions is a PRC class.

Definition 3.3

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

It is obvious from this definition that

Corollary 3.4

The class of primitive recursive functions is a PRC class.

Actually we can say more:

Theorem 3.5

A function is primitive recursive if and only if it belongs to every PRC class.

Corollary 3.6

Every primitive recursive function is computable.

4 Some Primitive Recursive Functions

The *predecessor function* $p(x)$ is defined as follows:

$$p(x) = \begin{cases} x - 1 & \text{if} \quad x \neq 0 \\ 0 & \text{if} \quad x = 0. \end{cases}$$

5 Primitive Recursive Predicates

Theorem 5.1

Let $\mathscr{C}$ be a PRC class. If P, Q are predicates that belong to $\mathscr{C}$, then so are $\sim P$, $P \vee Q$, and $P\&Q$.

A result which refers to PRC classes can be applied to the two classes we have shown to be PRC. That is, taking $\mathscr{C}$ to be the class of all primitive recursive functions, we have

Corollary 5.2

If P, Q are primitive recursive predicates, then so are $\sim P$, $P \vee Q$, and $P\&Q$.

Similarly taking $\mathscr{C}$ to be the class of all computable functions, we have

Corollary 5.3

If P, Q are computable predicates, then so are $\sim P$, $P \vee Q$, and $P\&Q$.

Theorem 5.4: Definition by Cases

Let $\mathscr{C}$ be a PRC class. Let the function g, h and the predicate P belong to $\mathscr{C}$. Let

$$f(x_1,\ldots,x_n)=\begin{cases} g(x_1,\ldots,x_n) & \text{if} \quad P(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n) & \text{otherwise.}\end{cases}$$

Then f belongs to $\mathscr{C}$.
This will be recognized as a version of the familiar "if...then..., else..." statement.

Corollary 5.5

Let $\mathscr{C}$ be a PRC class, let n-ary functions $g_1,\ldots,g_m,h$ and predicates $P_1,\ldots,P_m$ belong to $\mathscr{C}$, and let

$$P_i(x_1,\ldots,x_n)\&P_j(x_1,\ldots,x_n)=0$$

for all $1\le i<j\le m$ and all $x_1,\ldots,x_n$. If

$$f(x_1,\ldots,x_n)=\begin{cases} g_1(x_1,\ldots,x_n) & \text{if} \quad P_1(x_1,\ldots,x_n) \\ \quad\vdots & \qquad\vdots \\ g_m(x_1,\ldots,x_n) & \text{if} \quad P_m(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n) & \text{otherwise,}\end{cases}$$

then f also belongs to $\mathscr{C}$.

6 Iterated Operations and Bounded Quantifiers

Theorem 6.1

Let $\mathscr{C}$ be a PRC class. If $f(t,x_1,\ldots,x_n)$ belongs to $\mathscr{C}$, then so do the functions

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

and

$$h(y,x_1,\ldots,x_n)=\prod_{t=0}^{y} f(t,x_1,\ldots,x_n).$$

A common error is to attempt to prove this by using mathematical induction on y. A little reflection reveals that such an argument by induction shows that

$$g(0,x_1,\ldots,x_n),g(1,x_1,\ldots,x_n),\ldots$$

all belong to $\mathscr{C}$, but not that the function $g(y,x_1,\ldots,x_n)$, one of whose arguments is y, belongs to $\mathscr{C}$.

Sometimes we will want to begin the summation (or product) at 1 instead of 0. Then the initial recursion equations can be taken to be

$$g(0, x_1, \ldots, x_n) = 0,$$

$$h(0, x_1, \ldots, x_n) = 1,$$

with the equations for $g(t+1, x_1, \ldots, x_n)$ and $h(t+1, x_1, \ldots, x_n)$. Note that we are implicitly defining a vacuous sum to be 0 and a vacuous product to be 1. With this understanding we have proved

Corollary 6.2

If $f(t, x_1, \ldots, x_n)$ belongs to the PRC class $\mathscr{C}$, then so do the functions

$$g(y, x_1, \ldots, x_n) = \sum_{t=1}^{y} f(t, x_1, \ldots, x_n)$$

and

$$h(y, x, \ldots, x_n) = \prod_{t=1}^{y} f(t, x_1, \ldots, x_n).$$

We have

Theorem 6.3

If the predicate $P(t, x_1, \ldots, x_n)$ belongs to some PRC class $\mathscr{C}$, then so do the predicates

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) \quad \text{and} \quad (\exists t)_{\leq y} P(t, x_1, \ldots, x_n).$$

The predicate "x is a prime" is primitive recursive since

$$\text{Prime}(x) \Leftrightarrow x > 1 \& (\forall t)_{\leq x}\{t = 1 \vee t = x \vee \sim (t|x)\}$$

(A number is a *prime* if it is greater than 1 and it has no divisors other than 1 and itself.)

7 Minimalization

Let $P(t, x_1, \ldots, x_n)$ belong to some given PRC class $\mathscr{C}$. Then by Theorem 6.1, the function

$$g(y, x_1, \ldots, x_n) = \sum_{u=0}^{y} \prod_{t=0}^{u} \alpha(P(t, x_1, \ldots, x_n))$$

also belongs to $\mathscr{C}$. Suppose for definiteness that for some value of $t_0 \leq y$,

$$P(t, x_1, \ldots, x_n) = 0 \quad \text{for } t < t_0$$

but

$$P(t_0, x_1, \ldots, x_n) = 1,$$

i.e., that t_0 *is the least value of* $t \leq y$ *for which* $P(t, x_1, \ldots, x_n)$ *is true.* Then

$$\prod_{t=0}^{u} \alpha(P(t, x_1, \ldots, x_n)) = \begin{cases} 1 & \text{if } u < t_0 \\ 0 & \text{if } u \geq t_0. \end{cases}$$

Hence,

$$g(y, x_1, \ldots, x_n) = \sum_{u<t_0} 1 = t_0,$$

so that $g(y, x_1, \ldots, x_n)$ is the least value of t for which $P(t, x, \ldots, x_n)$ is true. Now, we define

$$\min_{t\leq y} P(t, x_1, \ldots, x_n) = \begin{cases} g(y, x_1, \ldots, x_n) & \text{if } (\exists t)_{\leq y} P(t, x_1, \ldots, x_n) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\min_{t\leq y} P(t, x_1, \ldots, x_n)$ *is the least value of* $t \leq y$ *for which* $P(t, x_1, \ldots, x_n)$ *is true, if such exists; otherwise it assumes the* (default) *value* 0. Using Theorems 5.4 and 6.3, we have

Theorem 7.1

If $P(t, x_1, \ldots, x_n)$ belongs to some PRC class $\mathscr{C}$ and $f(y, x_1, \ldots, x_n) = \min_{t\leq y} P(t, x_1, \ldots, x_n)$, then f also belongs to $\mathscr{C}$.
The operation "$\min_{t\leq y}$" is called *bounded minimalization*.

$R(x, y)$ is the *remainder* when x is divided by y.

Here, for $n > 0$, p_n is the nth prime number (in order of size). So that p_n be a total function, we set $p_0 = 0$.

Consider the recursion equations

$$p_0 = 0,$$
$$p_{n+1} = \min_{t\leq p_n!+1} [\text{Prime}(t) \& t > p_n].$$

To see that these equations are correct we must verify the inequality

$$p_{n+1} \leq (p_n)! + 1. \tag{3.3}$$

We write

$$\min_y P(x_1, \ldots, x_n, y)$$

for the least value of y for which the predicate P is true *if there is one. If there is no value of* y *for which* $P(x_1, \ldots, x_n, y)$ *is true, then* $\min_y P(x_1, \ldots, x_n, y)$ *is undefined.* Now, there are primitive recursive predicates $P(x, y)$ such that $\min_y P(x, y)$ is a total function which is *not* primitive recursive. However, we can prove

Theorem 7.2

If $P(x_1, \ldots, x_n, y)$ is a computable predicate and if

$$g(x_1, \ldots, x_n) = \min_y P(x_1, \ldots, x_n, y),$$

then g is a partially computable function.

8 Pairing Functions and Gödel Numbers

If z is any given number, there is a *unique* solution x, y to the equation

$$\langle x, y \rangle = z, \tag{3.4}$$

namely, x is the largest number such that $2^x|(z+1)$, and y is then the solution of the equation

$$2y + 1 = (z+1)/2^x$$

this last equation has a (unique) solution because $(z+1)/2^x$ must be odd.

We summarize the properties of the functions $\langle x, y\rangle$, $l(z)$, and $r(z)$ in

Theorem 8.1: Pairing Function Theorem

The functions $\langle x, y\rangle$, $l(z)$, and $r(z)$ have the following properties:

1. they are primitive recursive;
2. $l(\langle x, y\rangle) = x, r(\langle x, y\rangle) = y$;
3. $\langle l(z), r(z)\rangle = z$;
4. $l(z), r(z) \le z$.

We define the *Gödel number* of the sequence $(a_1, \ldots, a_n)$ to be the number

$$[a_1, \ldots, a_n] = \prod_{i=1}^{n} p_i^{a_i}.$$

Gödel numbering satisfying the following uniqueness property:

Theorem 8.2

If $[a_1, \ldots, a_n] = [b_1, \ldots, b_n]$, then

$$a_i = b_i, \quad i = 1, \ldots, n.$$

This result is an intermediate consequence of the uniqueness of the factorization of integers into primes, sometimes referred to as the *unique factorization theorem* or the *fundamental theorem of arithmetic*.

However, note that

$$[a_1, \ldots, a_n] = [a_1, \ldots, a_n, 0] \tag{3.5}$$

because $p_{n+1}^0 = 1$.

We will now define a primitive recursive function $(x)_i$ so that if

$$x = [a_1, \ldots, a_n],$$

then $(x)_i = a_i$.

We shall also use the primitive recursive function

$$\mathrm{Lt}(x) = \min_{i \le x}((x)_i \ne 0 \& (\forall j)_{\le x}(j \le i \vee (x)_j = 0)).$$

We summarize the key properties of these primitive recursive functions.

Theorem 8.3: Sequence Number Theorem

a. $([a_1, \ldots, a_n])_i = \begin{cases} a_i & \text{if} \quad 1 \leq i \leq n \\ 0 & \text{otherwise.} \end{cases}$

b. $[(x)_1, \ldots, (x)_n] = x \quad \text{if} \quad n \geq \mathrm{Lt}(x).$

CHAPTER 4

A UNIVERSAL PROGRAM

1 Coding Programs by Numbers

Note that for any given number q there is a unique instruction I with $\#(I) = q$. If $l(q) = 0$, I is unlabeled; otherwise I has the $l(q)$th label in our list. To find the variable mentioned in I, we compute $i = r(r(q)) + 1$ and locate the ith variable V in our list. Then, the statement in I will be

$$\begin{aligned} V \leftarrow V \quad & \text{if} \quad l(r(q)) = 0, \\ V \leftarrow V + 1 \quad & \text{if} \quad l(r(q)) = 1, \\ V \leftarrow V - 1 \quad & \text{if} \quad l(r(q)) = 2, \\ \text{IF } V \neq 0 \text{ GOTO } L \quad & \text{if} \quad j = l(r(q)) - 2 > 0 \end{aligned}$$

and L is the jth label in our list.

Finally, let a program $\mathscr{P}$ consist of the instructions $I_1, I_2, \ldots, I_k$. Then we set

$$\#(\mathscr{P}) = [\#(I_1), \#(I_2), \ldots, \#(I_k)] - 1. \tag{4.1}$$

Note that the number of the unlabeled instruction $Y \leftarrow Y$ is

$$\langle 0, \langle 0, 0 \rangle \rangle = \langle 0, 0 \rangle = 0.$$

Thus, by the ambiguity in Gödel numbers [recall Eq. 3.5], the number of a program will be unchanged if an unlabeled $Y \leftarrow Y$ is tacked onto its end. Of corse this is a harmless ambiguity; the longer program computes exactly what the shorter one does. However, we remove even this ambiguity by adding to our official definition of program of $\mathscr{P}$ the harmless stipulation that *the final instruction in a program is not permitted to be the unlabeled statement* $Y \leftarrow Y$.

2 The Halting Problem

For given y, let $\mathscr{P}$ be the program such that $\#(\mathscr{P}) = y$. Then HALT(x, y) *is true if* $\psi_{\mathscr{P}}^{(1)}(x)$ is defined and false if $\psi_{\mathscr{P}}^{(1)}(x)$ is undefined.

We now prove the remarkable

Theorem 2.1

HALT(x, y) is not a computable predicate.

To begin with, this theorem provides us with an example of a function that is not computable by any program in the language $\mathscr{L}$. But we would like to go further; we would like to conclude the following:

> *There is no algorithm that, given a program of $\mathscr{L}$ and an input to that program, can determine whether or not the given program will eventually halt on the given input.*

In this form the result is called the *unsolvability of the halting problem.* We reason as follows: if there were such an algorithm, we could use it to check the truth or falsity of HALT(x, y) for given x, y by first obtaining program $\mathscr{Q}$ with $\#(\mathscr{Q}) = y$ and then checking whether $\mathscr{Q}$ eventually halts on input x. But we have reason to believe that *any algorithm for computing on numbers can be carried out by a program of $\mathscr{L}$.*

The last italicized assertion is a form of what has come to be called *Church's thesis.* But, since the word *algorithm* has no general definition separated from a particular language, Church's thesis cannot be proved as a mathematical theorem.

In the light of Church's thesis, Theorem 2.1 tells us that there really is no algorithm for testing a given program and input to determine whether it will ever halt. Anyone who finds it surprising that no algorithm exists for such a "simple" proble should be made to realize that it is easy to construct relatively short programs (of $\mathscr{L}$) such that nobody is in a position to tell whether they will ever halt. For example, consider the assertion from number theory that every even number ≥ 4 is the sum of two prime numbers. This assertion, known as *Goldbach's conjecture*, is clearly true for small even numbers.

3 Universality

One of the key tools in computability theory is

Theorem 3.1: Universality Theorem

For each $n > 0$, the function $\Phi^{(n)}(x_1, \ldots, x_n, y)$ is partially computable.

We shall prove this theorem by showing how to construct, for each $n > 0$, a program $\mathscr{U}_n$ which computes $\Phi^{(n)}$. The programs $\mathscr{U}_n$ are called *universal.* For example, $\mathscr{U}_1$ can be used to compute *any* partially computable function of one variable, namely, if $f(x)$ is computed by a program $\mathscr{P}$ and $y = \#(\mathscr{P})$, then $f(x) = \Phi^{(1)}(x, y) = \psi^{(2)}_{\mathscr{U}_1}(x, y)$.

Notice in particular that the input variables are those whose position in our list is an *even* number.

We proceed to give the program $\mathscr{U}_n$ for computing

$$Y = \Phi^{(n)}(X_1, \ldots, X_n, X_{n+1}).$$

We begin by exhibiting $\mathscr{U}_n$ in sections, explaining what each part does.

We begin:

$$\begin{aligned} Z &\leftarrow X_{n+1} + 1 \\ S &\leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i} \\ K &\leftarrow 1 \end{aligned}$$

Next,

$$[C] \text{ IF } K = \text{Lt}(Z) + 1 \vee K = 0 \text{ GOTO } F$$

If the computation has ended GOTO F, where the proper value will be output. Otherwise, the current instruction must be decoded and executed:

$$\begin{aligned} U &\leftarrow r((Z)_K) \\ P &\leftarrow p_{r(U)+1} \end{aligned}$$

$(Z)_K = \langle a, \langle b, c \rangle \rangle$ is the number of the Kth instruction. Thus, $U = \langle b, c \rangle$ is the code for the *statement* about to be executed. The variable mentioned in the K th instruction is the $(c+1)$th, i.e., the $(r(U)+1)$th, in our list. Thus, its current value is stored as the exponent to which P divides S:

$$\begin{array}{l} \text{IF } l(U) = 0 \text{ GOTO } N \\ \text{IF } l(U) = 1 \text{ GOTO } A \\ \text{IF } \sim (P \mid S) \text{ GOTO } N \\ \text{IF } l(U) = 2 \text{ GOTO } M \end{array}$$

If $l(U) \neq 0, 1$, then the current instruction is either of the form $V \leftarrow V - 1$ or IF $V \neq 0$ GOTO L. In either case, if P is not a divisor of S, i.e., if the current value of V is 0, the computation need to do *nothing* to S.

A simple modification of the programs $\mathscr{U}_n$ would enable us to prove that the predicates

$$\begin{array}{lcl} \text{STP}^{(n)}(x_1, \ldots, x_n, y, t) & \Leftrightarrow & \text{Program number } y \text{ halts after } t \text{ or fewer} \\ & & \text{steps on inputs } x_1, \ldots, x_n \\ & \Leftrightarrow & \text{There is a computation of program } y \text{ of} \\ & & \text{length} \leq t+1, \text{ beginning with inputs} \\ & & x_1, \ldots, x_n \end{array}$$

are computable. We simply need to add a counter to determine when we have simulated t steps. However, we can prove a stronger result.

Theorem 3.2: Step-Counter Theorem

For each $n > 0$, the predicate $\text{STP}^{(n)}(x_1, \ldots, x_n, y, t)$ is primitive recursive.

By using the technique of the above proof, we can obtain the following important result

Theorem 3.3: Normal Form Theorem

Let $f(x_1, \ldots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \ldots, x_n, y)$ such that

$$f(x_1, \ldots, x_n) = l\left(\min_z R(x_1, \ldots, x_n, z)\right).$$

The normal form theorem leads to another characterization of the class of partially computable functions.

Theorem 3.4

A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization.

When $\min_y R(x_1, \ldots, x_n, y)$ is a total function [that is, when for each $x_1, \ldots, x_n$ there is at least one y for which $R(x_1, \ldots, x_n, y)$ is true], we say that we are applying the operation of *proper minimalization* to R. Now, if

$$l\left(\min_y R(x_1, \ldots, x_n, y)\right)$$

is total, then $\min_y R(x_1, \ldots, x_n, y)$ must be total. Hence we have

Theorem 3.5

A function is computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and *proper* minimalization.

4 Recursively Enumerable Sets

To say that a set B, where $B \subseteq N^m$, belongs to some class of *functions* means that the characteristic function $P(x_1, \ldots, x_m)$ of B belongs to the class in question.

We have, for example,

Theorem 4.1

Let the sets B, C belong to some PRC class $\mathscr{C}$. Then so do the sets $B \cup C, B \cap C, \overline{B}$.

We have, for example,

Theorem 4.2

Let $\mathscr{C}$ be a PRC class, and let B be a subset of $N^m, m \geq 1$. Then B belongs to $\mathscr{C}$ if and only if

$$B' = \{[x_1, \ldots, x_m] \in N \mid (x_1, \ldots, x_m) \in B\}$$

belongs to $\mathscr{C}$.

It immediately follows, for example, that $\{[x, y] \in N \mid \mathrm{HALT}(x, y)\}$ is not a computable set.

Definition 4.3

The set $B \subseteq N$ is called *recursively enumerable* if there is a partially computable function $g(x)$ such that

$$B = \{x \in N \mid g(x) \downarrow\}. \tag{4.2}$$

The term *recursively enumerable* is usually abbreviated *r.e.* If $\mathscr{P}$ is a program that computes the function g in (4.2), then B is simply the set of all inputs to $\mathscr{P}$ for which $\mathscr{P}$ eventually halts. If we think of $\mathscr{P}$ as providing an algorithm for testing for membership in B, we see that for numbers that do belong to B, the algorithm will provide "yes" answer; but for numbers that do not, the algorithm will never terminate. Such algorithms, sometimes called *semi-decision procedures*, provide a kind of "approximation" to solving the problem of testing membership in B.

Theorem 4.4

If B is a recursive set, then B is r.e.