Second Edition

Computability, Complexity, and Languages

Fundamentals of Theoretical Computer Science

Martin D. Davis

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, New York

Ron Sigal

Departments of Mathematics and Computer Science
Yale University
New Haven, Connecticut

Elaine J. Weyuker

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, New York

ACADEMIC PRESS
Harcourt, Brace & Company
Boston San Diego New York
London Sydney Tokyo Toronto

Contents

Preface

Theoretical computer science is the mathematical study of models of computation. As such, it originated in the 1930s, well before the existence of modern computers, in the work of the logicians Church, Gödel, Kleene, Post, and Turing. This early work has had a profound influence on the practical and theoretical development of computer science. Not only has the Turing machine model proved basic for theory, but the work of these pioneers presaged many aspects of computational practice that are now commonplace and whose intellectual antecedents are typically unknown to users. Included among these are the existence in principle of all-purpose (or universal) digital computers, the concept of a program as a list of instructions in a formal language, the possibility of interpretive programs, the duality between software and hardware, and the representation of languages by formal structures, based on productions. While the spotlight in computer science has tended to fall on the truly breathtaking technological advances that have been taking place, important work in the foundations of the subject has continued as well. It is our purpose in writing this book to provide an introduction to the various aspects of theoretical computer science for undergraduate and graduate students that is sufficiently comprehensive that the professional literature of treatises and research papers will become accessible to our readers.

We are dealing with a very young field that is still finding itself. Computer scientists have by no means been unanimous in judging which parts of the subject will turn out to have enduring significance. In this situation, fraught with peril for authors, we have attempted to select topics that have already achieved a polished classic form, and that we believe will play an important role in future research.

In this second edition, we have included new material on the subject of programming language semantics, which we believe to be established as an important topic in theoretical computer science. Some of the material on computability theory that had been scattered in the first edition has been brought together, and a few topics that were deemed to be of only peripheral interest to our intended audience have been eliminated. Numerous exercises have also been added. We were particularly pleased to be able to include the answer to a question that had to be listed as open in the first edition. Namely, we present Neil Immerman's surprisingly straightforward proof of the fact that the class of languages accepted by linear bounded automata is closed under complementation.

We have assumed that many of our readers will have had little experience with mathematical proof, but that almost all of them have had substantial programming experience. Thus the first chapter contains an introduction to the use of proofs in mathematics in addition to the usual explanation of terminology and notation. We then proceed to take advantage of the reader's background by developing computability theory in the context of an extremely simple abstract programming language. By systematic use of a macro expansion technique, the surprising power of the language is demonstrated. This culminates in a universal program, which is written in all detail on a single page. By a series of simulations, we then obtain the equivalence of various different formulations of computability, including Turing's. Our point of view with respect to these simulations is that it should not be the reader's responsibility, at this stage, to fill in the details of vaguely sketched arguments, but rather that it is our responsibility as authors to arrange matters so that the simulations can be exhibited simply, clearly, and completely.

This material, in various preliminary forms, has been used with undergraduate and graduate students at New York University, Brooklyn College, The Scuola Matematica Interuniversitaria -Perugia, The University of California-Berkeley, The University of California-Santa Barbara, Worcester Polytechnic Institute, and Yale University.

Although it has been our practice to cover the material from the second part of the book on formal languages after the first part, the chapters on regular and on context-free languages can be read immediately after Chapter 1. The Chomsky-Schiitzenberger representation theorem for context-free languages in used to develop their relation to pushdown automata in a way that we believe is clarifying. Part 3 is an exposition of the aspects of logic that we think are important for computer science and can also be read immediately following Chapter 1. Each of the chapters of Part 4 introduces an important theory of computational complexity, concluding with the theory of NP-completeness. Part 5, which is new to the second edition, uses recursion equations to expand upon the notion of computability developed in Part 1, with an emphasis on the techniques of formal semantics, both denotational and operational. Rooted in the early work of Gödel, Herbrand, Kleene, and others, Part 5 introduces ideas from the modern fields of functional programming languages, denotational semantics, and term rewriting systems.

Because many of the chapters are independent of one another, this book can be used in various ways. There is more than enough material for a full-year course at the graduate level on ***theory of computation.*** We have used the unstarred sections of Chapters 1-6 and Chapter 9 in a successful one-semester junior-level course, Introduction to Theory of Computation, at New York University. A course on ***finite automata and formal languages*** could be based on Chapters 1, 9, and 10. A semester or quarter course on ***logic for computer scientists*** could be based on selections from Parts 1 and 3. Part 5 could be used for a third semester on the theory of computation or an introduction to ***programming language semantics.*** Many other arrangements and courses are possible, as should be apparent from the dependency graph, which follows the Acknowledgments. It is our hope, however, that this book will help readers to see theoretical computer science not as a fragmented list of discrete topics, but rather as a unified subject drawing on powerful mathematical methods and on intuitions derived from experience with computing technology to give valuable insights into a vital new area of human knowledge.

Note to the Reader

Many readers will wish to begin with Chapter 2, using the material of Chapter 1 for reference as required. Readers who enjoy skipping around will find the ***dependency graph*** useful.

Sections marked with an asterisk (*) may be skipped without loss of continuity. The relationship of these sections to later material is given in the dependency graph.

Exercises marked with an asterisk either introduce new material, refer to earlier material in ways not indicated in the dependency graph, or simply are considered more difficult than unmarked exercises.

A reference to Theorem 8.1 is to Theorem 8.1 of the chapter in which the reference is made. When a reference is to a theorem in another chapter, the chapter is specified. The same system is used in referring to numbered formulas and to exercises.

Acknowledgments

It is a pleasure to acknowledge the help we have received. Charlene Herring, Debbie Herring, Barry Jacobs, and Joseph Miller made their student classroom notes available to us. James Cox, Keith Harrow, Steve Henkind, Karen Lemone, Colm O'Dunlaing, and James Robinett provided helpful comments and corrections. Stewart Weiss was kind enough to redraw one of the figures. Thomas Ostrand, Norman Shulman, Louis Salkind, Ron Sigal, Patricia Teller, and Elia Weixelbaum were particularly generous with their time, and devoted many hours to helping us. We are especially grateful to them.

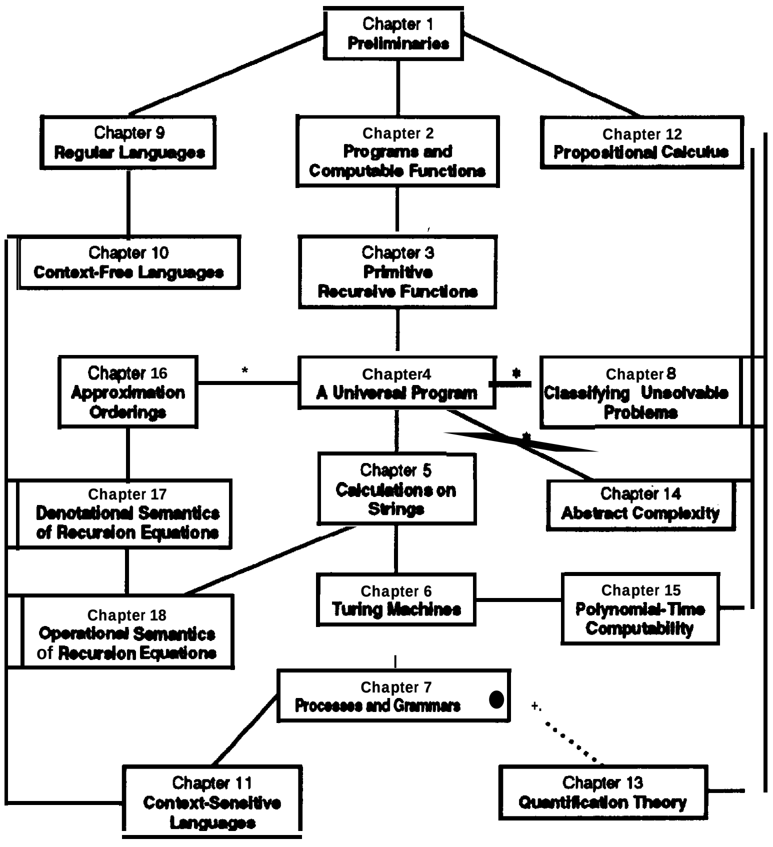
Acknowledgments to Corrected Printing

We have taken this opportunity to correct a number of errors. We are grateful to the readers who have called our attention to errors and who have suggested corrections. The following have been particularly helpful: Alissa Bernholc, Domenico Cantone, John R. Cowles, Herbert Enderton, Phyllis Frankl, Fred Green, Warren Hirsch, J. D. Monk, Steve Rozen, and Stewart Weiss.

Acknowledgments to Second Edition

Yuri Gurevich, Paliath Narendran, Robert Paige, Carl Smith, and particularly Robert McNaughton made numerous suggestions for improving the first edition. Kung Chen, William Hurwood, Dana Latch, Sidd Puri, Benjamin Russell, Jason Smith, Jean Toal, and Niping Wu read a preliminary version of Part 5.

Dependency Graph


Chapter 1 Preliminaries
Chapter 9 Regular Languages
Chapter 2 Programs and Computable Functions
Chapter 12 Propositional Calculus
Chapter 10 Context-Free Languages
Chapter 3 Primitive Recursive Functions
Chapter 16 Approximation Orderings
*
Chapter 4 A Universal Program
*
Chapter 8 Classifying Unsolvable Problems
*
Chapter 17 Denotational Semantics of Recursion Equations
Chapter 5 Calculations on Strings
Chapter 14 Abstract Complexity
Chapter 18 Operational Semantics of Recursion Equations
Chapter 6 Turing Machines
Chapter 15 Polynomial-Time Computability
Chapter 7 Processes and Grammars
Chapter 11 Context-Sensitive Languages
Chapter 13 Quantification Theory

A solid line between two chapters indicates the dependence of the unstarred sections of the higher numbered chapter on the unstarred sections of the lower numbered chapter. An asterisk next to a solid line indicates that knowledge of the starred sections of the lower numbered chapter is also assumed. A dotted line shows that knowledge of the unstarred sections of the lower numbered chapter is assumed for the starred sections of the higher numbered 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 of 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*** $0, 1, 2, 3, \ldots$. In this book the word ***number*** will always mean ***natural number*** except in contexts where the contrary is explicitly stated.

We write

$$a \in S$$

to mean that a belongs to S or, equivalently, is a member of the set S, and

$$a \notin S$$

to mean that a does not belong to S. It is useful to speak of the ***empty set,*** written 0, 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. Thus, $R = S$ if and only if $R \subseteq S$ and $S \subseteq R$. Note also that for any set R, $0 \subseteq R$ and $R \subseteq R$. We write R c 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 sets, we write R U 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* n 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. S may contain objects not in R. Thus ***R* – *S* = *R* – (*R* n 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. Most frequently we shall be writing $\bar{S}$ for N – S. The De Morgan identities

$$\overline{R \cup S} = \bar{R} \cap \bar{S},$$

$$\overline{R \cap S} = \bar{R} \cup \bar{S}$$

are very useful; they are easy to check and any reader not already familiar with them should do so. We write

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

for the set consisting of the n objects ***a,,*** $a_2, \ldots,$ ***a,,*** . Sets that can be written in this form as well as the empty set are called ***jinite.*** Sets that are not finite, e.g., N, are called ***infinite.*** It should be carefully noted that a and {a} are not the same thing. In particular, $a \in S$ is true if and only if $\{a\} \subseteq$ S. Since two sets are equal if and only if they have the same members, it follows that, for example, ***{a, b, c} = {a, c, b} = {b, a, c}.*** 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.*** We write n-tuples using parentheses rather than curly braces:

$$(a_1, \ldots, a_n).$$

Naturally, the elements making up an n-tuple need not be distinct. Thus $(4,1,4,2)$ is a 4-tuple. 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

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

if ***and only if***

$$a, = b_1, \qquad a_2 = b_2, \qquad \ldots, \qquad \textit{and} \qquad a,, = b_n.$$

If $S_1, S_2, \ldots, S_n$ are given sets, then we write $S_1 \times S_2 \times \cdots \times S_n$ **for** the set of all n-tuples $(a_1, a_2, \ldots, a_n)$ such that $a_1 \in S_1, a_2 \in S_2, \ldots, a_n \in S_n$.

$S_1 \times S_2 \times \cdots \times S_n$ is sometimes called the ***Cartesian product*** of $S_1, S_2, \ldots, S_n$. In case $S_1 = S_2 = \cdots = S_n = S$ we write S^n for the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$.

2. Functions

Functions play an important role in virtually every branch of pure and applied mathematics. We may define a function simply as a set f, all of whose members are ordered pairs and that has the special property

$$(a, b) \in f \text{ and } (a, c) \in f \quad \text{implies } b = c.$$

However, intuitively it is more helpful to think of the pairs listed as the rows of a table. 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.

As an example, let f be the set of ordered pairs (n, n^2) for $n \in N$. Then, for each $n \in N$, $f(n) = n^2$. The domain of f is N. The range of f is the set of perfect squares.

Functions f are often specified by ***algorithms*** that provide procedures for obtaining $f(a)$ from a. This method of specifying functions is particularly important in computer science. However, as we shall see in Chapter 4, 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. An example of a partial function on N is given by $g(n) = \sqrt{n}$, where the domain of g is the set of perfect squares. 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 $\emptyset$ is itself a function. Considered as a partial function on some set S, *it is nowhere defined.*

For a partial function f on a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$, we write $f(a_1, \ldots, a_n)$ rather than $f((a_1, \ldots, a_n))$. 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. For n-ary partial functions, we often write $f(x_1, \ldots, x_n)$ instead of f as a way of showing explicitly that f is n-ary.

Sometimes it is useful to work with particular kinds of functions. A function f is ***one-one*** if, for all x, y in the domain of f, $f(x)$ = f(y) implies x = y. Stated differently, if $x \neq$ y then $f(x) \neq$ f(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.*** For example, f(n) = n^2 is one-one, and f is onto the set of perfect squares, but it is not onto N.

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. For example, N is closed under $f(n) = n^2$, but it is not closed under $h(n) = \sqrt{n}$ (where h is a total function on N).

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. Instead of writing a word as ***(a„ $a_2, \ldots,$ a„) we*** write simply $a_1a_2 \cdots$ ***a,.*** If $u = a_1a_2 \cdots$ ***a,,*** then we say that n is the length of u and write $|u| = n$. We allow a unique null word, written 0, of length 0. (The reason for using the same symbol for the number zero and the null word will become clear in Chapter 5.) 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. If u, v $\in A^*$, then we write $\widehat{uv}$ for the word obtained by placing the string v after the string u. For example, if $A = \{a, b, c\}$, $u = bab$, and v = ***caa,*** then

$$\widehat{uv} = babcaa \qquad \text{and} \qquad \widehat{vu} = caabab.$$

Where no confusion can result, we write uv instead of $\widehat{uv}$. It is obvious that, for all u,

$$\text{uo} = \text{ou} = u,$$

and that, for all u, v, w,

$$u(vw) = (uv)w.$$

Also, if either uv = uw or vu = wu, then v = w.

If u is a string, and $n \in N$, $n > 0$, ***we*** write

$$u^{[n]} = \underbrace{uu \cdots u}_{n}.$$

We also write $u^{[0]} = 0$ We use the square brackets to avoid confusion with numerical exponentiation.

If $u \in A^*$, we write u^R for u written backward; i.e., if $u = a_1 a_2 \cdots a_n$ for $a_n, \ldots, a_n \in A$, then $u^R = a_n \cdots a_2 a_1$. Clearly, $0^R = 0$ and $(uv)^R = v^R u^R$ for $u, v \in A^*$.

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

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

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. For our purposes it is useful to identify the truth values with specific numbers, so we set

$$\text{TRUE} = 1 \qquad \text{and} \qquad \text{FALSE} = 0.$$

Thus, a predicate is a special kind of function with values in N. Predicates on a set S are usually specified by expressions which become statements, either true or false, when variables in the expression are replaced by symbols designating fixed elements of S. Thus the expression

$$x < 5$$

specifies a predicate on N, namely,

$$P(x) = \begin{cases} 1 & \text{if } x = 0,1,2,3,4 \\ 0 & \text{otherwise.} \end{cases}$$

Three basic operations on truth values are defined by the tables in Table 4.1. Thus if P and Q are predicates on a set S, there are also the predicates $\sim P$, $P \,\&\, Q$, $P \vee Q$. $\sim P$ is true just when P is false; $P \,\&\, Q$ is true when both P and Q are true, otherwise it is false; $P \vee Q$ is true when either P or Q or both are true, otherwise it is false. Given a predicate P

Table 4.1

P	$\sim p$
0	1
1	0

P	4	$p \,\&\, q$	$p \vee q$
1	1	1	1
0	1	0	1
1	0	0	1
0	0	0	0

on a set S, there is a corresponding subset R of S, namely, the set of all elements $a \in S$ for which $P(a) = 1$. We write

$$R = \{a \in S | P(a)\}.$$

Conversely, given a subset R of a given set S, the expression

$$x \in R$$

defines a predicate on S, namely, the predicate defined by

$$P(x) = \begin{cases} 1 & \text{if } x \in R \\ 0 & \text{if } x \notin R. \end{cases}$$

Of course, in this case,

$$R = \{x \in S | P(x)\}.$$

The predicate P is called the *characteristic function* of the set R. The close connection between sets and predicates is such that one can readily translate back and forth between discourse involving one of these notions and discourse involving the other. Thus we have

$$\{x \in S \mid P(x) \,\&\, Q(x)\} = \{x \in S \mid P(x)\} \cap \{x \in S \mid Q(x)\},$$

$$\{x \in S \mid P(x) \vee Q(x)\} = \{x \in S \mid P(x)\} \cup \{x \in S \mid Q(x)\},$$

$$\{x \in S \mid {\sim}P(x)\} = S - \{x \in S \mid P(x)\}.$$

To indicate that two expressions containing variables define the same predicate we place the symbol $\Leftrightarrow$ between them. Thus,

$$x < 5 \Leftrightarrow x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \vee x = 4.$$

The De Morgan identities from Section 1 can be expressed as follows in terms of predicates on a set S:

$$P(x) \,\&\, Q(x) \Leftrightarrow {\sim}({\sim}P(x) \vee {\sim}Q(x)),$$

$$P(x) \vee Q(x) \Leftrightarrow {\sim}({\sim}P(x) \,\&\, {\sim}Q(x)).$$

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. Here and later we omit the phrase "on N" when the meaning is clear.

Thus, let $P(t, x_1, \ldots, x_n)$ be an (n + 1)-ary predicate. Consider the predicate $Q(y, x_1, \ldots, x_n)$ defined by

$$Q(y, x_1, \ldots, x_n) \Leftrightarrow P(0, x_1, \ldots, x_n) \vee P(1, x_1, \ldots, x_n)$$
$$\vee \cdots \vee P(y, x_1, \ldots, x_n).$$

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

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

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

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

This predicate is true just in case $P(t, x_1, \ldots, x_n)$ is true for ***all*** $t \le y$. *The* expression "$(\forall t)_{\le y}$" is called a *bounded* ***universal*** *quantifier.* We also write $(\exists t)_{< y} P(t, x_1, \ldots, x_n)$ for the predicate that is true just in case $P(t, x_1, \ldots, x_n)$ is true for at least one value of $t < y$ and $(\forall t)_{< y} P(t, x_1, \ldots, x_n)$ for the predicate that is true just in case $P(t, x_1, \ldots, x_n)$ is true for all values of $t < y$.

We write

$$Q(x_1, \ldots, x_n) \Leftrightarrow (\exists t) P(t, x_1, \ldots, x_n)$$

for the predicate which is true if there exists some $t \in N$ for which $P(t, x_1, \ldots, x_n)$ is true. Similarly, $(\forall t) P(t, x_1, \ldots, x_n)$ is true if $P(t, x_1, \ldots, x_n)$ is true for all $t \in N$.

The following generalized De Morgan identities are sometimes useful:

$$\sim (\exists t)_{\le y} P(t, x_1, \ldots, x_n) \Leftrightarrow (\forall t)_{\le y} \sim P(t, x_1, \ldots, x_n),$$
$$\sim (\exists t) P(t, x_1, \ldots, x_n) \Leftrightarrow (\forall t) \sim P(t, x_1, \ldots, x_n).$$

The reader may easily verify the following examples:

$$(\exists y)(x + y = 4) \Leftrightarrow x \le 4,$$
$$(\exists y)(x + y = 4) \Leftrightarrow (\exists y)_{\le 4}(x + y = 4),$$
$$(\forall y)(xy = 0) \Leftrightarrow x = 0,$$
$$(\exists y)_{\le z}(x + y = 4) \Leftrightarrow (x + z \ge 4 \,\&\, x \le 4).$$

6. Proof by Contradiction

In this book we will be calling many of the assertions we make ***theorems*** (or *corollaries* or ***lemmas)*** and providing ***proofs*** that they are correct. Why are proofs necessary? The following example should help in answering this question.

Recall that a number is called a ***prime*** if it has ***exactly two distinct divisors,*** itself and 1. Thus 2, 17, and 41 are primes, but 0, 1, 4, and 15 are not. Consider the following assertion:

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

This assertion is in fact ***false.*** Namely, for $n = 41$ the expression becomes

$$41^2 - 41 + 41 = 41^2,$$

which is certainly not a prime. However, the assertion is true (readers with access to a computer can easily check this!) for all $n \le 40$. This example shows that inferring a result about all members of an infinite set (such as N) from even a large finite number of instances can be very dangerous. A proof is intended to overcome this obstacle.

A proof begins with some initial statements and uses logical reasoning to infer additional statements. (In Chapters 12 and 13 we shall see how the notion of logical reasoning can be made precise; but in fact, our use of logical reasoning will be in an informal intuitive style.) When the initial statements with which a proof begins are already accepted as correct, then any of the additional statements inferred can also be accepted as correct. But proofs often cannot be carried out in this simple-minded pattern. In this and the next section we will discuss more complex proof patterns.

In a ***proof by contradiction,*** one begins by supposing that the assertion we wish to prove is false. Then we can feel free to use the negation of what we are trying to prove as one of the initial statements in constructing a proof. In a proof by contradiction we look for a pair of statements developed in the course of the proof which ***contradict*** one another. Since both cannot be true, we have to conclude that our original supposition was wrong and therefore that our desired conclusion is correct.

We give two examples here of proof by contradiction. There will be many in the course of the book. Our first example is quite famous. We recall that every number is either even (i.e., $= 2n$ for some $n \in N$) or odd (i.e., $= 2n + 1$ for some $n \in N$). Moreover, if m is even, $m = 2n$, then $m^2 = 4n^2 = 2 \cdot 2n^2$ is even, while if m is odd, $m = 2n + 1$, then $m^2 = 4n^2 + 4n + 1 = 2(2n^2 + 2n) + 1$ is odd. We wish to prove that the equation

$$2 = (m/n)^2 \tag{6.1}$$

has no solution for m, $n \in N$ (that is, that $\sqrt{2}$ is not a "rational" number). We suppose that our equation has a solution and proceed to derive a contradiction. Given our supposition that (6.1) has a solution, it must have a solution in which m and n are not both even numbers. This is true because if m and n are both even, we can repeatedly "cancel" 2 from numerator and denominator until at least one of them is odd. On the other hand, we shall prove that for every solution of (6.1) m and n must both be even. The contradiction will show that our supposition was false, i.e., that (6.1) has no solution.

It remains to show that in every solution of (6.1), m and n are both even. We can rewrite (6.1) as

$$m^2 = 2n^2,$$

which shows that m^2 is even. As we saw above this implies that m is even, say $m = 2k$. Thus, $m^2 = 4k^2 = 2n^2$, or $n^2 = 2k^2$. Thus, n^2 is even and hence n is even. ■

Note the symbol ■ , which means "the proof is now complete."

Our second example involves strings as discussed in Section 3.

Theorem 6.1. Let $x \in \{a, b\}^*$ such that $xa = ax$. Then $x = a^{[n]}$ for some $n \in N$.

Proof. Suppose that $xa = ax$ but x contains the letter b. Then we can write $x = a^{[n]}bu$, where we have explicitly shown the first (i.e., leftmost) occurrence of b in x. Then

$$a^{[n]}bua = aa^{[n]}bu = a^{[n+1]}bu.$$

Thus,

$$bua = abu.$$

But this is impossible, since the same string cannot have its first symbol be both b and a. This contradiction proves the theorem. ■

Exercises

1. Prove that the equation $(p/q)^2 = 3$ has no solution for $p, q \in N$.
2. Prove that if $x \in \{a, b\}^*$ and $abx = xab$, then $x = (ab)^{[n]}$ 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

$$(\forall n)(\textit{If } P(n) \textit{ then } P(n+1)). \tag{7.1}$$

Once we have succeeded in proving these auxiliary statements we can regard $(\forall n)P(n)$ as also proved. The justification for this is as follows.

From the second auxiliary statement we can infer each of the infinite set of statements:

$$\textit{If } P(0) \textit{ then } P(1),$$
$$\textit{If } P(1) \textit{ then } P(2),$$
$$\textit{If } P(2) \textit{ then } P(3), \ldots.$$

Since we have proved $P(0)$, we can infer $P(1)$. Having now proven $P(1)$ we can get $P(2)$, etc. Thus, we see that $P(n)$ is true for all n and hence $(\forall n)P(n)$ is true.

Why is this helpful? Because sometimes it is much easier to prove (7.1) 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.*** This methodology enables us to use $P(k)$ as one of the initial statements in the proof we are constructing.

There are some paradoxical things about proofs by mathematical induction. One is that considered superficially, it seems like an example of circular reasoning. One seems to be assuming $P(k)$ for an arbitrary k, which is exactly what one is supposed to be engaged in proving. Of course, one is not really assuming $(\forall n)P(n)$. 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 can put this schematically as follows. 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). Proving the stronger statement by induction requires that we prove

$$P(0) \,\&\, Q(0)$$

and

$$(\forall n)[\textit{If } P(n) \,\&\, Q(n) \textit{ then } P(n+1) \,\&\, Q(n+1)].$$

In proving this second auxiliary statement, we may take $P(k) \,\&\, Q(k)$ as our induction hypothesis. Thus, although strengthening the statement to be proved gives us more to prove, it also gives us a stronger induction hypothesis and, therefore, more to work with. The technique of deliberately strengthening what is to be proven for the purpose of making proofs by induction easier is called ***induction loading.***

It is time for an example of a proof by induction. The following is useful in doing one of the exercises in Chapter 6.

Theorem 7.1. For all $n \in N$ we have $\sum_{i=0}^{n}(2i + 1) = (n + 1)^2$.

Proof. For $n = 0$, our theorem states simply that $1 = 1^2$, which is true.

Suppose the result known for $n = k$. That is, our induction hypothesis is

$$\sum_{i=0}^{k} (2i + 1) = (k + 1)^2.$$

Then

$$\begin{aligned}\sum_{i=0}^{k+1} (2i + 1) &= \sum_{i=0}^{k}(2i + 1) + 2(k + 1) + 1\\ &= (k + 1)^2 + 2(k + 1) + 1\\ &= (k + 2)^2.\end{aligned}$$

But this is the desired result for $n = k + 1$. ■

Another form of mathematical induction that is often very useful is called ***course-of-values induction*** or sometimes ***complete induction.*** In the case of course-of-values induction we prove the single auxiliary statement

$$(\forall n)[\textit{If } (\forall m)_{m<n}P(m) \textit{ then } P(n)], \tag{7.2}$$

and then conclude that $(\forall n)P(n)$ is true. A potentially confusing aspect of course-of-values induction is the apparent lack of an initial statement $P(0)$. But in fact there is no such lack. The case $n = 0$ of *(7.2)* is

$$\textit{If } (\forall m)_{m<0}P(m) \textit{ then } P(0).$$

But the "induction hypothesis" $(\forall m)_{m<0}P(m)$ is entirely vacuous because there is no $m \in N$ such that $m < 0$. So in proving (7.2) for $n = 0$ we really are just proving $P(0)$. In practice it is sometimes possible to give a single proof of (7.2) that works for all n including $n = 0$. But often the case $n = 0$ has to be handled separately.

To see why course-of-values induction works, consider that, in the light of what we have said about the $n = 0$ case, (7.2) leads to the following infinite set of statements:

$$P(0),$$
$$\textit{If } P(0) \textit{ then } P(1),$$
$$\textit{If } P(0) \,\&\, P(1) \textit{ then } P(2),$$
$$\textit{If } P(0) \,\&\, P(1) \,\&\, P(2) \textit{ then } P(3),$$

Here is an example of a theorem proved by course-of-values induction.

Theorem 7.2. There is no string $x \in \{a, b\}^*$ such that $ax = xb$.

Proof. Consider the following predicate: *If* $x \in \{a, b\}^*$ and $|x| = n$, *then* $ax \neq xb$. We will show that this is true for all $n \in N$. *So we* assume it true for all $m < k$ for some given k and show that it follows for k. This proof will be by contradiction. Thus, suppose that $|x| = k$ and $ax = xb$. The equation implies that a is the first and b the last symbol in x. So, we can write $x = aub$. Then

$$aaub = aubb,$$

i.e.,

$$au = ub.$$

But $|u| < |x|$. Hence by the induction hypothesis $au \neq ub$. This contradiction proves the theorem. ■

Proofs by course-of-values induction can always be rewritten so as to involve reference to the principle that if some predicate is true for some element of N, then there must be a least element of N for which it is true. Here is the proof of Theorem 7.2 given in this style.

Proof. Suppose there is a string $x \in \{a, b\}^*$ such that $ax = xb$. Then there must be a string satisfying this equation of minimum length. Let x be such a string. Then $ax = xb$, but, if $|u| < |x|$, then $au \neq ub$. However, $ax = xb$ implies that $x = aub$, *so* that $au = ub$ and $|u| < |x|$. This contradiction proves the theorem. ■

Exercises

1. Prove by mathematical induction that $\sum_{i=1}^{n} i = n(n+1)/2$.
2. Here is a "proof' by mathematical induction that if $x, y \in N$, then $x = y$. What is wrong?

Let

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

for $x, y \in N$. Consider the predicate

$$(\forall x)(\forall y)[\textit{If } \max(x, y) = n, \textit{ then } x = y].$$

For $n = 0$, this is clearly true. Assume the result for $n = k$, and let $\max(x, y) = k + 1$. Let $x_1 = x - 1$, $y_1 = y - 1$. Then $\max(x_1, y_1) = k$. By the induction hypothesis, $x_1 = y_1$ and therefore $x = x_1 + 1 = y_1 + 1 = y$.

3. Here is another incorrect proof that purports to use mathematical induction to prove that all flowers have the same color! What is wrong?

Consider the following predicate: If S is a set of flowers containing exactly n elements, then all the flowers in S have the same color. The predicate is clearly true if $n = 1$. We suppose it true for $n = k$ and prove the result for $n = k + 1$. Thus, let S be a set of $k + 1$ flowers. If we remove one flower from S we get a set of k flowers. Therefore, by the induction hypothesis they all have the same color. Now return the flower removed from S and remove another. Again by our induction hypothesis the remaining flowers all have the same color. But now both of the flowers removed have been shown to have the same color as the rest. Thus, all the flowers in S have the same color.

4. Show that there are no strings $x, y \in \{a, b\}^*$ such that $xay = ybx$.

5. Give a "one-line" proof of Theorem 7.2 that does not use mathematical induction.

2

Programs and Computable Functions

1. A Programming Language

Our development of computability theory will be based on a specific programming language $\mathscr{S}$. We will use certain letters as variables whose values are ***numbers.*** (In this book the word ***number*** will always mean nonnegative integer, unless the contrary is specifically stated.) In particular, the letters

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

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

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

will be called the ***local variables*** of $\mathscr{S}$. The subscript 1 is often omitted; i.e., X stands for X_1 and Z for Z_1. Unlike the programming languages in actual use, there is no upper limit on the values these variables can assume. Thus from the outset, $\mathscr{S}$ must be regarded as a purely theoretical entity. Nevertheless, readers having programming experience will find working with $\mathscr{S}$ very easy.

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

Table 1.1

Instruction	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 $V \neq 0$ GOT0 L	If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list.

instructions. For example, for each variable V there will be an instruction:

$$V \leftarrow V + 1$$

A simple example of a program of $\mathscr{S}$ is

$$X \leftarrow X + 1$$
$$X \leftarrow X + 1$$

"Execution" of this program has the effect of increasing the value of X by 2. In addition to variables, we will need "labels." In $\mathscr{S}$ these are

$$A_1\quad B_1\ C_1\ D_1\ E_1\quad A_2\quad B_2\ C_2\ D_2\ E_2\quad A_3\quad \cdots .$$

Once again the subscript 1 can be omitted. We give in Table 1.1 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.*** We will sometimes indicate the value of a variable by writing it in lowercase italics. Thus x_5 is the value of X_5.

Instructions may or may not have labels. When an instruction is labeled, the label is written to its left in square brackets. For example,

$$[B] \qquad Z \leftarrow Z - 1$$

In order to base computability theory on the language $\mathscr{S}$, we will require formal definitions. But before we supply these, it is instructive to work informally with programs of $\mathscr{S}$.

2. Some Examples of Programs

(a) Our first example is the program

```
[A]     X ← X − 1
        Y ← Y + 1
        IF X ≠ 0 GOTO A
```

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. (By our conventions the initial value of Y is 0.) If $x = 0$, then the program halts with Y having the value 1. We will say that this program ***computes*** the function

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

This program halts when it executes the third instruction of the program with X having the value 0. In this case the condition X ≠ 0 is not fulfilled and therefore the branch is not taken. When an attempt is made to move on to the nonexistent fourth instruction, the program halts. A program will also halt if an instruction labeled L is to be executed, but there is no instruction in the program with that label. In this case, we usually will use the letter E (for "exit") as the label which labels no instruction.

(b) Although the preceding program is a perfectly well-defined program of our language $\mathscr{S}$, 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.

```
[A]     IF $X \neq 0$ GOTO $B$
        $Z \leftarrow Z + 1$
        IF $Z \neq 0$ GOTO $E$
[B]     $X \leftarrow X - 1$
        $Y \leftarrow Y + 1$
        $Z \leftarrow Z + 1$
        IF $Z \neq 0$ GOTO $A$
```

As we can easily convince ourselves, this program does copy the value of X into Y for all initial values of X. Thus, we say that it computes the function $f(x) = x$. 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

```
$Z \leftarrow Z + 1$
IF $Z \neq 0$ GOTO $L$          (2.1)
```

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

GOTO L

such as is available in most programming languages. To see that this is true we note that the first instruction of the segment guarantees that Z has a nonzero value. Thus the condition Z ≠ 0 is always true and hence the next instruction performed will be the instruction labeled L. Now GOTO L is not an instruction in our language $\mathscr{S}$, but since we will frequently have use for such an instruction, we can use it as an abbreviation for the program segment (2.1). Such an abbreviating pseudoinstruction will be called a ***macro*** and the program or program segment which it abbreviates will be called its ***macro expansion.***

The use of these terms is obviously motivated by similarities with the notion of a macro instruction occurring in many programming languages. At this point we will not discuss how to ensure that the variables local to the macro definition are distinct from the variables used in the main program. Instead, we will manually replace any such duplicate variable uses with unused variables. This will be illustrated in the "expanded" multiplication program in (e). In Section 5 this matter will be dealt with in a formal manner.

(c) Note that although the program of (b) does copy the value of X into Y, in the process the value of X is "destroyed" and the program terminates with X having the value 0. Of course, typically, programmers want to be able to copy the value of one variable into another without the original being "zeroed out." This is accomplished in the next program. (Note that we use our macro instruction GOT0 L several times to shorten the program. Of course, if challenged, we could produce a legal program of $\mathscr{S}$ by replacing each GOT0 L by a macro expansion. These macro expansions would have to use a local variable other than Z so as not to interfere with the value of Z in the main program.)

```
[A]    If X ≠ 0 GOTO B
       GOT0  C
[B]    X ← X − 1
       Y ← Y + 1
       Z ← Z + 1
       GOT0  A
[C]    IF Z ≠ 0 GOTO D
       GOT0  E
[D]    Z ← Z − 1
       X ← X + 1
       GOT0  C
```

In the first loop, this program copies the value of X into both Y and Z, while in the second loop, the value of X is restored. When the program terminates, both X and Y contain X's original value and $z = 0$.

We wish to use this program to justify the introduction of a macro which we will write

$$V \leftarrow V'$$

the execution of which will replace the contents of the variable V by the contents of the variable V' while leaving the contents of V' unaltered. Now, this program (c) functions correctly as a copying program only under our assumption that the variables Y and Z are initialized to the value 0. Thus, we can use the program as the basis of a macro expansion of $V \leftarrow V'$ only if we can arrange matters so as to be sure that the corresponding variables have the value 0 whenever the macro expansion is entered. To solve this problem we introduce the macro

$$V \leftarrow 0$$

which will have the effect of setting the contents of V equal to 0. The corresponding macro expansion is simply

```
[L]    $V \leftarrow V - 1$
       IF $V \neq 0$ GOTO $L$
```

where, of course, the label L is to be chosen to be different from any of the labels in the main program. We can now write the macro expansion of $V \leftarrow V'$ by letting the macro $V \leftarrow 0$ precede the program which results when X is replaced by V' and Y is replaced by V in program (c). The result is as follows:

```
       $V \leftarrow 0$
[A]    IF $V' \neq 0$ GOTO $B$
       GOTO C
[B]    $V' \leftarrow V' - 1$
       $V \leftarrow V + 1$
       $Z \leftarrow Z + 1$
       GOTO $A$
[C]    IF $Z \neq 0$ GOTO $D$
       GOTO $E$
[D]    $Z \leftarrow Z - 1$
       $V' \leftarrow V' + 1$
       GOTO C
```

With respect to this macro expansion the following should be noted:

1. It is unnecessary (although of course it would be harmless) to include a $Z \leftarrow 0$ macro at the beginning of the expansion because, as has already been remarked, program (c) terminates with $z = 0$.
2. When inserting the expansion in an actual program, the variable Z will have to be replaced by a local variable which does not occur in the main program.

3. Likewise the labels A, B, C, *D* will have to be replaced by labels which do not occur in the main program.
4. Finally, the label *E* in the macro expansion must be replaced by a label *L* such that the instruction which follows the macro in the main program (if there is one) begins [*L*].

(d) A program with two inputs that computes the function

$$f(x_1, x_2) = x_1 + x_2$$

is as follows:

```
        Y ← X_1
        Z ← X_2
[B]     IF Z ≠ 0 GOTO A
        GOT0  E
[A]     Z ← Z − 1
        Y ← Y + 1
        GOT0  B
```

Again, if challenged we would supply macro expansions for "$Y \leftarrow X_1$" and "$Z \leftarrow X_2$" as well as for the two unconditional branches. Note that Z is used to preserve the value of X_2.

(e) We now present a program that multiplies, i.e. that computes $f(x_1, x_2) = x_1 \cdot x_2$. Since multiplication can be regarded as repeated addition, we are led to the "program"

```
        Z_2 ← X_2
[B]     IF Z_2 ≠ 0 GOTO A
        GOT0  E
[A]     Z_2 ← Z_2 ∸ 1
        Z_1 ← X_1 + Y
        Y ← Z_1
        GOT0  B
```

Of course, the "instruction" $Z_1 \leftarrow X_1 + Y$ is not permitted in the language $\mathscr{S}$. What we have in mind is that since we already have an addition program, we can replace the macro $Z_1 \leftarrow X_1 + Y$ by a program for computing it, which we will call its macro expansion. At first glance, one might wonder why the pair of instructions

$$Z_1 \leftarrow X_1 + Y$$

$$Y \leftarrow Z_1$$

was used in this program rather than the single instruction

$$Y \leftarrow X_1 + Y$$

since we simply want to replace the current value of Y by the sum of its value and x_1. The sum program in (d) computes $Y = X_1 + X_2$. If we were to use that as a template, we would have to replace X_2 in the program by Y. Now if we tried to use Y also as the variable being assigned, the macro expansion would be as follows:

```
       Y ← X_1
       Z ← Y
[B]    IF Z ≠ 0 GOTO A
       GOTO E
[A]    Z ← Z − 1
       Y ← Y + 1
       GOTO B
```

What does this program actually compute? It should not be difficult to see that instead of computing $x_1 + y$ as desired, this program computes $2x_1$. Since X_1 is to be added over and over again, it is important that X_1 not be destroyed by the addition program. Here is the multiplication program, showing the macro expansion of $Z_1 \leftarrow X_1 + Y$:

```
        Z_2 ← X_2
[B]     IF Z_2 ≠ 0 GOTO A
        GOTO E
[A]     Z_2 ← Z_2 − 1
        Z_1 ← X_1              |
        Z_3 ← Y                |
[B_2]   IF Z_3 ≠ 0 GOTO A_2    |  Macro Expansion of
        GOTO E_2               |  Z_1 ← X_1 + Y
[A_2]   Z_3 ← Z_3 − 1          |
        Z_1 ← Z_1 + 1          |
        GOTO B_2               |
[E_2]   Y ← Z_1
        GOTO B
```

Note the following:

1. The local variable Z_1 in the addition program in (d) must be replaced by another local variable (we have used Z_3) because Z_1 (the other name for Z) is also used as a local variable in the multiplication program.

2. The labels A, B, E are used in the multiplication program and hence cannot be used in the macro expansion. We have used A,, B_2, E_2 instead.
3. The instruction GOT0 E_2 terminates the addition. Hence, it is necessary that the instruction immediately following the macro expansion be labeled E_2.

In the future we will often omit such details in connection with macro expansions. All that is important is that our infinite supply of variables and labels guarantees that the needed changes can always be made.

(f) For our final example, we take the program

```
        Y ← X_1
        Z ← X_2
[C]     IF Z ≠ 0 GOTO A
        GOT0 E
[A]     IF Y ≠ 0 GOTO B
        GOT0 A
[B]     Y ← Y − 1
        Z ← Z − 1
        GOT0 C
```

If we begin with $X_1 = 5$, $X_2 = 2$, the program first sets Y = 5 and $Z = 2$. Successively the program sets Y = 4, $Z = 1$ and Y = 3, $Z = 0$. Thus, the computation terminates with Y = 3 = 5 − 2. Clearly, if we begin with $X_1 = m$, $X_2 = n$, where $m \geq n$, the program will terminate with Y = $m - n$.

What happens if we begin with a value of X_1 less than the value of X_2, e.g., $X_1 = 2$, $X_2 = 5$? The program sets Y = 2 and $Z = 5$ and successively sets Y = 1, $Z = 4$ and Y = 0, $Z = 3$. At this point the computation enters the "loop":

```
[A]     IF Y ≠ 0 GOTO B
        GOT0 A
```

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*

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

(Partial functions are discussed in Chapter 1, Section 2.)

Exercises

1. Write a program in $\mathscr{S}$ (using macros freely) that computes the function $f(x) = 3x$.
2. Write a program in $\mathscr{S}$ that solves Exercise 1 using no macros.
3. Let $f(x) = 1$ if x is even; $f(x) = 0$ if x is odd. Write a program in $\mathscr{S}$ that computes f.
4. Let $f(x) = 1$ if x is even; $f(x)$ undefined if x is odd. Write a program in $\mathscr{S}$ that computes f.
5. Let $f(x_1, x_2) = 1$ if $x_1 = x_2$; $f(x_1, x_2) = 0$ if $x_1 \neq x_2$. Without using macros, write a program in $\mathscr{S}$ that computes f.
6. Let f(x) be the greatest number n such that $n^2 \leq x$. Write a program in $\mathscr{S}$ that computes f.
7. Let $\gcd(x_1, x_2)$ be the greatest common divisor of x_1 and x_2. Write a program in $\mathscr{S}$ that computes gcd.

3. Syntax

We are now ready to be mercilessly precise about the language $\mathscr{S}$. Some of the description recapitulates the preceding discussion.

The symbols

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

are called ***input variables,***

$$Z_1 \, Z_2 \, Z_3 \quad \cdots$$

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

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

are called ***labels*** of $\mathscr{S}$. (As already indicated, in practice the subscript 1 is often omitted.) 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.

Note that we have included among the statements of $\mathscr{S}$ the "dummy" commands $V \leftarrow V$. Since execution of these commands leaves all values unchanged, they have no effect on what a program computes. They are included for reasons that will not be made clear until much later. But their inclusion is certainly quite harmless.

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 program. It is useful to include the ***empty program*** of length 0, which of course contains no instructions.

As we have seen informally, in the course of a computation, the variables of a program assume different numerical values. This suggests the following definition:

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 of (b) from Section 2, which contains the variables X Y Z. The list

$$X = 4, \qquad Y = 3, \qquad Z = 3$$

is thus a state of $\mathscr{P}$. (The definition of ***state*** does not require that the state can actually be "attained" from some initial state.) The list

$$X_1 = 4, \qquad X_2 = 5, \qquad Y = 4, \qquad Z = 4$$

is also a state of $\mathscr{P}$. (Recall that X is another name for X_1 and note that the definition permits inclusion of equations involving variables not actually occurring in $\mathscr{P}$.) The list

$$X = 3, \qquad Z = 3$$

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

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

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. For example, the value of X at the state

$$X = 4, \qquad Y = 3, \qquad Z = 3$$

is 4.

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, a) where $1 \leq i \leq n + 1$, and σ is a state of $\mathscr{P}$. (Intuitively the number i indicates that it is the ith instruction which is about to be executed; $i = n + 1$ corresponds to a "stop" instruction.)

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

A snapshot (i, a) 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 $V \neq 0$ GOT0 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 = $n + 1$.

For an example, we return to the program of (b), Section 2. Let σ be the state

$$x = 4, \qquad Y = 0, \qquad Z = 0$$

and let us compute the successor of the snapshots (i, a) for various values of i.

For $i = 1$, the successor is (4, a) where σ is as above. For $i = 2$, the successor is (3, τ), where τ consists of the equations

$$x = 4, \qquad Y = 0, \qquad z = 1.$$

For $i = 7$, the successor is (8, a). This is a terminal snapshot.

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 $i = 1, 2, \ldots, k - 1$ 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. Thus, for example, the program

```
[A]    X + - - X - 1
       IF $X \neq 0$ GOTO A
[A]    $X \leftarrow X + 1$
```

is equivalent to the program

```
[A]    $X \leftarrow X - 1$
       IF $X \neq 0$ GOTO A
       $X \leftarrow X + 1$
```

Exercises

1. Let $\mathscr{P}$ be the program of(b), Section 2. Write out a computation of $\mathscr{P}$ beginning with the snapshot (1, a), where σ consists of the equations x = 2, Y = 0, Z = 0.
2. Give a program $\mathscr{P}$ such that for every computation $s_1, \ldots, s_k$ of $\mathscr{P}$, $k = 5$.
3. Give a program $\mathscr{P}$ such that for any $n \geq 0$ and every computation $s_1 = (1, \sigma), s_2, \ldots, s_k$ of $\mathscr{P}$ that has the equation X = n in σ, $k = 2n + 1$.

4. Computable Functions

We have been speaking of the function computed by a program $\mathscr{P}$. It is now time to make this notion precise.

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 be local. Although this has been and will continue to be our practice, it is convenient not to make it a formal requirement. According to the definitions we are going to present, any program $\mathscr{P}$ of the language $\mathscr{S}$ can be used to compute a function of one variable, a function of two variables, and, in general, for each $m \geq 1$, a function of m variables.

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

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

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 $s_1, s_2, \ldots, s_k$ of $\mathscr{P}$ beginning with the initial snapshot. Then we write $\psi_{\mathscr{P}}^{(m)}(r_1, r_2, \ldots, r_m)$ 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 $s_1, s_2, s_3, \ldots$ beginning with the initial snapshot where each s_{i+1} is the successor of s_i. In this case $\psi_{\mathscr{P}}^{(m)}(r_1, \ldots, r_m)$ is undefined.

Let us reexamine the examples in Section 2 from the point of view of this definition. We begin with the program of (b). For this program $\mathscr{P}$, we have

$$\psi_{\mathscr{P}}^{(1)}(x) = x$$

for all x. For this one example, we give a detailed treatment. The following list of snapshots is a computation of $\mathscr{P}$:

$$\begin{array}{l}
(1, \{X = r, Y = 0, Z = 0\}), \\
(4, \{X = r, Y = 0, Z = 0\}), \\
(5, \{X = r - 1, Y = 0, Z = 0\}), \\
(6, \{X = r - 1, Y = 1, Z = 0\}), \\
(7, \{X = r - 1, Y = 1, Z = 1\}), \\
(1, \{X = r - 1, Y = 1, Z = 1\}), \\
\vdots \\
(1, \{X = 0, Y = r, Z = r\}), \\
(2, \{X = 0, Y = r, Z = r\}), \\
(3, \{X = 0, Y = r, Z = r + 1\}), \\
(8, \{X = 0, Y = r, Z = r + 1\}).
\end{array}$$

We have included a copy of $\mathscr{P}$ showing line numbers:

```
[A]    IF $X \neq 0$ GOTO $B$          (1)
       $Z \leftarrow Z + 1$            (2)
       IF $Z \neq 0$ GOTO $E$          (3)
[B]    $X \leftarrow X - 1$            (4)
       $Y \leftarrow Y + 1$            (5)
       $Z \leftarrow Z + 1$            (6)
       IF $Z \neq 0$ GOTO $A$          (7)
```

For other examples of Section 2 we have

$$
\begin{array}{ll}
\text{(a)} & \psi^{(1)}(r) = \begin{cases} 1 & \text{if } r = 0 \\ r & \text{otherwise,} \end{cases} \\
\text{(b),(c)} & \psi^{(1)}(r) = r, \\
\text{(d)} & \psi^{(2)}(r_1, r_2) = r_1 + r_2, \\
\text{(e)} & \psi^{(2)}(r_1, r_2) = r_1 \cdot r_2, \\
\text{(f)} & \psi^{(2)}(r_1, r_2) = \begin{cases} r_1 - r_2 & \text{if } r_1 \geq r_2 \\ \uparrow & \text{if } r_1 < r_2. \end{cases}
\end{array}
$$

Of course in several cases the programs written in Section 2 are abbreviations, and we are assuming that the appropriate macro expansions have been provided.

As indicated, we are permitting each program to be used with any number of inputs. If the program has n input variables, but only $m < n$ are specified, then according to the definition, the remaining input variables are assigned the value 0 and the computation proceeds. If on the other hand, m values are specified where $m > n$ the extra input values are ignored. For example, referring again to the examples from Section 2, we have

$$
\begin{array}{ll}
\text{(c)} & \psi_{\mathscr{P}}^{(2)}(r_1, r_2) = r_1, \\
\text{(d)} & \psi_{\mathscr{P}}^{(1)}(r_1) = r_1 + 0 = r_1, \\
 & \psi_{\mathscr{P}}^{(3)}(r_1, r_2, r_3) = r_1 + r_2.
\end{array}
$$

For any program $\mathscr{P}$ and any positive integer m, the function $\psi_{\mathscr{P}}^{(m)}(x_1, \ldots, x_m)$ 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. That is, g is partially computable if there is a program $\mathscr{P}$ such that

$$g(r_1, \ldots, r_m) = \psi_{\mathscr{P}}^{(m)}(r_1, \ldots, r_m)$$

for all $r_1, \ldots, r_m$. Here this ‘equation must be understood to mean not only that both sides have the same value when they are defined, but also that when either side of the equation is undefined, the other is also.

As explained in Chapter 1, a given function g of m variables is called ***total*** if $g(r_1, \ldots, r_m)$ is defined for *all* $r_1, \ldots, r_m$. A function is said to be ***computable*** if 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. The*** reason for this terminology is largely historical and will be discussed later.

Our examples from Section 2 give us a short list of partially computable functions, namely: $x, x + y, x \cdot y$, and $x - y$. Of these, all except the last one are total and hence computable.

Computability theory (also called recursion theory) studies the class of partially computable functions. In order to justify the name, we need some evidence that for every function which one can claim to be "computable" on intuitive grounds, there really is a program of the language $\mathscr{S}$ which computes it. Such evidence will be developed as we go along.

We close this section with one final example of a program of $\mathscr{S}$:

```
[A]    X ← X + 1
       IF X ≠ 0 GOTO A
```

For this program $\mathscr{P}, \psi_{\mathscr{P}}^{(1)}(x)$ is undefined for all x. So, the nowhere defined function (see Chapter 1, Section 2) must be included in the class of partially computable functions.

Exercises

1. Let $\mathscr{P}$ be the program

```
       IF X ≠ 0 GOTO A
[A]    X ← X + 1
       IF X ≠ 0 GOTO A
[A]    Y ← Y + 1
```

What is $\psi_{\mathscr{P}}^{(1)}(x)$?

2. The same as Exercise 1 for the program

```
[B]    IF X ≠ 0 GOTO A
       Z ← Z + 1
       IF Z ≠ 0 GOTO B
[A]    X ← X
```

3. The same as Exercise 1 for the empty program.

4. Let $\mathscr{P}$ be the program

```
       Y ← X_1
[A]    IF X_2 = 0 GOTO E
       Y ← Y + 1
       Y ← Y + 1
       X_2 ← X_2 − 1
       GOTO A
```

What is $\psi_{\mathscr{P}}^{(1)}(r_1)$? $\psi_{\mathscr{P}}^{(2)}(r_1, r_2)$? $\psi_{\mathscr{P}}^{(3)}(r_1, r_2, r_3)$?

5. Show that for every partially computable function $f(x_1, \ldots, x_n)$, there is a number $m \geq 0$ such that f is computed by infinitely many programs of length m.

6. **(a)** For every number $k \geq 0$, let f_k be the constant function $f_k(x) = k$. Show that for every k, f_k is computable.

   (b) Let us call an $\mathscr{S}$ program a ***straightline program*** if it contains no (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOTO L. Show by induction on the length of programs that if the length of a straightline program $\mathscr{P}$ is k, then $\psi_{\mathscr{P}}^{(1)}(x) \leq k$ for all x.

   (c) Show that, if $\mathscr{P}$ is a straightline program that computes f_k, then the length of $\mathscr{P}$ is at least k.

   (d) Show that no straightline $\mathscr{S}$ program computes the function $f(x) = x + 1$. Conclude that the class of functions computable by straightline $\mathscr{S}$ programs is contained in but is not equal to the class of computable functions.

7. Let us call an $\mathscr{S}$ program $\mathscr{P}$ ***forward-branching*** if the following condition holds for each occurrence in $\mathscr{P}$ of a (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOTO L. If IF $V \neq 0$ GOTO L is the ith instruction of $\mathscr{P}$, then either L does not appear as the label of an instruction in $\mathscr{P}$, or else, if j is the least number such that L is the label of the jth instruction in $\mathscr{P}$, then $i < j$. Show that a function is computed by some forward-branching program if and only if it is computed by some straightline program (see Exercise 6).

8. Let us call a unary function ***f(x) partially n-computable*** if it is computed by some $\mathscr{S}$ program $\mathscr{P}$ such that $\mathscr{P}$ has no more than n instructions, every variable in $\mathscr{P}$ is among X, Y, $Z_1, \ldots, Z_n$, and every label in $\mathscr{P}$ is among $A_1, \ldots, A_n, E$.

   (a) Show that if a unary function is computed by a program with no more than n instructions, then it is partially n-computable.

   (b) Show that for every $n \geq 0$, there are only finitely many distinct partially n-computable unary functions.

   (c) Show that for every $n \geq 0$, there are only finitely many distinct unary functions computed by $\mathscr{S}$ programs of length no greater than n.

   (d) Conclude that for every $n \geq 0$, there is a partially computable unary function which is not computed by any $\mathscr{S}$ program of length less than n.

5. More about Macros

In Section 2 we gave some examples of computable functions (i.e., $x + y$, $x \cdot y$) giving rise to corresponding macros. Now we consider this process in general.

Let $f(x_1, \ldots, x_n)$ be some partially computable function computed by the program $\mathscr{P}$. We shall assume that the variables that occur in $\mathscr{P}$ are all included in the list Y, $X_1, \ldots, X_n, Z_1, \ldots, Z_k$ and that the labels that occur in $\mathscr{P}$ are all included in the list $E, A_1, \ldots, A_l$. We also assume that for each instruction of $\mathscr{P}$ of the form

$$\text{IF } V \neq 0 \text{ GOTO } A_i$$

there is in $\mathscr{P}$ an instruction labeled A_i. (In other words, E is the only "exit" label.) It is obvious that, if $\mathscr{P}$ does not originally meet these conditions, it will after minor changes in notation. We write

$$\mathscr{P} = \mathscr{P}(Y, X_1, \ldots, X_n, Z_1, \ldots, Z_k; E, A_1, \ldots, A_l)$$

in order that we can represent programs obtained from $\mathscr{P}$ by replacing the variables and labels by others. In particular, we will write

$$\mathscr{Q}_m = \mathscr{P}(Z_m, Z_{m+1}, \ldots, Z_{m+n}, Z_{m+n+1}, \ldots, Z_{m+n+k}; E_m, A_{m+1}, \ldots, A_{m+l})$$

for each given value of m. Now we want to be able to use macros like

$$W \leftarrow f(V_1, \ldots, V_n)$$

in our programs, where $V_1, \ldots, V_n, W$ can be any variables whatever. (In particular, W might be one of $V_1, \ldots, V_n$.) We will take such a macro to be an abbreviation of the following expansion:

```
        Z_m ← 0
        Z_{m+1} ← V_1
        Z_{m+2} ← V_2
        .

        Z_{m+n} ← V_n
        Z_{m+n+1} ← 0
        Z_{m+n+2} ← 0

        .
        Z_{m+n+k} ← 0
        Q_m
[E_m]   W ← Z_m
```

Here it is understood that the number m is chosen so large that none of the variables or labels used in $\mathscr{Q}_m$ occur in the main program of which the expansion is a part. Notice that the expansion sets the variables corresponding to the output and local variables of $\mathscr{P}$ equal to 0 and those corresponding to $X_1, \ldots, X_n$ equal to the values of $V_1, \ldots, V_n$, respectively. Setting the variables equal to 0 is necessary (even though they are all local variables automatically initialized to 0) because the expansion may be part of a loop in the main program; in this case, at the second and subsequent times through the loop the local variables will have whatever values they acquired the previous time around, and so will need to be reset. Note that when $\mathscr{Q}_m$ terminates, the value of Z_m is $f(V_1, \ldots, V_n)$, so that W finally does get the value $f(V_1, \ldots, V_n)$.

If $f(V_1, \ldots, V_n)$ is undefined, the program $\mathscr{Q}_m$ will never terminate. Thus if f is not total, and the macro

$$W \leftarrow f(V_1, \ldots, V_n)$$

is encountered in a program where $V_1, \ldots, V_n$ have values for which f is not defined, the main program will never terminate.

Here is an example:

$$Z \leftarrow X_1 - X_2$$
$$Y \leftarrow Z + X_3$$

This program computes the function $f(x_1, x_2, x_3)$, where

$$f(x_1, x_2, x_3) = \begin{cases} (x_1 - x_2) + x_3 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2. \end{cases}$$

In particular, $f(2,5,6)$ is undefined, although $(2 - 5) + 6 = 3$ is positive. The computation never gets past the attempt to compute $2 - 5$.

So far we have augmented our language $\mathscr{S}$ to permit the use of macros which allow assignment statements of the form

$$W \leftarrow f(V_1, \ldots, V_n),$$

where f is any partially computable function. Nonetheless there is available only one highly restrictive conditional branch statement, namely,

$$\text{IF } V \neq 0 \text{ GOTO } L$$

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

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

where $P(x_1, \ldots, x_n)$ is a computable predicate. Here we are making use of the convention, introduced in Chapter 1, that

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

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.

Let $P(x_1, \ldots, x_n)$ be any computable predicate. Then the appropriate macro expansion of

$$\text{IF } P(V_1, \ldots, V_n) \text{ GOT0 } L$$

is simply

```
$Z \leftarrow P(V_1, \ldots, V_n)$
IF $Z \neq 0$ GOTO $L$
```

Note that P is a computable function and hence we have already shown how to expand the first instruction. The second instruction, being one of the basic instructions in the language $\mathcal{S}$, needs no further expansion.

A simple example of this general kind of conditional branch statement which we will use frequently is

$$\text{IF } V = 0 \text{ GOTO } L$$

To see that this is legitimate we need only check that the-predicate $P(x)$, defined by $P(x)$ = TRUE if $x = 0$ and $P(x)$ = FALSE otherwise, is computable. Since TRUE = 1 and FALSE = 0, the following program does the job:

```
IF $X \neq 0$ GOTO $E$
$Y \leftarrow Y + 1$
```

The use of macros has the effect of enabling us to write much shorter programs than would be possible restricting ourselves to instructions of the original language $\mathcal{S}$. The original "assignment" statements $V \leftarrow V + 1$, $V \leftarrow V - 1$ are now augmented by general assignment statements of the form $W \leftarrow f(V_1, \ldots, V_n)$ for any partially computable function f. Also, the original conditional branch statements IF $V \neq 0$ GOT0 L are now augmented by general conditional branch statements of the form IF $P(V_1, \ldots, V_n)$ GOT0 L for any computable predicate P. The fact that any function which can be computed using these general instructions could already have been computed by a program of our original language $\mathcal{S}$ (since the general instructions are merely abbreviations of programs of $\mathcal{S}$) is powerful evidence of the generality of our notion of computability.

Our next task will be to develop techniques that will make it easy to see that various particular functions are computable.

Exercises

1. (a) Use the process described in this section to expand the program in example (d) of Section 2.

   (b) What is the length of the $\mathscr{S}$ program expanded from example (e) by this process?

2. Replace the instructions

$$Z_1 \leftarrow X_1 + Y$$
$$Y \leftarrow Z_1$$

   in example (e) of Section 2 with the instruction $Y \leftarrow X_1 + Y$, and expand the result by the process described in this section. If $\mathscr{P}$ is the resulting $\mathscr{S}$ program, what is $\psi_{\mathscr{P}}^{(2)}(r_1, r_2)$?

3. Let $f(x), g(x)$ be computable functions and let $h(x) = f(g(x))$. Show that h is computable.

4. Show by constructing a program that the predicate $x_1 \leq x_2$ is computable.

5. Let $P(x)$ be a computable predicate. Show that the function f defined by

$$f(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{if } P(x_1 + x_2) \\ \uparrow & \text{otherwise} \end{cases}$$

   is partially computable.

6. Let $P(x)$ be a computable predicate. Show that

$$EX_P(r) = \begin{cases} 1 & \text{if there are at least } r \text{ numbers } n \text{ such that } P(n) = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

   is partially computable.

7. Let π be a computable permutation (i.e., one-one, onto function) of N, and let π^{-1} be the inverse of π, i.e.,

$$\pi^{-1}(y) = x \qquad \text{if and only if} \qquad \pi(x) = y.$$

   Show that π^{-1} is computable.

8. Let $f(x)$ be a partially computable but not total function, let M be a finite set of numbers such that $f(m)\uparrow$ for all $m \in M$, and let g(x) be

an arbitrary partially computable function. Show that

$$h(x) = \begin{cases} g(x) & \text{if } x \in M \\ f(x) & \text{otherwise} \end{cases}$$

is partially computable.

9. Let $\mathscr{S}^+$ be a programming language that extends $\mathscr{S}$ by permitting instructions of the form $V \leftarrow k$, for any $k \geq 0$. *These* instructions have the obvious effect of setting the value of V to k. Show that a function is partially computable by some $\mathscr{S}^+$ program if and only if it is partially computable.

10. Let $\mathscr{S}'$ be a programming language defined like $\mathscr{S}$ except that its (labeled and unlabeled) instructions are of the three types

$$\begin{array}{l} V \leftarrow V' \\ V \leftarrow V + 1 \\ \text{If } V \neq V' \text{ GOTO } L \end{array}$$

These instructions are given the obvious meaning. Show that a function is partially computable in $\mathscr{S}'$ if and only if it is partially computable.

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. 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.***

Of course, the functions $f, g_1, \ldots, g_k$ need not be total. $h(x_1, \ldots, x_n)$ will be defined when all of $z_1 = g_1(x_1, \ldots, x_n), \ldots, z_k = g_k(x_1, \ldots, x_n)$ are defined and also $f(z_1, \ldots, z_k)$ is defined.

Using macros it is very easy to prove

Theorem 1.1. 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.

Proof. The following program obviously computes ***h:***

$$
\begin{aligned}
&Z_1 \leftarrow g_1(X_1,\dots,X_n)\\
&\\
&Z_k \leftarrow g_k(X_1,\dots,X_n)\\
&Y \leftarrow f(Z_1,\dots,Z_k)
\end{aligned}
$$

If $f, g_1, \dots, g_k$ are not only partially computable but are also total, then so is ***h.*** ■

By Section 4 of Chapter 2, we know that x, $x + y$, $x \cdot y$, and $x - y$ are partially computable. So by Theorem 1.1 we see that $2x = x + x$ and $4x^2 = (2x) \cdot (2x)$ are computable. So are $4x^2 + 2x$ and $4x^2 - 2x$. Note that $4x^2 - 2x$ is total, although it is obtained from the nontotal function $x - y$ by composition with $4x^2$ and 2x.

2. Recursion

Suppose ***k*** is some fixed number and

$$
\begin{aligned}
h(0) &= k,\\
h(t+1) &= g(t, h(t)),
\end{aligned}
\tag{2.1}
$$

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.***[1]

Theorem 2.1. Let h be obtained from g as in (2.1), and let g be computable. Then h is also computable.

Proof. We first note that the constant function $f(x) = k$ is computable; in fact, it is computed by the program

$$
\left.
\begin{aligned}
&Y \leftarrow Y + 1\\
&Y \leftarrow Y + 1\\
&\vdots\\
&Y \leftarrow Y + 1
\end{aligned}
\right\} \ k \text{ lines}
$$

[1] Primitive recursion, characterized by Equations (2.1) and (2.2), is just one specialized form of recursion, but it is the only one we will be concerned with in this chapter, so we will refer to it simply as recursion. We will consider more general forms of recursion in Part 5.

Hence we have available the macro $Y \leftarrow k$. The following is a program that computes $h(x)$:

```
        $Y \leftarrow k$
[A]     IF $X = 0$ GOTO $E$
        $Y \leftarrow g(Z, Y)$
        $Z \leftarrow Z + 1$
        $X \leftarrow X - 1$
        GOTO $A$
```

To see that this program does what it is supposed to do, note that, if Y has the value $h(z)$ before executing the instruction labeled A, then it has the value $g(z, h(z)) = h(z + 1)$ after executing the instruction $Y \leftarrow g(Z, Y)$. Since Y is initialized to $k = h(0)$, Y successively takes on the values $h(0), h(1), \ldots, h(x)$ and then terminates. ■

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

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n, t + 1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n). \end{aligned} \tag{2.2}$$

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). The recursion (2.2) is just like (2.1) except that parameters $x_1, \ldots, x_n$ are involved. Again we have

Theorem 2.2. Let h be obtained from f and g as in (2.2) and let f, g be computable. Then h is also computable.

Proof. The proof is almost the same as for Theorem 2.1. The following program computes $h(x_1, \ldots, x_n, x_{n+1})$:

```
        $Y \leftarrow f(X_1, \ldots, X_n)$
[A]     IF $X_{n+1} = 0$ GOTO $E$
        $Y \leftarrow g(Z, Y, X_1, \ldots, X_n)$
        $Z \leftarrow Z + 1$
        $X_{n+1} \leftarrow X_{n+1} - 1$
        GOTO $A$
```

■

3. PRC Classes

So far we have considered the operations of composition and recursion. Now we need some functions on which to get started. These will be

$$s(x) = x + 1,$$
$$n(x) = 0,$$

and the ***projection functions***

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

[For example, $u_3^4(x_1, x_2, x_3, x_4) = x_3$.] The functions s, n, and u_i^n are called the ***initial functions.***

Definition. A class of total functions $\mathscr{C}$ is called a ***PRC***[2] ***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 3.1. The class of computable functions is a PRC class.

Proof. By Theorems 1.1, 2.1, and 2.2, we need only verify that the initial functions are computable.

Now this is obvious; $s(x) = x + 1$ is computed by

$$Y \leftarrow X + 1$$

n(x) is computed by the empty program, and $u_i^n(x_1, \ldots, x_n)$ is computed by the program

$$\mathrm{Y} \leftarrow X_i$$ ■

Definition. 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

[2] This is an abbreviation for "primitive recursively closed."

Corollary 3.2. The class of primitive recursive functions is a PRC class.

Actually we can say more:

Theorem 3.3. A function is primitive recursive if and only if it belongs to every PRC class.

Proof. If a function belongs to every PRC class, then, in particular, by Corollary 3.2, it belongs to the class of primitive recursive functions.

Conversely let a function f be a primitive recursive function and let $\mathscr{C}$ be some PRC class. We want to show that f belongs to $\mathscr{C}$. Since f is a primitive recursive function, there is a list $f_1, f_2, \ldots, f_n$ of functions such that $f_n = f$ and each f_i in the list is either an initial function or can be obtained from preceding functions in the list by composition or recursion. Now the initial functions certainly belong to the PRC class $\mathscr{C}$. Moreover the result of applying composition or recursion to functions in $\mathscr{C}$ is again a function belonging to $\mathscr{C}$. Hence each function in the list $f_1, \ldots, f_n$ belongs to $\mathscr{C}$. Since $f_n = f$, f belongs to $\mathscr{C}$. ■

Corollary 3.4. Every primitive recursive function is computable.

Proof. By the theorem just proved, every primitive recursive function belongs to the PRC class of computable functions. ■

In Chapter 4 we shall show how to obtain a computable function that is not primitive recursive. Hence it will follow that the set of primitive recursive functions is a proper subset of the set of computable functions.

Exercises

1. Let $\mathscr{C}$ be a PRC class, and let g_1, g_2, g_3, g_4 belong to $\mathscr{C}$. Show that if

$$h_1(x, y, z) = g_1(z, y, x),$$

$$h_2(x) = g_2(x, x, x), \text{ and}$$

$$h_3(w, x, y, z) = h_1(g_3(w, y), z, g_4(2, g_4(y, z))),$$

then h_2, h_2, h_3 also belong to $\mathscr{C}$.

2. **Show** that the class of all total functions is a PRC class.
3. Let $n > 0$ be some given number, and let $\mathscr{C}$ be a class of total functions of no more than n variables. Show that $\mathscr{C}$ is not a PRC class.

4. Let $\mathscr{C}$ be a PRC class, let h belong to $\mathscr{C}$, and let

$$f(x) = h(g(x)) \text{ and}$$
$$g(x) = h(f(x)).$$

Show that f belongs to $\mathscr{C}$ if and only if g belongs to $\mathscr{C}$.

5. Prove Corollary 3.4 directly from Theorems 1.1, 2.1, 2.2, and the proof of Theorem 3.1.

4. Some Primitive Recursive Functions

We proceed to make a short list of primitive recursive functions. Being primitive recursive, they are also computable.

1. $x + y$

To see that this is primitive recursive, we have to show how to obtain this function from the initial functions using only the operations of composition and recursion.

If we write $f(x, y) = x + y$, we have the recursion equations

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

We can rewrite these equations as

$$f(x,0) = u_1^1(x),$$
$$f(x,y + 1) = g(y, f(x,y), x),$$

where $g(x_1, x_2, x_3) = s(u_2^3(x_1, x_2, x_3))$. The functions $u_1^1(x), u_2^3(x_1, x_2, x_3)$, and $s(x)$ are primitive recursive functions; in fact they are initial functions. Also, $g(x_1, x_2, x_3)$ is a primitive recursive function, since it is obtained by composition of primitive recursive functions. Thus, the preceding is a valid application of the operation of recursion to primitive recursive functions. Hence $f(x, y) = x + y$ is primitive recursive.

Of course we already knew that $x + y$ was a computable function. So we have only obtained the additional information that it is in fact primitive recursive.

2. $x \cdot y$

The recursion equations for $h(x,y) = x \cdot y$ are

$$h(x,0) = 0,$$
$$h(x,y + 1) = h(x,y) + x.$$

This can be rewritten

$$h(x,0) = n(x)$$
$$h(x, y+1) = g(y, h(x,y), x).$$

Here, n(x) is the zero function,

$$g(x_1, x_2, x_3) = f(u_2^3(x_1, x_2, x_3), u_3^3(x_1, x_2, x_3)),$$

$f(x_1, x_2)$ is $x_1 + x_2$, and $u_2^3(x_1, x_2, x_3)$, $u_3^3(x_1, x_2, x_3)$ are projection functions. Notice that the functions n(x), $u_2^3(x_1, x_2, x_3)$, and $u_3^3(x_1, x_2, x_3)$ are all primitive recursive functions, since they are all initial functions. We have just shown that $f(x_1, x_2) = x_1 + x_2$ is primitive recursive, so $g(x_1, x_2, x_3)$ is a primitive recursive function since it is obtained from primitive recursive functions by composition. Finally, we conclude that

$$h(x,y) = x \cdot y$$

is primitive recursive.

3. $x!$

The recursion equations are

$$0! = 1,$$
$$(x+1)! = x! \cdot s(x).$$

More precisely, x! = ***h(x),*** where

$$h(0) = 1,$$
$$h(t+1) = g(t, h(t)),$$

and

$$g(x_1, x_2) = s(x_1) \cdot x_2.$$

Finally, g is primitive recursive because

$$g(x_1, x_2) = s(u_1^2(x_1, x_2)) \cdot u_2^2(x_1, x_2)$$

and multiplication is already known to be primitive recursive.

In the examples that follow, we leave it to the reader to check that the recursion equations can be put in the precise form called for by the definition of the operation of recursion.

4. x^y

The recursion equations are

$$x^0 = 1,$$
$$x^{y+1} = x^y \cdot x.$$

Note that these equations assign the value 1 to the "indeterminate" 0^0.

5. *p(x)*

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

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

It corresponds to the instruction in our programming language X ← X − 1.
The recursion equations for ***p(x)*** are simply

$$p(0) = 0,$$
$$p(t + 1) = t.$$

Hence, ***p(x)*** is primitive recursive.

6. $x \dot{-} y$

The function $x \dot{-} y$ is defined as follows:

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y. \end{cases}$$

This function should not be confused with the function $x - y$, which is undefined if $x < y$. In particular, $x \dot{-} y$ is total, while $x - y$ is not.

We show that $x \dot{-} y$ is primitive recursive by displaying the recursion equations:

$$x \dot{-} 0 = x,$$
$$x \dot{-} (t + 1) = p(x \dot{-} t).$$

7. $|x - y|$

The function $|x - y|$ is defined as the absolute value of the difference between x and y. It can be expressed simply as

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

and thus is primitive recursive.

8. $\alpha(x)$

The function $\alpha(x)$ is defined as

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

$\alpha(x)$ is primitive recursive since

$$\alpha(x) = 1 \mathbin{\dot-} x.$$

Or we can simply write the recursion equations:

$$\alpha(0) = 1,$$
$$\alpha(t + 1) = 0.$$

Exercises

1. Give a detailed argument that $x^y, p(x)$, and $x \mathbin{\dot-} y$ are primitive recursive.

2. Show that for each k, the function $f(x) = k$ is primitive recursive.

3. Prove that if $f(x)$ and $g(x)$ are primitive recursive functions, so is $f(x) + g(x)$.

4. Without using $x + y$ as a macro, apply the constructions in the proofs of Theorems 1.1, 2.2, and 3.1 to give an $\mathscr{S}$ program that computes $x \cdot y$.

5. For any unary function $f(x)$, the nth ***iteration*** of f, written f^n, is

$$f^n(x) = f(\cdots f(x) \cdots),$$

where f is composed with itself n times on the right side of the equation. (Note that $f^0(x) = x$.) Let $\iota_f(n,x) = f^n(x)$. Show that if f is primitive recursive, then ι_f is also primitive recursive.

6.* (a) Let $E(x) = 0$ if x is even, $E(x) = 1$ if x is odd. Show that $E(x)$ is primitive recursive.

(b) Let $H(x) = x/2$ if x is even, $(x-1)/2$ if x is odd. Show that $H(x)$ is primitive recursive.

7.* Let $f(0) = 0$, $f(1) = 1$, $f(2) = 2^2$, $f(3) = 3^{3^3} = 3^{27}$, etc. In general, $f(n)$ is written as a stack n high, of n's as exponents. Show that f is primitive recursive.

8.* Let k be some fixed number, let f be a function such that $f(x+1) < x+1$ for all x, and let

$$h(0) = k$$
$$h(t+1) = g(h(f(t+1))).$$

Show that if f and g belong to some PRC class $\mathscr{C}$, then so does h. *[Hint:* Define $f'(x) = \min_{t \le x} f^t(x) = 0$. See Exercise 5 for the definition of $f^t(x)$.]

9.* Let $g(x)$ be a primitive recursive function and let $f(0,x) = g(x)$, $f(n+1,x) = f(n, f(n,x))$. Prove that $f(n,x)$ is primitive recursive.

10.* Let COMP be the class of functions obtained from the initial functions by a finite sequence of compositions.

(a) Show that for every function $f(x_1, \ldots, x_n)$ in COMP, either $f(x_1, \ldots, x_n) = k$ for some constant k, or $f(x_1, \ldots, x_n) = x_i + k$ for some $1 \le i \le n$ and some constant k.

(b) An n-ary function f is ***monotone*** if for all n-tuples $(x_1, \ldots, x_n)$, $(y_1, \ldots, y_n)$ such that $x_i \le y_i$, $1 \le i \le n$, $f(x_1, \ldots, x_n) \le f(y_1, \ldots, y_n)$. Show that every function in COMP is monotone.

(c) Show that COMP is a proper subset of the class of primitive recursive functions.

(d) Show that the class of functions computed by straightline $\mathscr{S}$ programs is a proper subset of COMP. [See Exercise 4.6 in Chapter 2 for the definition of straightline programs.]

11.* Let $\mathscr{P}_1$ be the class of all functions obtained from the initial functions by any finite number of compositions and no more than one recursion (in any order).

(a) Let $f(x_1, \ldots, x_n)$ belong to COMP. [See Exercise 10 for the definition of COMP.] Show that there is a $k > 0$ such that $f(x_1, \ldots, x_n) \le \max\{x_1, \ldots, x_n\} + k$.

(b) Let

$$h(0) = c$$
$$h(t+1) = g(t, h(t)),$$

where c is some given number and g belongs to COMP. Show that there is a $k > 0$ such that $h(t) \le tk + c$.

(c) Let

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

where f, g belong to COMP. Show that there are $k, l > 0$ such that $h(x_1, \ldots, x_n, t) \leq tk + \max\{x_1, \ldots, x_n\} + l$.

(d) Let $f(x_1, \ldots, x_n)$ belong to $\mathscr{P}_1$. Show that there are $k, l > 0$ such that $f(x_1, \ldots, x_n) \leq \max\{x_1, \ldots, x_n\} \cdot k + l$.

(e) Show that $\mathscr{P}_1$ is a proper subset of the class of primitive recursive functions.

5. Primitive Recursive Predicates

We recall from Chapter 1, Section 4, that predicates or Boolean-valued functions are simply total functions whose values are 0 or 1. (We have identified 1 with TRUE and 0 with FALSE.) Thus we can speak without further ado of primitive recursive predicates.

We continue our list of primitive recursive functions, including some that are predicates.

9. $x = y$

The predicate $x = y$ is defined as 1 if the values of x and y are the same and 0 otherwise. Thus we wish to show that the function

$$d(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

is primitive recursive. This follows immediately from the equation

$$d(x, y) = \alpha(|x - y|).$$

10. $x \leq y$

This predicate is simply the primitive recursive function $\alpha(x \dot{-} y)$.

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 $P \,\&\, Q$.[3]

Proof. Since $\sim P = \alpha(P)$, it follows that $\sim P$ belongs to $\mathscr{C}$. (α was defined in Section 4, item 8.)

[3] See Chapter 1, Section 4.

Also, we have

$$P \mathbin{\&} Q = P \cdot Q,$$

so that $P \mathbin{\&} Q$ belongs to $\mathscr{C}$.

Finally, the De Morgan law

$$P \vee Q \Leftrightarrow \sim(\sim P \mathbin{\&} \sim Q)$$

shows, using what we have already done, that $P \vee Q$ belongs to $\mathscr{C}$. ■

A result like Theorem 5.1 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 $P \mathbin{\&} Q$.

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 $P \mathbin{\&} Q$.

As a simple example we have

11. $x < y$

We can write

$$x < y \Leftrightarrow x \leq y \mathbin{\&} \sim(x = y),$$

or more simply

$$x < y \Leftrightarrow \sim(y \leq x).$$

Theorem 5.4 (Definition by Cases). Let $\mathscr{C}$ be a PRC class. Let the functions g, h and the predicate P belong to $\mathscr{C}$. Let

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

Then f belongs to $\mathscr{C}$.

This will be recognized as a version of the familiar "if.. . then.. . , else ..." statement.

Proof. The result is obvious because

$$f(x_1, \ldots, x_n)$$
$$= g(x_1, \ldots, x_n) \cdot P(x_1, \ldots, x_n) + h(x_1, \ldots, x_n) \cdot \alpha(P(x_1, \ldots, x_n)).$$

■

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

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

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

Proof. We argue by induction on m. ***The*** case for $m = 1$ is given by Theorem 5.4, so let

$$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n) & \text{if } P_1(x_1, \ldots, x_n) \\ \quad\vdots & \quad\vdots \\ g_{m+1}(x_1, \ldots, x_n) & \text{if } P_{m+1}(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n) & \text{otherwise,} \end{cases}$$

and let

$$h'(x_1, \ldots, x_n) = \begin{cases} g_{m+1}(x_1, \ldots, x_n) & \text{if } P_{m+1}(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n) & \text{otherwise.} \end{cases}$$

Then

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

and h' belongs to $\mathscr{C}$ by Theorem 5.4, so f belongs to $\mathscr{C}$ by the induction hypothesis. ■

Exercise

1. Let us call a predicate ***trivial*** if it is always TRUE or always FALSE. Show that no nontrivial predicates belong to COMP (see Exercise 4.10 for the definition of COMP.)

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 $\mathscr{C}$.

We proceed with the correct proof.

Proof. We note the recursion equations

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

and recall that since + is primitive recursive, it belongs to $\mathscr{C}$.

Similarly,

$$h(0, x_1, \ldots, x_n) = f(0, x_1, \ldots, x_n),$$
$$h(t+1, x_1, \ldots, x_n) = h(t, x_1, \ldots, x_n) \cdot f(t+1, x_1, \ldots, x_n). \quad \blacksquare$$

Sometimes we will want to begin the summation (or product) at 1 instead of 0. That is, we will want to consider

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

or

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

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 $h(t+1, x_1, \ldots, x_n)$ as in the preceding proof. 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[4]

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

proof. We need only observe that

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) \Leftrightarrow \left[\prod_{t=0}^{y} P(t, x_1, \ldots, x_n)\right] = 1$$

and

$$(\exists t)_{\leq y} P(t, x_1, \ldots, x_n) \Leftrightarrow \left[\sum_{t=0}^{y} P(t, x_1, \ldots, x_n)\right] \neq 0. \qquad \blacksquare$$

Actually for the universal quantifier it would even have been correct to write the equation

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) = \prod_{t=0}^{y} P(t, x_1, \ldots, x_n).$$

[4] See Chapter 1, Section 5.

Sometimes in applying Theorem 6.3 we want to use the quantifier

$$(\forall t)_{<y} \qquad \text{or} \qquad (\exists t)_{<y}.$$

That the theorem is still valid is clear from the relations

$$(\exists t)_{<y} P(t, x_1, \ldots, x_n) \Leftrightarrow (\exists t)_{\leq y}[t \neq \mathrm{y} \;\&\; P(t, x_1, \ldots, x_n)],$$

$$(\forall t)_{<y} P(t, x_1, \ldots, x_n) \Leftrightarrow (\forall t)_{\leq y}[t = \mathrm{y} \vee P(t, x_1, \ldots, x_n)].$$

We continue our list of examples.

12. $y \mid x$

This is the predicate "y is a divisor of x." For example,

$$3 \mid 12 \qquad \text{is true}$$

while

$$3 \mid 13 \qquad \text{is false.}$$

The predicate is primitive recursive since

$$y \mid x \Leftrightarrow (\exists t)_{\leq x}(y \cdot t = x).$$

13. Prime(x)

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

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

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

Exercises

1. Let $f(x) = 2x$ if x is a perfect square; $f(x) = 2x + 1$ otherwise. Show that f is primitive recursive.
2. Let $\sigma(x)$ be the sum of the divisors of x if $x \neq 0$; $\sigma(0) = 0$ [e.g., a(6) = 1 + 2 + 3 + 6 = 12]. Show that a(x) is primitive recursive.
3. Let $\pi(x)$ be the number of primes that are $\leq x$. Show that $\pi(x)$ is primitive recursive.
4. Let SQSM(x) be true if x is the sum of two perfect squares; false otherwise. Show that SQSM(x) is primitive recursive.

5. Let $\mathscr{C}$ be a PRC class, let $P(t, x_1, \ldots, x_n)$ be a predicate in $\mathscr{C}$, and let

$$g(y, z, x_1, \ldots, x_n) = (\forall t)_{y \le t \le z} P(t, x_1, \ldots, x_n) \text{ and}$$
$$h(y, z, x_1, \ldots, x_n) = (\exists t)_{y \le t \le z} P(t, x_1, \ldots, x_n),$$

(where $(\forall t)_{y \le t \le z} P(t, x_1, \ldots, x_n)$ and $(\exists t)_{y \le t \le z} P(t, x_1, \ldots, x_n)$ mean that $P(t, x_1, \ldots, x_n)$ is true for all t (respectively, for some t) from y to z). Show that g, h also belong to $\mathscr{C}$.

6. Let RP (x, y) be true if x and y are relatively prime (i.e., their greatest common divisor is 1). Show that RP(x, y) is primitive recursive.

7. Give a sequence of compositions and recursions that shows explicitly that Prime(x) is primitive recursive.

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 $\mathscr{C}$. (Recall that the primitive recursive function α was defined in Section 4.) Let us analyze this function g. Suppose for definiteness that for some value of $t_0 \le y$,

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

but

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

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

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

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

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

Thus, $\min_{t \le y} P(t, x_1, \ldots, x_n)$ ***is the least value of*** $t \le y$ ***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 $f(y, x_1, \ldots, x_n) = \min_{t \le y} P(t, x_1, \ldots, x_n)$, then f also belongs to $\mathscr{C}$.

The operation "$\min_{t \le y}$" is called ***bounded minimalization.***

Continuing our list:

14. $\lfloor x/y \rfloor$

$\lfloor x/y \rfloor$ is the "integer part" of the quotient x/y. For example, $\lfloor 7/2 \rfloor = 3$ and $\lfloor 2/3 \rfloor = 0$. The equation

$$\lfloor x/y \rfloor = \min_{t \le x} [(t+1) \cdot y > x]$$

shows that $\lfloor x/y \rfloor$ is primitive recursive. Note that according to this equation, we are taking $\lfloor x/0 \rfloor = 0$.

15. $R(x, y)$

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

$$\frac{x}{y} = \lfloor x/y \rfloor + \frac{R(x,y)}{y},$$

we can write

$$R(x, y) = x \mathbin{\dot{-}} (y \cdot \lfloor x/y \rfloor),$$

so that $R(x, y)$ is primitive recursive. [Note that $R(x, 0) = x$.]

16. p_n

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 $p_0 = 0$. Thus, $p_0 = 0$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc.

Consider the recursion equations

$$p_0 = 0,$$

$$p_{n+1} = \min_{t \le p_n! + 1} [\text{Prime}(t) \mathbin{\&} t > p_n].$$

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

$$p_{n+1} \le (p_n)! + 1. \tag{7.1}$$

To do so note that for $0 < i \leq n$ *we* have

$$\frac{(p_n)!+1}{p_i} = K + \frac{1}{p_i},$$

where K is an integer. Hence $(p_n)! + 1$ is not divisible by any of the primes $p_1, p_2, \ldots, p_n$. ***So,*** either $(p_n)! + 1$ is itself a prime or it is divisible by a prime $> p_n$. In either case there is a prime q such that $p_n < q \leq (p_n)! + 1$, which gives the inequality (7.1). (This argument is just Euclid's proof that there are infinitely many primes.)

Before we can confidently assert that p_n is a primitive recursive function, we need to justify the interleaving of the recursion equations with bounded minimalization. To do so, we first define the primitive recursive function

$$h(y,z) = \min_{t \leq z} [\text{Prime}(t) \; \& \; t > y].$$

Then we set

$$k(x) = h(x, x!+1),$$

another primitive recursive function. Finally, our recursion equations reduce to

$$p_0 = 0,$$

$$p_{n+1} = k(p_n),$$

so that we can conclude finally that p_n is a primitive recursive function.

It is worth noting that by using our various theorems (and appropriate macro expansions) we could now obtain explicitly a program of $\mathscr{S}$ which actually computes p_n. Of course the program obtained in this way would be extremely inefficient.

Now we want to discuss minimalization when there is no bound. 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.*** (Note carefully the difference with bounded minimalization.) Thus unbounded minimalization of a predicate can easily produce a function which is not total. For example,

$$x - y = \min_z [y + z = x]$$

is undefined for $x < y$. Now, as we shall see later, 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.

Proof. ***The*** following program obviously computes g:

```
[A]     IF $P(X_1, \ldots, X_n, Y)$ GOTO $E$
        $Y \leftarrow Y + 1$
        GOTO $A$
```

■

Exercises

1. Let $h(x)$ be the integer n such that $n \le \sqrt{2}x < n + 1$. Show that $h(x)$ is primitive recursive.
2. Do the same when $h(x)$ is the integer n such that
$$n \le (1 + \sqrt{2})x < n + 1.$$
3. p is called a ***larger twin prime*** if p and $p - 2$ are both primes. (5, 7, 13, 19 are larger twin primes.) Let $T(0) = 0$, $T(n)$ = the nth larger twin prime. It is widely believed, but has not been proved, that there are infinitely many larger twin primes. Assuming that this is true prove that $T(n)$ is computable.
4. Let $u(n)$ be the nth number in order of size which is the sum of two squares. Show that $u(n)$ is primitive recursive.
5. Let $R(x, t)$ be a primitive recursive predicate. Let
$$g(x, y) = \max_{t \le y} R(x, t),$$
i.e., $g(x, y)$ is the largest value of $t \le y$ for which $R(x, t)$ is true; if there is none, $g(x, y) = 0$. Prove that $g(x, y)$ is primitive recursive.
6. Let $\gcd(x, y)$ be the greatest common divisor of x and y. Show that $\gcd(x, y)$ is primitive recursive.
7. Let $\operatorname{lcm}(x, y)$ be the least common multiple of x and y. Show that $\operatorname{lcm}(x, y)$ is primitive recursive.

8. Give a computable predicate $P(x_1, \ldots, x_n, y)$ such that the function $\min_y P(x_1, \ldots, x_n, y)$ is not computable.

9.' A function is ***elementary*** if it can be obtained from the functions $s, n, u_j^i, +, \dot{-}$ by a finite sequence of applications of composition, bounded summation, and bounded product. (By application of bounded summation we mean obtaining the function $\sum_{t=0}^{y} f(t, x_1, \ldots, x_n)$ from $f(t, x_1, \ldots, x_n)$, and similarly for bounded product.)

(a) Show that every elementary function is primitive recursive.

(b) Show that $x \cdot y$, x^y, and $x!$ are elementary.

(c) Show that if $n + 1$-ary predicates P and Q are elementary, then so are $\sim P, P \vee Q, P \;\&\; Q, (\forall t)_{\leq y} P(t, x_1, \ldots, x_n)$, $(\exists t)_{\leq y} P(t, x_1, \ldots, x_n)$, and $\min_{t \leq y} P(t, x_1, \ldots, x_n)$.

(d) Show that Prime(x) is elementary.

(e) Let the binary function $\exp_y(x)$ be defined

$$\exp_0(x) = x$$

$$\exp_{y+1}(x) = 2^{\exp_y(x)}.$$

Show that for every elementary function $f(x_1, \ldots, x_n)$, there is a constant k such that $f(x_1, \ldots, x_n) \leq \exp_k(\max\{x_1, \ldots, x_n\})$. [*Hint:* Show that for every n there is an $m \geq n$ such that $x \cdot \exp_n(x) \leq \exp_m(x)$ for all x.]

(f) Show that $\exp_y(x)$ is not elementary. Conclude that the class of elementary functions is a proper subset of the class of primitive recursive functions.

8. Pairing Functions and Gödel Numbers

In this section we shall study two convenient coding devices which use primitive recursive functions. The first is for coding pairs of numbers by single numbers, and the second is for coding lists of numbers.

We define the primitive recursive function

$$\langle x, y \rangle = 2^x(2y + 1) \dot{-} 1.$$

Note that $2^x(2y + 1) \neq 0$ so

$$\langle x, y \rangle + 1 = 2^x(2y + 1).$$

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

$$\langle x, y \rangle = z, \tag{8.1}$$

namely, x is the largest number such that $2^x \mid (z+1)$, 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. (The twos have been "divided out.") Equation (8.1) thus defines functions

$$x = l(z), \qquad y = r(z).$$

Since Eq. (8.1) implies that $x, y < z + 1$ we have

$$l(z) \le z, \qquad r(z) \le z.$$

Hence we can write

$$l(z) = \min_{x \le z} [(\exists y)_{\le z}(z = \langle x, y \rangle)],$$
$$r(z) = \min_{y \le z} [(\exists x)_{\le z}(z = \langle x, y \rangle)],$$

so that $l(z), r(z)$ are primitive recursive functions.

The definition of $l(z), r(z)$ can be expressed by the statement

$$\langle x, y \rangle = z \Leftrightarrow x = l(z) \;\&\; y = r(z).$$

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 next obtain primitive recursive functions that encode and decode arbitrary finite sequences of numbers. The method we use, first employed by Gödel, depends on the prime power decomposition of integers.

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

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

Thus, the Gödel number of the sequence (3, 1, 5, 4, 6) is

$$[3, 1, 5, 4, 6] = 2^3 \cdot 3^1 \cdot 5^5 \cdot 7^4 \cdot 11^6.$$

For each fixed n, the function $[a_1, \ldots, a_n]$ is clearly primitive recursive.

Gödel numbering satisfies the following uniqueness property:

Theorem 8.2. If [a,, $\ldots, a_n] = [b_1, \ldots, b_n]$, then

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

This result is an immediate consequence of the uniqueness of the factorization of integers into primes, sometimes referred to as the ***unique factorisation theorem*** or the ***fundamental theorem of arithmetic.*** (For a proof, see any elementary number theory textbook.)

However, note that

$$[a_1, \ldots . a_n] = [a_1, \ldots, a_n, 0] \tag{8.2}$$

because $p_{n+1}^0 = 1$. This same result obviously holds for any finite number of zeros adjoined to the right end of a sequence. In particular, since

$$1 = 2^0 = 2^0 3^0 = 2^0 3^0 5^0 = \cdots,$$

it is natural to regard 1 as the Gödel number of the "empty" sequence of length 0, and it is useful to do so.

If one adjoins 0 to the left end of a sequence, the Gijdel number of the new sequence will not be the same as the Gijdel number of the original sequence. For example,

$$[2,3] = 2^2 \cdot 3^3 = 108,$$

and

$$[2,3,0] = 2^2 \cdot 3^3 \cdot 5^0 = 108,$$

but

$$[0,2,3] = 2^0 \cdot 3^2 \cdot 5^3 = 1125.$$

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 set

$$(x)_i = \min_{t \le x} (\sim p_i^{t+1} \mid x).$$

Note that $(x)_0 = 0$, and $(0)_i = 0$ for all i.

We shall also use the primitive recursive function

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

(Lt stands for "length.") Thus, if x = 20 = $2^2 \cdot 5^1 = [2,0,1]$, then $(x)_3 = 1$, but $(x)_4$ = (x), = --- = $(x)_{20}$ = 0. so, Lt(20) = 3. Also, Lt(0) = Lt(1) = 0.

If $x > 1$, and $\mathrm{Lt}(x) = n$, then p_n divides x but no prime greater than p_n divides x. Note that $\mathrm{Lt}([a_1, \ldots, a_n]) = n$ if and only if $a_n \neq 0$.

We summarize the key properties of these primitive recursive functions.

Theorem 8.3 (Sequence Number Theorem).

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

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

Our main application of these coding techniques is given in the next chapter. The following exercises indicate that they can also be used to show that PRC classes are closed under various interesting and useful forms of recursion.

Exercises

1. Let $f(x_1, \ldots, x_n)$ be a function of n variables, and let $f'(x)$ be a unary function defined so that $f'([x_1, \ldots, x_n]) = f(x_1, \ldots, x_n)$ for all $x_1, \ldots, x_n$. Show that f' is partially computable if and only if f is partially computable.
2. Define $\mathrm{Sort}([x_1, \ldots, x_n]) = [y_1, \ldots, y_n]$, where $y_1, \ldots, y_n$ is a permutation of $x_1, \ldots, x_n$ such that $y_1 \le y_2 \le \cdots \le y_n$. Show that Sort(x) is primitive recursive.
3. Let $F(0) = 0$, $F(1) = 1$, $F(n+2) = F(n+1) + F(n)$. [$F(n)$ is the nth so-called Fibonacci number.] Prove that $F(n)$ is primitive recursive.
4. (Simultaneous Recursion) Let

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

Prove that if f_1, f_2, g_1, g_2 all belong to some PRC class $\mathscr{C}$, then h_1, h_2 do also.

5.* (Course-of-Values Recursion)

(a) For $f(n)$ any function, we write

$$\tilde{f}(0) = 1, \tilde{f}(n) = [f(0), f(1), \ldots, f(n-1)] \text{ if } n \neq 0.$$

Let

$$f(n) = g(n, \tilde{f}(n))$$

for all n. Show that if g is primitive recursive so is f.

(b) Let

$$f(0) = 1, \qquad f(1) = 4, \qquad f(2) = 6,$$

$$f(x + 3) = f(x) + f(x + 1)^2 + f(x + 2)^3.$$

Show that $f(x)$ is primitive recursive.

(c) Let

$$h(0) = 3$$

$$h(x + 1) = \sum_{t=0}^{x} h(t).$$

Show that h is primitive recursive.

6.* (Unnested Double Recursion) Let

$$f(0, y) = g_1(y)$$

$$f(x + 1, 0) = g_2(x)$$

$$f(x + 1, y + 1) = h(x, y, f(x, y + 1), f(x + 1, y)).$$

Show that if g_1, g_2, and h all belong to some PRC class $\mathscr{C}$, then f also belongs to $\mathscr{C}$.

4

A Universal Program

1. Coding Programs by Numbers

We are going to associate with each program $\mathscr{P}$ of the language $\mathscr{S}$ a number, which we write $\#(\mathscr{P})$, in such a way that the program can be retrieved from its number. To begin with we arrange the variables in order as follows:

$$Y \;\; X_1 \; Z_1 \; X_2 \; Z_2 \; X_3 \; Z_3 \ldots .$$

Next we do the same for the labels:

$$A, \quad B_1 \, C_1 \, D_1 \, E_1 \;\; A, \;\; B_2 \, C_2 \, D_2 \, E_2 \;\; A,. \;\; . \;\; . \;\; .$$

We write $\#(V), \#(L)$ for the position of a given variable or label in the appropriate ordering. Thus $\#(X_2) = 4$, $\#(Z_1) = \#(Z) = 3$, $\#(E) = 5$, $\#(B_2) = 7$.

Now let I be an instruction (labeled or unlabeled) of the language $\mathscr{S}$. Then we write

$$\#(I) = \langle a, \langle b, c \rangle \rangle$$

where

1. if I is unlabeled, then $a = 0$; if I is labeled L, then $a = \#(L)$;
2. if the variable V is mentioned in I, then c = $\#(V) - 1$;

3. if the statement in I is

$$V \leftarrow V \qquad \text{or} \qquad V \leftarrow V+1 \qquad \text{or} \qquad V \leftarrow V-1,$$

then $b = 0$ or 1 or 2, respectively;
4. if the statement in I is

$$\text{IF } V \neq \text{OGOTO} L'$$

then $b = \#(L') + 2$.

Some examples:
The number of the unlabeled instruction $X \leftarrow X + 1$ is

$$\langle 0, \langle 1,1 \rangle \rangle = \langle 0,5 \rangle = 10,$$

whereas the number of the instruction

$$[A] \quad X \leftarrow X + 1$$

is

$$(1,\ (1,1)) = \langle 1,5 \rangle = 21.$$

Note that for any given number q there is a unique instruction I with #(I) = q. We first calculate $l(q)$. If Z(q) = 0, Z 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 Z will be

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

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

$$\#(\mathscr{P}) = [\#(I_1), \#(I_2), \ldots, \#(I_k)] - 1. \tag{1.1}$$

Since Gödel numbers tend to be very large, the number of even rather simple programs usually will be quite enormous. We content ourselves with a simple example:

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

The reader will recognize this as the example given in Chapter 2 of a program that computes the nowhere defined function. Calling these instructions I_1 and I_2, respectively, we have seen that $\#(I_1) = 21$. Since I_2 is unlabeled,

$$\#(I_2) = (0, \langle 3,1\rangle\rangle = \langle 0,23\rangle = 46.$$

Thus, finally, the number of this short program is

$$2^{21} \cdot 3^{46} - 1.$$

Note that the number of the unlabeled instruction Y ← Y is

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

Thus, by the ambiguity in Gödel numbers [recall Eq. (8.2), Chapter 3], the number of a program will be unchanged if an unlabeled Y ← Y is tacked onto its end. Of course 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 $\mathscr{S}$ the harmless stipulation that ***the final instruction in a program is not permitted to be the unlabeled statement $Y \leftarrow Y$.***

With this last stipulation each number determines a unique program. As an example, let us determine the program whose number is 199. We have

$$199 + 1 = 200 = 2^3 \cdot 3^0 \cdot 5^2 = [3,0,2].$$

Thus, if $\#(\mathscr{P}) = 199$, $\mathscr{P}$ consists of 3 instructions, the second of which is the unlabeled statement Y ← Y. We have

$$3 = (2,O) = (2, \langle 0,0\rangle\rangle$$

and

$$2 = \langle 0,1\rangle = \langle 0, \langle 1,0\rangle\rangle.$$

Thus, the program is

```
[B] Y ← Y
    Y ← Y
    Y ← Y + 1
```

a not very interesting program that computes the function y = 1.

Note also that the empty program has the number 1 − 1 = 0.

Exercises

1. Compute $\#(\mathscr{P})$ for $\mathscr{P}$ the programs of Exercises 4.1, 4.2, Chapter 2.
2. Find $\mathscr{P}$ such that $\#(\mathscr{P}) = 575$.

2. The Halting Problem

In this section we want to discuss a predicate HALT(x, y), which we now define. 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. To put it succinctly:

$$\text{HALT}(x, y) \Leftrightarrow \text{program number y eventually halts on input } x.$$

We now prove the remarkable:

Theorem 2.1. HALT(x, y) is not a computable predicate.

Proof. Suppose that HALT(x, y) were computable. Then we could construct the program $\mathscr{P}$:

$$[A] \quad \text{IF HALT}(X, \text{X}) \text{ GOT0 A}$$

(Of course $\mathscr{P}$ is to be the macro expansion of this program.) It is quite clear that $\mathscr{P}$ has been constructed so that

$$\psi_{\mathscr{P}}^{(1)}(x) = \begin{cases} \text{undefined} & \text{if} \quad \text{HALT}(x, x) \\ 0 & \text{if} \quad \sim \text{HALT}(x, \text{x}). \end{cases}$$

Let $\#(\mathscr{P}) = y_0$. Then using the definition of the HALT predicate,

$$\text{HALT}(x, y_0) \Leftrightarrow \sim \text{HALT}(x, \text{ x}).$$

Since this equivalence is true for all x, we can set $x = y_0$:

$$\text{HALT}(y_0, y_0) \Leftrightarrow \sim \text{HALT}(y_0, y_0).$$

But this is a contradiction. ■

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

> ***There is no algorithm that, given a program of $\mathscr{S}$ 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 haltingproblem.*** 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 $\mathscr{S}$. Hence this would contradict the fact that $\mathrm{HALT}(x,y)$ is not computable.

The last italicized assertion is a form of what has come to be called ***Church's thesis.*** We have already accumulated some evidence for it, and we will see more later. 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 fact, we will use Church's thesis freely in asserting the nonexistence of algorithms whenever we have shown that some problem cannot be solved by a program of $\mathscr{S}$.

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" problem should be made to realize that it is easy to construct relatively short programs (of $\mathscr{S}$) 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: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, etc. It is easy to write a program $\mathscr{P}$ of $\mathscr{S}$ that will search for a counterexample to Goldbach's conjecture, that is, an even number $n \geq 4$ that is not the sum of two primes. Note that the test that a given even number n is a counterexample only requires checking the primitive recursive predicate

$$\sim (\exists x)_{\leq n}(\exists y)_{\leq n}[\mathrm{Prime}(x) \; \& \mathrm{Prime}(y) \; \& \; x + y = n].$$

The statement that $\mathscr{P}$ never halts is equivalent to Goldbach's conjecture. Since the conjecture is still open after 250 years, nobody knows whether this program $\mathscr{P}$ will eventually halt.

Exercises

1. Show that $\mathrm{HALT}(x,x)$ is not computable.
2. Let $\overline{\mathrm{HALT}}(x,y)$ be defined

$$\overline{\mathrm{HALT}}(x,y) \Leftrightarrow \text{program number } y \text{ never halts on input } x.$$

   Show that $\overline{\mathrm{HALT}}(x,y)$ is not computable.
3. Let $\mathrm{HALT}^1(x)$ be defined $\mathrm{HALT}'(x) \Leftrightarrow \mathrm{HALT}(l(x), r(x))$. Show that $\mathrm{HALT}'(x)$ is not computable.

4. Prove or disprove: If $f(x_1, \ldots, x_n)$ is a total function such that for some constant k, $f(x_1, \ldots, x_n) \le k$ for all $x_1, \ldots, x_n$, then f is computable.
5. Suppose we claim that $\mathscr{P}$ is a program that computes $\text{HALT}(x, x)$. Give a counterexample that shows the claim to be false. That is, give an input x for which $\mathscr{P}$ gives the wrong answer.
6. Let

$$f(x) = \begin{cases} x & \text{if Goldbach's conjecture is true} \\ 0 & \text{otherwise.} \end{cases}$$

Show that $f(x)$ is primitive recursive.

3. Universality

The negative character of the results in the previous section might lead one to believe that it is not possible to compute in a useful way with numbers of programs. But, as we shall soon see, this belief is not justified.

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \psi_{\mathscr{P}}^{(n)}(x_1, \ldots, x_n), \qquad \text{where } \#(\mathscr{P}) = y.$$

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 $\Phi^{(n)}$. That is, we shall have for each $n > 0$,

$$\psi_{\mathscr{U}_n}^{(n+1)}(x_1, \ldots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \ldots, x_n, x_{n+1}).$$

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 $f(x) = \Phi^{(1)}(x, y) = \psi_{\mathscr{U}_1}^{(2)}(x, y)$. The program $\mathscr{U}_n$ will work very much like an interpreter. It must keep track of the current snapshot in a computation and by "decoding" the number of the program being interpreted, decide what to do next and then do it.

In writing the programs $\mathscr{U}_n$ we shall freely use macros corresponding to functions that we know to be primitive recursive using the methods of Chapter 3. We shall also freely ignore the rules concerning which letters may be used to represent variables or labels of $\mathscr{S}$.

In considering the state of a computation we can assume that all variables which are not given values have the value 0. With this understanding, we can code the state in which the ith variable in our list has the value a_i and all variables after the mth have the value 0, by the Gödel number $[a_1, \ldots, a_m]$. For example, the state

$$Y = 0, \qquad X_1 = 2, \qquad X_2 = 1$$

is coded by the number

$$[0,2,0,1] = 3^2 \cdot 7 = 63.$$

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

Now in the universal programs, we shall allocate storage as follows:

K will be the number such that the Kth instruction is about to be executed;

S will store the current state coded in the manner just explained.

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. Finally, we shall put the pieces together. We begin:

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

If $X_{n+1} = \#(\mathscr{P})$, where $\mathscr{P}$ consists of the instructions $I_1, \ldots, I_m$, then Z gets the value $[\#(I_1), \ldots, \#(I_m)]$ [see Eq. (1.1)]. S is initialized as $[0, X_1, 0, X_2, \ldots, 0, X_n]$, which gives the first n input variables their appropriate values and gives all other variables the value 0. K, the instruction counter, is given the initial value 1 (so that the computation can begin with the first instruction). Next,

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

If the computation has ended, GOTO F, where the proper value will be output. (The significance of $K = 0$ will be explained later.) Otherwise, the current instruction must be decoded and executed:

$$U \leftarrow r((Z)_K)$$
$$P \leftarrow p_{r(U)+1}$$

$(Z)_K = (a, \langle b,c\rangle)$ is the number of the Kth instruction. Thus, $U = (b, c)$ is the code for the *statement* about to be executed. The variable mentioned in the Kth 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:

```
IF Z(U) = 0 GOT0 N
IF Z(U) = 1 GOT0 A
IF $\sim (P \mid S)$ GOTO $N$
IF l(U) = 2 GOT0 A4
```

If $l(U) = 0$, the instruction is a dummy $V \leftarrow V$ and the computation need do nothing to S. If $l(U) = 1$, the instruction is of the form $V \leftarrow V + 1$, so that 1 has to be added to the exponent on P in the prime power factorization of S. The computation executes a GOT0 A (for Add). If $l(U) \neq 0, 1$, then the current instruction is either of the form $V \leftarrow V - 1$ or IF $V \neq 0$ GOT0 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 do *nothing* to S. If $P \mid S$ and $l(U) = 2$, then the computation executes a GOT0 M (for Minus), so that 1 can be subtracted from the exponent to which P divides S. To continue,

```
$K \leftarrow \min_{i \leq Lt(Z)} [l((Z)_i) + 2 = Z(U)]$
GOT0 C
```

If $l(U) > 2$ and $P \mid S$, the current instruction is of the form IF $V \neq 0$ GOT0 L where V has a nonzero value and L is the label whose position in our list is $l(U) - 2$. Accordingly the next instruction should be the first with this label. That is, K should get as its value the least i for which $l((Z)_i) = f(U) - 2$. If there is no instruction with the appropriate label, K gets the value 0, which will lead to termination the next time through the main loop. In either case the GOT0 C causes a "jump" to the beginning of the loop for the next instruction (if any) to be processed. Continuing,

```
[M]  $S \leftarrow \lfloor S/P \rfloor$
     GOT0 N
[A]  $S \leftarrow S \cdot P$
[N]  $K \leftarrow K + 1$
     GOT0 $C$
```

1 is subtracted or added to the value of the variable mentioned in the current instruction by dividing or multiplying S by P, respectively. The

```
        $Z \leftarrow X_{n+1} + 1$
        $S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$
        $K \leftarrow 1$
[C]     IF $K = \mathrm{Lt}(Z) + 1 \vee K = 0$ GOTO $F$
        $U \leftarrow r((Z)_K)$
        $P \leftarrow p_{r(U)+1}$
        IF $l(U) = 0$ GOTO $N$
        IF $l(U) = 1$ GOTO A
        IF $\sim(P \mid S)$ GOTO $N$
        IF $l(U) = 2$ GOTO $M$
        $K \leftarrow \min_{i \le \mathrm{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$
        GOTO C
[M]     $S \leftarrow \lfloor S/P \rfloor$
        GOTO $N$
[A]     $S \leftarrow S \cdot P$
[N]     $K \leftarrow K + 1$
        GOTO C
[F]     $Y \leftarrow (S)_1$
```

Figure 3.1. Program $\mathscr{U}_n$, which computes Y = $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$.

instruction counter is increased by 1 and the computation returns to process the next instruction. To conclude the program,

$$[F] \quad Y \leftarrow (S)_1$$

On termination, the value of Y for the program being simulated is stored as the exponent on p_1(= 2) in S. We have now completed our description of $\mathscr{U}_n$ and we put the pieces together in Fig. 3.1.

For each $n > 0$, the sequence

$$\Phi^{(n)}(x_1, \ldots, x_n, 0), \Phi^{(n)}(x_1, \ldots, x_n, 1), \ldots$$

enumerates all partially computable functions of n variables. When we want to emphasize this aspect of the situation we write

$$\Phi_y^{(n)}(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, y).$$

It is often convenient to omit the superscript when $n = 1$, writing

$$\Phi_y(x) = \Phi(x, y) = \Phi^{(1)}(x, y).$$

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

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

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 $\mathrm{STP}^{(n)}(x_1,\dots,x_n,y,t)$ is primitive recursive.

Proof. ***The*** idea is to provide numeric versions of the notions of snapshot and successor snapshot and to show that the necessary functions are primitive recursive. We use the same representation of program states that we used in defining the universal programs, and if z represents state σ, then $\langle i, z\rangle$ represents the snapshot (i,σ).

We begin with some functions for extracting the components of the ith instruction of program number y:

$$\begin{aligned}\mathrm{LABEL}(i,y) &= l((y+1)_i)\\ \mathrm{VAR}(i,y) &= r(r((y+1)_i))+1\\ \mathrm{INSTR}(i,y) &= l(r((y+1)_i))\\ \mathrm{LABEL}'(i,y) &= l(r((y+1)_i)) \dot{-} 2\end{aligned}$$

Next we define some predicates that indicate, for program y and the snapshot represented by x, which kind of action is to be performed next.

$$\begin{aligned}\mathrm{SKIP}(x,y) \Leftrightarrow{}& [\mathrm{INSTR}(l(x),y) = 0 \;\&\; l(x) \le \mathrm{Lt}(y+1)]\\ &\vee \left[\mathrm{INSTR}(l(x),y) \ge 2 \;\&\; {\sim}\left(p_{\mathrm{VAR}(l(x),y)} \mid r(x)\right)\right]\\ \mathrm{INCR}(x,y) \Leftrightarrow{}& \mathrm{INSTR}(l(x),y) = 1\\ \mathrm{DECR}(x,y) \Leftrightarrow{}& \mathrm{INSTR}(l(x),y) = 2 \;\&\; p_{\mathrm{VAR}(l(x),y)} \mid r(x)\\ \mathrm{BRANCH}(x,y) \Leftrightarrow{}& \mathrm{INSTR}(l(x),y) > 2 \;\&\; p_{\mathrm{VAR}(l(x),y)} \mid r(x)\\ &\& \; (\exists i)_{\le \mathrm{Lt}(y+1)} \mathrm{LABEL}(i,y) = \mathrm{LABEL}'(l(x),y)\end{aligned}$$

Now we can define SUCC (x, y), which, for program number y, gives the representative of the successor to the snapshot represented by x.

$$\mathrm{SUCC}(x,y) = \begin{cases} \langle l(x)+1, r(x)\rangle & \text{if } \mathrm{SKIP}(x,y) \\ \langle l(x)+1, r(x)\cdot p_{\mathrm{VAR}(l(x),y)}\rangle & \text{if } \mathrm{INCR}(x,y) \\ \langle l(x)+1, \lfloor r(x)/p_{\mathrm{VAR}(l(x),y)}\rfloor\rangle & \text{if } \mathrm{DECR}(x,y) \\ \langle \min_{i \le \mathrm{Lt}(y+1)}[\mathrm{LABEL}(i,y) = \mathrm{LABEL}'(l(x),y)], r(x)\rangle & \\ & \text{if } \mathrm{BRANCH}(x,y) \\ \langle \mathrm{Lt}(y+1)+1, r(x)\rangle & \text{otherwise.} \end{cases}$$

We also need

$$\mathrm{INIT}^{(n)}(x_1,\ldots,x_n) = \langle 1, \prod_{i=1}^{n} (p_{2i})^{x_i}\rangle,$$

which gives the representation of the initial snapshot for inputs $x_1,\ldots,x_n$, and

$$\mathrm{TERM}(x,y) \Leftrightarrow l(x) > \mathrm{Lt}(y+1),$$

which tests whether x represents a terminal snapshot for program y.

Putting these together we can define a primitive recursive function that gives the numbers of the successive snapshots produced by a given program.

$$\mathrm{SNAP}^{(n)}(x_1,\ldots,x_n,y,0) = \mathrm{INIT}^{(n)}(x_1,\ldots,x_n)$$

$$\mathrm{SNAP}^{(n)}(x_1,\ldots,x_n,y,i+1) = \mathrm{SUCC}(\mathrm{SNAP}^{(n)}(x_1,\ldots,x_n,y,i),y)$$

Thus,

$$\mathrm{STP}^{(n)}(x_1,\ldots,x_n,y,t) \Leftrightarrow \mathrm{TERM}(\mathrm{SNAP}^{(n)}(x_1,\ldots,x_n,y,t),y),$$

and it is clear that $\mathrm{STP}^{(n)}(x_1,\ldots,x_n,y,t)$ 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

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

Proof. Let y_0 be the number of a program that computes $f(x_1, \ldots, x_n)$. We shall prove the following equation, which clearly implies the desired result:

$$f(x_1, \ldots, x_n) = \left(\min_z R(x_1, \ldots, x_n, z)\right) \tag{3.1}$$

where $R(x_1, \ldots, x_n, z)$ is the predicate

$$\begin{aligned} &\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \\ &\qquad \& \, (r(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_1 \\ &\quad = l(z). \end{aligned}$$

First consider the case when the righthand side of this equation is defined. Then, in particular, there exists a number z such that

$$\begin{aligned} &\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, r(z)) \\ &\qquad \text{and } (r(\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, y_0, r(z))))_1 \\ &\quad = l(z). \end{aligned}$$

For any such z, the computation by the program with number y_0 has reached a terminal snapshot in $r(z)$ or fewer steps and $l(z)$ is the value held in the output variable Y, i.e., $l(z) = f(x_1, \ldots, x_n)$.

If, on the other hand, the right side is undefined, it must be the case that $\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y_0, t)$ is false for all values of t, i.e., $f(x_1, \ldots, x_n)\uparrow$. ■

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.

Proof. That every function which can be so obtained is partially computable is an immediate consequence of Theorems 1.1, 2.1, 2.2, 3.1, and 7.2 in Chapter 3. Note that a partially computable predicate is necessarily computable, so Theorem 7.2 covers all applications of minimalization to a predicate obtained as described in the theorem.

Conversely, we can use the normal form theorem to write any given partially computable function in the form

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

where R is a primitive recursive predicate and so is obtained from the initial functions by a finite number of applications of composition and recursion. Finally, our given function is obtained from R by one use of minimalization and then by composition with the primitive recursive function 1. ■

When min, $R(x_1, \ldots, x_n, y)$ 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, $R(x_1, \ldots, x_n, y)$ 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.

Exercises

1. Show that for each u, there are infinitely many different numbers v such that for all x, $\Phi_u(x) = \Phi_v(x)$.

2. **(a)** Let

$$H_1(x) = \begin{cases} 1 & \text{if } \Phi(x,x)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

Show that H,(x) is partially computable.

(b) Let $A = \{a,, \ldots, a,\}$ be a finite set such that $\Phi(a_i, a_i)\uparrow$ for $1 \leq i \leq n$, and let

$$H_2(x) = \begin{cases} 1 & \text{if } \Phi(x,x)\downarrow \\ 0 & \text{if } x \in A \\ \uparrow & \text{otherwise.} \end{cases}$$

Show that $H_2(x)$ is partially computable.

(c) Give an infinite set B such that $\Phi(b,b)\uparrow$ for all $b \in B$ and such that

$$\text{H,(x)} = \begin{cases} 1 & \text{if } \Phi(x,\text{x})\downarrow \\ 0 & \text{if } x \in B \\ \uparrow & \text{otherwise} \end{cases}$$

is partially computable.

(d) Give an infinite set C such that $\Phi(c,c)\uparrow$ for all $c \in C$ and such that

$$H_4(x) = \begin{cases} 1 & \text{if } \Phi(x,x)\downarrow \\ 0 & \text{if } x \in C \\ \uparrow & \text{otherwise} \end{cases}$$

is not partially computable.

3. Give a program $\mathscr{P}$ such that $H_{\mathscr{P}}(x_1, x_2)$, defined

$$H_{\mathscr{P}}(x_1, x_2) \Leftrightarrow \text{program } \mathscr{P} \text{ eventually halts on inputs } x_1, x_2$$

is not computable.

4. Let $f(x_1, \ldots, x_n)$ be computed by program $\mathscr{P}$, and suppose that for some primitive recursive function $g(x_1, \ldots, x_n)$,

$$\text{STP}^{(n)}(x_1, \ldots, x_n, \#(\mathscr{P}), g(x_1, \ldots, x_n))$$

is true for all $x_1, \ldots, x_n$. Show that $f(x_1, \ldots, x_n)$ is primitive recursive.

5.* Give a primitive recursive function counter(x) such that if Φ_n is a computable predicate, then

$$\Phi_n(\text{counter}(n)) \Leftrightarrow \sim \text{HALT}(\text{counter}(n), \text{counter}(n)).$$

That is, counter(n) is a counterexample to the possibility that Φ_n computes HALT(x, x). [Compare this exercise with Exercise 2.5.]

6.* Give an upper bound on the length of the shortest $\mathscr{S}$ program that computes the function $\Phi_y(x)$.

4. Recursively Enumerable Sets

The close relation between predicates and sets, as described in Chapter 1, lets us use the language of sets in talking about solvable and unsolvable problems. For example, the predicate HALT(x, y) is the characteristic function of the set $\{(x,y) \in N^2 \mid \text{HALT}(x,y)\}$. 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. Thus, in particular, to say that the set B is computable or recursive is just to say that $P(x_1, \ldots, x_m)$ is a computable function. Likewise, B is a primitive recursive set if $P(x_1, \ldots, x_m)$ is a primitive recursive predicate.

We have, for example,

Theorem 4.1. Let the sets B, C belong to some PRC class $\mathscr{C}$. Then so do the sets $B \cup C$, $B \cap C$, $\bar{B}$.

Proof. This is an immediate consequence of Theorem 5.1, Chapter 3. ■

As long as the Gödel numbering functions $[x_1, \ldots, x_n]$ and $(x)_i$ are availaole, we can restrict our attention to subsets of N. We have, for example,

Theorem 4.2. Let $\mathscr{C}$ be a PRC class, and let B be a subset of N”, $m \geq 1$. 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 $\mathscr{C}$.

Proof. If $P_B(x_1, \ldots, x_m)$ is the characteristic function of B, then

$$P_{B'}(x) \Leftrightarrow P_B((x)_1, \ldots, (x)_m) \; \& \; \mathrm{Lt}(x) = m$$

is the characteristic function of B', and $P_{B'}$ clearly belongs to $\mathscr{C}$ if P_B belongs to $\mathscr{C}$. On the other hand, if $P_{B'}(x)$ is the characteristic function of B’, then

$$P_B(x_1, \ldots, x_m) \Leftrightarrow P_{B'}([x_1, \ldots, x_m])$$

is the characteristic function of B, and P_B clearly belongs to $\mathscr{C}$ if $P_{B'}$ 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. The set $B \subseteq N$ is called ***recursively enumerable*** if there is a partially computable function g(x) such that

$$B = \{x \in N \mid g(x)\downarrow\}. \tag{4.1}$$

The term ***recursively enumerable*** is usually abbreviated ***r.e.*** A set is recursively enumerable just when it is the domain of a partially computable function. If $\mathscr{P}$ is a program that computes the function g in (4.1), 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 a "yes" answer; but for numbers that do not, the algorithm will never terminate. If we invoke Church's thesis, r.e. sets B may be thought of intuitively as sets for which there exist algorithms related to B as in the previous sentence, but without stipulating that the algorithms be expressed by programs of the language $\mathscr{S}$. Such algorithms, sometimes called ***semi-decision procedures,*** provide a kind of "approximation" to solving the problem of testing membership in B.

We have

Theorem 4.3. If B is a recursive set, then B is r.e.

Proof. Consider the program $\mathscr{P}$:

$$[A] \quad \text{IF} \sim (X \in B) \text{ GOTO } A$$

Since B is recursive, the predicate $x \in B$ is computable and $\mathscr{P}$ can be expanded to a program of $\mathscr{S}$. Let $\mathscr{P}$ compute the function h(x). Then, clearly,

$$B = \{x \in N \mid h(x)\downarrow\}. \qquad \blacksquare$$

If B and $\bar{B}$ are both r.e., we have a pair of algorithms that will terminate in case a given input is or is not in B, respectively. We can think of combining these two algorithms to obtain a single algorithm that will always terminate and that will tell us whether a given input belongs to B. This combined algorithm might work by "running" the two separate algorithms for longer and longer times until one of them terminates. This method of combining algorithms is called ***dovetailing,*** and the step-counter theorem enables us to use it in a rigorous manner.

Theorem 4.4. The set B is recursive if and only if B and $\bar{B}$ are both r.e.

Proof. If B is recursive, then by Theorem 4.1 so is $\bar{B}$, and hence by Theorem 4.3, they are both r.e.

Conversely, if B and $\bar{B}$ are both r.e., we may write

$$B = \{x \in N \mid g(x)\downarrow\},$$

$$\bar{B} = \{x \in N \mid h(x)\downarrow\},$$

where g and h are both partially computable. Let g be computed by program $\mathscr{P}$ and h be computed by program $\mathscr{Q}$, and let $p = \#(\mathscr{P})$, $q = \#(\mathscr{Q})$. Then the program that follows computes B. (That is, the program computes the characteristic function of B.)

```
[A]    IF $STP^{(1)}(X,p,T)$ GOT0 C
       IF $STP^{(1)}(X,q,T)$ GOT0 $E$
       $T \leftarrow T + 1$
       GOT0 $A$
[C]    $Y \leftarrow 1$
```
■

Theorem 4.5. If B and C are r.e. sets so are B u C and $B \cap$ C.

Proof. Let

$$B = \{x \in N \mid g(x)\downarrow\},$$
$$C = \{x \in N \mid h(x)\downarrow\},$$

where g and h are both partially computable. Let $f(x)$ be the function computed by the program

```
$Y \leftarrow g(X)$
$Y \leftarrow h(X)$
```

Then $f(x)$ is defined if and only if g(x) and $h(x)$ are both defined. Hence

$$B \cap C = \{x \in N \mid f(x)\downarrow\},$$

so that $B \cap$ C is also r.e.

To obtain the result for $B \cup$ C we must use dovetailing again. Let g and h be computed by programs $\mathscr{P}$ and $\mathscr{Q}$, respectively, and let $\#(\mathscr{P}) = p$, $\#(\mathscr{Q}) = q$. Let $k(x)$ be the function computed by the program

```
[A]    IF $STP^{(1)}(X,p,T)$ GOT0 $E$
       IF $STP^{(1)}(X,q,T)$ GOT0 $E$
       $T \leftarrow T + 1$
       GOT0 $A$'
```

Then k(x) is defined just in case ***either g(x) or h(x)*** is defined. That is,

$$B \cup C = \{x \in N \mid k(x)\downarrow\}.$$ ■

Definition. We write

$$W_n = \{x \in N \mid \Phi(x,n)\downarrow\}.$$

Then we have

Theorem 4.6 (Enumeration Theorem). A set B is r.e. if and only if there is an n for which $B = W_n$.

Proof. This is an immediate consequence of the definition of $\Phi(x, n)$. ■

The theorem gets its name from the fact that the sequence

$$W_0, W_1, W_2, \ldots$$

is an enumeration of all r.e. sets.

We define

$$K = \{n \in N \mid n \in W_n\}.$$

Now,

$$n \in W_n \Leftrightarrow \Phi(n, n)\downarrow \Leftrightarrow \text{HALT}(n, n).$$

Thus, K is the set of all numbers n such that program number n eventually halts on input n. We have

Theorem 4.7. K is r.e. but not recursive.

Proof. Since $K = \{n \in N \mid \Phi(n, n)\downarrow\}$ and (by the universality theorem-Theorem 3.1), $\Phi(n, n)$ is certainly partially computable, K is clearly r.e. If $\bar{K}$ were also r.e., by the enumeration theorem we would have

$$\bar{K} = W_i$$

for some i. Then

$$i \in K \Leftrightarrow i \in W_i \Leftrightarrow i \in \bar{K},$$

which is a contradiction. ■

Actually the ***proof*** of Theorem 2.1 already shows not only that $\text{HALT}(x, z)$ is not computable, but also that $\text{HALT}(x, x)$ is not computable, i.e., that K is not a recursive set. (This was Exercise 2.1.)

We conclude this section with some alternative ways of characterizing r.e. sets.

Theorem 4.8. Let B be an r.e. set. Then there is a primitive recursive predicate $R(x, t)$ such that $B = \{x \in N \mid (\exists t) R(x, t)\}$.

Proof. Let $B = W_n$. Then $B = \{x \in N \mid (\exists t) \text{STP}^{(1)}(x, n, t)\}$, and $\text{STP}^{(1)}$ is primitive recursive by Theorem 3.2. ■

Theorem 4.9. Let S be a nonempty r.e. set. Then there is a primitive recursive function $f(u)$ such that $S = \{f(n) \mid n \in N\} = \{f(0), f(1), f(2), \ldots\}$. That is, S is the range of f.

Proof. By Theorem 4.8

$$S = \{x \mid (\exists t)R(x,t)\},$$

where R is a primitive recursive predicate. Let x_0 be some fixed member of S (for example, the smallest). Let

$$f(u) = \begin{cases} l(u) & \text{if } R(l(u), r(u)) \\ x_0 & \text{otherwise.} \end{cases}$$

Then by Theorem 5.4 in Chapter 3, f is primitive recursive. Each value $f(u)$ is in S, since x_0 is automatically in S, while if $R(l(u), r(u))$ is true, then certainly $(\exists t)R(l(u),t)$ is true, which implies that $f(u) = l(u) \in S$. Conversely, if $x \in S$, then $R(x,t_0)$ is true for some t_0. Then

$$f(\langle x,t_0\rangle) = l(\langle x,t_0\rangle) = x,$$

so that $x = f(u)$ for $u = \langle x,t_0\rangle$. ■

Theorem 4.10. Let $f(x)$ be a partially computable function and let $S = \{f(x) \mid f(x)\downarrow\}$. (That is, S is the ***range*** of f.) Then S is r.e.

Proof. Let

$$g(x) = \begin{cases} 0 & \text{if } x \in S \\ \uparrow & \text{otherwise.} \end{cases}$$

Since

$$S = \{x \mid g(x)\downarrow\},$$

it suffices to show that g(x) is partially computable. Let $\mathscr{P}$ be a program that computes f and let $\#(\mathscr{P}) = p$. Then the following program computes $g(x)$:

```
[A]  IF $\sim$STP$^{(1)}(Z,p,T)$ GOTO $B$
     $V \leftarrow f(Z)$
     IF $V = X$ GOTO $E$
[B]  $Z \leftarrow Z + 1$
     IF $Z \leq T$ GOTO $A$
     $T \leftarrow T + 1$
     $Z \leftarrow 0$
     GOTO A
```

Note that in this program the macro expansion of $V \leftarrow f(Z)$ will be entered only when the step-counter test has already guaranteed that f is defined. ■

Combining Theorems 4.9 and 4.10, we have

Theorem 4.11. Suppose that $S \neq 0$. Then the following statements are all equivalent:

1. S is r.e.;
2. S is the range of a primitive recursive function;
3. S is the range of a recursive function;
4. S is the range of a partial recursive function.

Proof. *By* Theorem 4.9, (1) implies (2). Obviously, (2) implies (3), and (3) implies (4). By Theorem 4.10, (4) implies (1). Hence all four statements are equivalent. ■

Theorem 4.11 provides the motivation for the term ***recursively enumerable.*** In fact, such a set (if it is nonempty) is enumerated by a recursive function.

Exercises

1. Let B be a subset of N^m, $m > 1$. We say that B is r.e. if $B = \{(x_1, \ldots, x_m) \in N^m \mid g(x_1, \ldots, x_m)\downarrow\}$ for some partially computable function $g(x_1, \ldots, x_m)$. Let

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

   Show that B' is r.e. if and only if B is r.e.

2. Let $K_0 = \{(x, y) \mid x \in W_y\}$. Show that K_0 is r.e.
3. Let f be an n-ary partial function. The graph of f, denoted $gr(f)$, is the set $\{[x_1, \ldots, x_n, f(x_1, \ldots, x_n)] \mid f(x_1, \ldots, x_n)\downarrow\}$.
   (a) Let $\mathscr{C}$ be a PRC class. Prove that if f belongs to $\mathscr{C}$ then $gr(f)$ belongs to $\mathscr{C}$.
   (b) Prove that if $gr(f)$ is recursive then f is partially computable.
   (c) Prove that the recursiveness of $gr(f)$ does not necessarily imply that f is computable.
4. Let $B = \{f(n) \mid n \in N\}$, where f is a strictly increasing computable function [i.e., $f(n+1) > f(n)$ for all n]. Prove that B is recursive.
5. Show that every infinite r.e. set has an infinite recursive subset.
6. Prove that an infinite set A is r.e. if and only if $A = \{f(n) \mid n \in N\}$ for some one-one computable function $f(x)$.

7. Let A, B be sets. Prove or disprove:
   (a) If $A \cup B$ is r.e., then A and B are both r.e.
   (b) If $A \subseteq B$ and B is r.e., then A is r.e.
8. Show that there is no computable function $f(x)$ such that $f(x) = \Phi(x, x) + 1$ whenever $\Phi(x, x)\downarrow$.
9. (a) Let g(x), h(x) be partially computable functions. Show there is a partially computable function f(x) such that $f(x)\downarrow$ for precisely those values of x for which either $g(x)\downarrow$ or $h(x)\downarrow$ (or both) and such that when $f(x)\downarrow$, either $f(x) = g(x)$ or $f(x) = h(x)$.
   (b) Can f be found fulfilling all the requirements of **(a)** but such that in addition $f(x) = g(x)$ whenever $g(x)\downarrow$? Proof?
10. **(a)** Let $A = \{y \mid (\exists t)P(t, y)\}$, where P is a computable predicate. Show that A is r.e.
    (b) Let $B = \{y \mid (\exists t_1) \cdots (\exists t_n)Q(t_1, \ldots, t_n, y)\}$, where Q is a computable predicate. Show that B is r.e.
11. Give a computable predicate $R(x, y)$ such that $\{y \mid (\forall t)R(t, y)\}$ is not r.e.

5. The Parameter Theorem

The parameter theorem (which has also been called the ***iteration theorem*** and the ***s-m-n theorem)*** is an important technical result that relates the various functions $\Phi^{(n)}(x_1, x_2, \ldots, x_n, y)$ for different values of n.

Theorem 5.1 (Parameter Theorem). For each $n, m > 0$, there is a primitive recursive function $S_m^n(u_1, u_2, \ldots, u_n, y)$ such that

$$\Phi^{(m+n)}(x_1, \ldots, x_m, u_1, \ldots, u_n, y) = \Phi^{(m)}(x_1, \ldots, x_m, S_m^n(u_1, \ldots, u_n, y)). \tag{5.1}$$

Suppose that values for variables $u_1, \ldots, u_n$ are fixed and we have in mind some particular value of y. Then the left side of (5.1) is a partially computable function of the m arguments $x_1, \ldots, x_m$. Letting q be the number of a program that computes this function of m variables, we have

$$\Phi^{(m+n)}(x_1, \ldots, x_m, u_1, \ldots, u_n, y) = \Phi^{(m)}(x_1, \ldots, x_m, q).$$

The parameter theorem tells us that not only does there exist such a number q, but that it can be obtained from $u_1, \ldots, u_n, y$ in a computable (in fact, primitive recursive) way.

Proof. ***The*** proof is by mathematical induction on ***n.***

For ***n*** = 1, we need to show that there is a primitive recursive function $S_m^1(u, y)$ such that

$$\Phi^{(m+1)}(x_1, \ldots, x_m, u, y) = \Phi^{(m)}(x_1, \ldots, x_m, S_m^1(u, y)).$$

Here $S_m^1(u, y)$ must be the number of a program which, given m inputs $x_1, \ldots, x_m$, computes the same value as program number y does when given the $m + 1$ inputs $x_1, \ldots, x_m, u$. Let $\mathscr{P}$ be the program such that $\#(\mathscr{P})$ = y. Then $S_m^1(u$, y) can be taken to be the number of a program which first gives the variable X_{m+1} the value u and then proceeds to carry out $\mathscr{P}$. X_{m+1} will be given the value u by the program

$$\left.\begin{array}{l} X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ X_{m+1} \leftarrow X_{m+1} + 1 \end{array}\right\} u$$

The number of the unlabeled instruction

$$X_{m+1} \leftarrow X_{m+1} + 1$$

is

$$\langle 0, \langle 1, 2m+1 \rangle\rangle = 16m + 10.$$

So we may take

$$S_m^1(u, y) = \left[\left(\prod_{i=1}^{u} p_i \right)^{16m+10} \prod_{j=1}^{\operatorname{Lt}(y+1)} p_{u+j}^{(y+1)_j} \right] \dot{-} 1,$$

a primitive recursive function. Here the numbers of the instructions of $\mathscr{P}$ which appear as exponents in the prime power factorization of y + 1 have been shifted to the primes $p_{u+1}, p_{u+2}, \ldots, p_{u+\operatorname{Lt}(y+1)}$.

To complete the proof, suppose the result known for ***n = k.*** Then we have

$$\begin{aligned} &\Phi^{(m+k+1)}(x_1, \ldots, x_m, u_1, \ldots, u_k, u_{k+1}, y) \\ &\quad = \Phi^{(m+k)}(x_1, \ldots, x_m, u_1, \ldots, u_k, S_{m+k}^1(u_{k+1}, y)) \\ &\quad = \Phi^{(m)}(x_1, \ldots, x_m, S_m^k(u_1, \ldots, u_k, S_{m+k}^1(u_{k+1}, y))), \end{aligned}$$

using first the result for $n = 1$ and then the induction hypothesis. But now, if we define

$$S_m^{k+1}(u_1, \ldots, u_k, u_{k+1}, y) = S_m^k(u_1, \ldots, u_k, S_{m+k}^1(u_{k+1}, y)),$$

we have the desired result. ■

We next give a sample application of the parameter theorem. It is desired to find a computable function $g(u,v)$ such that

$$\Phi_u(\Phi_v(x)) = \Phi_{g(u,v)}(x).$$

We have by the meaning of the notation that

$$\Phi_u(\Phi_v(x)) = \Phi(\Phi(x,v),u)$$

is a partially computable function of x, u, v. Hence, we have

$$\Phi_u(\Phi_v(X)) = \Phi^{(3)}(x, u, v, z_0)$$

for some number z_0. By the parameter theorem,

$$\Phi^{(3)}(x, u, v, z_0) = \Phi(x, S_1^2(u, v, z_0)) = \Phi_{S_1^2(u,v,z_0)}(x).$$

Exercises

1. Given a partially computable function $f(x,y)$, find a primitive recursive function $g(u,v)$ such that

$$\Phi_{g(u,v)}(x) = f(\Phi_u(x), \Phi_v(x)).$$

2. Show that there is a primitive recursive function $g(u, v, w)$ such that

$$\Phi^{(3)}(u, v, w, z) = \Phi_{g(u,v,w)}(z).$$

3. Let us call a partially computable function $g(x)$ ***extendable*** if there is a computable function $f(x)$ such that $f(x) = g(x)$ for all x for which $g(x)\downarrow$. Show that there is no algorithm for determining of a given z whether or not $\Phi_z(x)$ is extendable. [*Hint:* Exercise 8 of Section 4 shows that $\Phi(x,x) + 1$ is not extendable. Find an extendable function k(x) such that the function

$$h(x, t) = \begin{cases} \Phi(x,x) + 1 & \text{if } \Phi(t,t)\downarrow \\ k(x) & \text{otherwise} \end{cases}$$

is partially computable.]

4.* A ***programming system*** is an enumeration $S = \{\phi_i^{(n)} \mid i \in N, n > 0\}$ of the partially computable functions. That is, for each partially computable function $f(x_1, \ldots, x_n)$ there is an i such that f is $\phi_i^{(n)}$.

(a) A programming system S is ***universal*** if for each $n > 0$, the function $\Psi^{(n)}$, defined

$$\Psi^{(n)}(x_1, \ldots, x_n, i) = \phi_i^{(n)}(x_1, \ldots, x_n),$$

is partially computable. That is, S is universal if a version of the universality theorem holds for S. Obviously,

$$\{\Phi_i^{(n)} \mid i \in \mathrm{N},\ n > 0\}$$

is a universal programming system. Prove that a programming system S is universal if and only if for each $n > 0$ there is a computable function f_n such that $\phi_i^{(n)} = \Phi_{f_n(i)}^{(n)}$ for all i.

(b) A universal programming system S is ***acceptable*** if for each $n, m > 0$ there is a computable function $s_m^n(u_1, \ldots, u_n, y)$ such that

$$\Psi^{(m+n)}(x_1, \ldots, x_m, u_1, \ldots, u_n, y)$$
$$= \Psi^{(m)}(x_1, \ldots, x_m, s_m^n(u_1, \ldots, u_n, y)).$$

That is, S is acceptable if a version of the parameter theorem holds for S. Again, $\{\Phi_i^{(n)} \mid i \in N, n > 0\}$ is obviously an acceptable programming system. Prove that S is acceptable if and only if for each $n > 0$ there is a computable function g_n such that $\Phi_i^{(n)} = \phi_{g_n(i)}^{(n)}$ for all i.

6. Diagonalization and Reducibility

So far we have seen very few examples of nonrecursive sets. We now discuss two general techniques for proving that given sets are not recursive or even that they are not r.e. The first method, ***diagonalization,*** turns on the demonstration of two assertions of the following sort:

1. A certain set A can be enumerated in a suitable fashion.
2. It is possible, with the help of the enumeration, to define an object b that is different from every object in the enumeration, i.e., $b \notin A$.

We sometimes say that b is defined by *diagonalizing* ***over A.*** In some diagonalization arguments the goal is simply to find some ***b 4 A.*** We will give an example of such an argument later in the chapter. The arguments we will consider in this section have an additional twist: the definition of b is such that ***b must belong to A,*** contradicting the assertion that we began with an enumeration of *all* of the elements in A. The end of the argument, then, is to draw some conclusion from this contradiction.

For example, the proof given for Theorem 2.1 is a diagonalization argument that the predicate HALT(x, y), or equivalently, the set

$$\{(x,y) \in N^2 \mid \text{HALT}(x,y)\},$$

is not computable. The set A in this case is the class of unary partially computable functions, and assertion 1 follows from the fact that $\mathscr{S}$ programs can be coded as numbers. For each n, let $\mathscr{P}_n$ be the program with number n. Then all unary partially computable functions occur among $\psi^{(1)}_{\mathscr{P}_0}; \psi^{(1)}_{\mathscr{P}_1}, \ldots$. We began by assuming that HALT(x, y) is computable, and we wrote a program $\mathscr{P}$ that computes $\psi^{(1)}_{\mathscr{P}}$. The heart of the proof consisted of showing that $\psi^{(1)}_{\mathscr{P}}$ does not appear among $\psi^{(1)}_{\mathscr{P}_0}, \psi^{(1)}_{\mathscr{P}_1}, \ldots$. In particular, we wrote $\mathscr{P}$ so that for every $x, \psi^{(1)}_{\mathscr{P}}(x)\downarrow$ if and only if $\psi^{(1)}_{\mathscr{P}_x}(x)\uparrow$, i.e.,

$$\text{HALT}(x, \#(\mathscr{P})) \Leftrightarrow {\sim}\,\text{HALT}(x,x),$$

so $\psi^{(1)}_{\mathscr{P}}$ differs from each function $\psi^{(1)}_{\mathscr{P}_0}, \psi^{(1)}_{\mathscr{P}_1}, \ldots$ on at least one input value. That is, n is a counterexample to the possibility that $\psi^{(1)}_{\mathscr{P}}$ is $\psi^{(1)}_{\mathscr{P}_n}$, since $\psi^{(1)}_{\mathscr{P}}(n)\downarrow$ if and only if $\psi^{(1)}_{\mathscr{P}_n}(n)\uparrow$. *Now we* have the unary partially computable function $\psi^{(1)}_{\mathscr{P}}$ that is not among $\psi^{(1)}_{\mathscr{P}_0}, \phi^{(1)}_{\mathscr{P}_1}, \ldots$, so assertion 2 is satisfied, giving us a contradiction. In the proof of Theorem 2.1 the contradiction was expressed a bit differently: Because $\psi^{(1)}_{\mathscr{P}}$ is partially computable, it *must* appear among $\psi^{(1)}_{\mathscr{P}_0}, \psi^{(1)}_{\mathscr{P}_1}, \ldots$, and, in particular, it must be $\psi^{(1)}_{\mathscr{P}_{\#(\mathscr{P})}}$, since $\mathscr{P}_{\#(\mathscr{P})}$ is $\mathscr{P}$ by definition, but we have the counterexample $\psi^{(1)}_{\mathscr{P}}(\#(\mathscr{P}))\downarrow$ if and only if $\psi^{(1)}_{\mathscr{P}_{\#(\mathscr{P})}}(\#(\mathscr{P}))\uparrow$, i.e.,

$$\text{HALT}(\#(\mathscr{P}), \#(\mathscr{P})) \Leftrightarrow {\sim}\,\text{HALT}(\#(\mathscr{P}), \#(\mathscr{P})).$$

Since we know assertion 1 to be true, and since assertion 2 depended on the assumption that HALT(x, y) is computable, HALT(x, y) cannot be computable.

To present the situation more graphically, we can represent the values of each function $\psi^{(1)}_{\mathscr{P}_0}, \psi^{(1)}_{\mathscr{P}_1}, \ldots$ by the infinite array

$$\begin{array}{cccc}
\boxed{\psi^{(1)}_{\mathscr{P}_0}(0)} & \psi^{(1)}_{\mathscr{P}_0}(1) & \psi^{(1)}_{\mathscr{P}_0}(2) & \cdots \\[1em]
\psi^{(1)}_{\mathscr{P}_1}(0) & \boxed{\psi^{(1)}_{\mathscr{P}_1}(1)} & \psi^{(1)}_{\mathscr{P}_1}(2) & \cdots \\[1em]
\psi^{(1)}_{\mathscr{P}_2}(0) & \psi^{(1)}_{\mathscr{P}_2}(1) & \boxed{\psi^{(1)}_{\mathscr{P}_2}(2)} & \cdots \\
\vdots & \vdots & \vdots &
\end{array}$$

Each row represents one function. It is along the ***diagonal*** of this array that we have arranged to find the counterexamples, which explains the origin of the term ***diagonalization.***

We can use a similar argument to give an example of a non-r.e. set. Let TOT be the set of all numbers p such that p is the number of a program that computes a total function $f(x)$ of one variable. That is,

$$\text{TOT} = \{z \in N \mid (\forall x)\Phi(x,z)\downarrow\}.$$

Since

$$\Phi(x,z)\downarrow \Leftrightarrow x \in W_z,$$

TOT is simply the set of numbers z such that W_z is the set of all nonnegative integers.

We have

Theorem 6.1. TOT is not r.e.

Proof. Suppose that TOT were r.e. Since TOT $\neq \emptyset$, by Theorem 4.9 there is a computable function $g(x)$ such that TOT = {g(O), g(l), g(2), . . . }. Let

$$h(x) = \Phi(x, g(x)) + 1.$$

Since each value g(x) is the number of a program that computes a total function, $\Phi(u,$ g(x)) $\downarrow$ for all x, u and hence, in particular, h(x) $\downarrow$ for all x. Thus h is itself a computable function. Let h be computed by program $\mathscr{P}$, and let $p = \#(\mathscr{P})$. Then $p \in$ TOT, so that $p = g(i)$ for some i. Then

$$\begin{aligned} h(i) &= \Phi(i, g(i)) + 1 && \text{by definition of } h \\ &= \Phi(i,p) + 1 && \text{since } p = g(i) \\ &= h(i) + 1 && \text{since } h \text{ is computed by } \mathscr{P}, \end{aligned}$$

which is a contradiction. ■

Note that in the proof of Theorem 6.1, the set A is TOT itself, and this time assertion 1 was taken as an assumption, while assertion 2 is shown to be true. Theorem 6.1 helps to explain why we base the study of computability on partial functions rather than total functions. By Church's thesis, Theorem 6.1 implies that there is no algorithm to determine if an $\mathscr{S}$ program computes a total function.

Once some set such as K has been shown to be nonrecursive, we can use that set to give other examples of nonrecursive sets by way of the ***reducibility*** method.

Definition. Let A, B be sets. A is ***many-one reducible to B,*** written $A \leq_m B$, if there is a computable function f such that

$$A = \{x \in N \mid f(x) \in B\}.$$

That is, $x \in A$ if and only if $f(x) \in B$. (The word ***many-one*** simply refers to the fact that we do not require f to be one-one.)

If A I, *B,* then in a sense testing membership in A is "no harder than" testing membership in B. In particular, to test $x \in$ A, we can compute $f(x)$ and then test $f(x) \in B$.

Theorem 6.2. Suppose $A \leq_m B$.

1. If B is recursive, then A is recursive.
2. If B is r.e., then A is r.e.

Proof. Let $A = \{x \in N \mid f(x) \in B\}$, where f is computable, and let $P_B(x)$ be the characteristic function of B. Then

$$A = \{x \in N \mid P_B(f(x))\},$$

and if B is recursive then $P_B(f(x))$, the characteristic function of A, is computable.

Now suppose that B is r.e. Then $B = \{x \in N \mid g(x)\downarrow\}$ for some partially computable function g, and $A = \{x \in N \mid g(f(x))\downarrow\}$. But $g(f(x)>$ is partially computable, so A is r.e. ■

We generally use Theorem 6.2 in the form: If A is not recursive (r.e.), then B is not recursive (respectively: not r.e.). For example, let

$$K_0 = \left\{x \in N \mid \Phi_{r(x)}(l(x))\downarrow\right\} = \left\{\langle x, y\rangle \mid \Phi_y(x)\downarrow\right\}.$$

K_0 is clearly r.e. However, we can show by reducing K to *K,,* that is, by showing that K I, K_0, that K_0 is not recursive: $x \in K$ if and only if $\langle x, x\rangle \in K_0$, and the function $f(x) = \langle x, x\rangle$ is computable. In fact, it is easy to show that every r.e. set is many-one reducible to *K,:* if A is r.e., then

$$\begin{aligned} A &= \{x \in N \mid g(x)\downarrow\} && \text{for some partially computable } g \\ &= \{x \in N \mid \Phi(x, z_0)\downarrow\} && \text{for some } z_0 \\ &= \{x \in N \mid \langle x, z_0\rangle \in K_0\}. \end{aligned}$$

Definition. A set ***A is m-complete if***

1. A is r.e., and
2. for every r.e. set B, B I, A.

So K_0 is m-complete. We can also show that K is m-complete. First we show that $K_0 \leq_m$ K. This argument is somewhat more involved because K_0 seems, at first glance, to contain more information than $K. K_0$ represents the halting behavior of all partially computable functions on all inputs, while K represents only the halting behavior of partially computable functions on a single argument. We wish to take a pair $\langle n, q \rangle$ and transform it to a number $f(\langle n, q \rangle)$ of a single program such that

$$\Phi_q(n)\downarrow \quad \text{if and only if} \quad \Phi_{f(\langle n,q\rangle)}(f(\langle n,q\rangle))\downarrow,$$

i.e., such that $\langle n,q\rangle \in K_0$ if and only if $f(\langle n,q\rangle) \in K$. *The* parameter theorem turns out to be very useful here. Let $\mathscr{P}$ be the program

$$Y \leftarrow \Phi^{(1)}(l(X_2), r(X_2))$$

and let $p = \#(\mathscr{P})$. Then $\psi_{\mathscr{P}}(x_1, x_2) = \Phi^{(1)}(l(x_2), r(x_2))$, and

$$\psi_{\mathscr{P}}(x_1, x_2) = \Phi^{(2)}(x_1, x_2, p) = \Phi^{(1)}(x_1, S_1^1(x_2, p))$$

by the parameter theorem, so for any pair $\langle n$, q),

$$\Phi^{(1)}(n,q) = \psi_{\mathscr{P}}(x_1, \langle n,q\rangle) = \Phi^{(1)}_{S_1^1(\langle n,q\rangle,p)}(x_1). \tag{6.1}$$

Now, (6.1) holds for all values of x_1, so, in particular,

$$\Phi^{(1)}(n,q) = \Phi^{(1)}_{S_1^1(\langle n,q\rangle,p)}(S_1^1(\langle n,q\rangle,p)),$$

and therefore

$$\Phi^{(1)}(n,q)\downarrow \quad \text{if and only if} \quad \Phi^{(1)}_{S_1^1(\langle n,q\rangle,p)}(S_1^1(\langle n,q\rangle,p))\downarrow,$$

i.e.,

$$(n,q) \in K_0 \quad \text{if and only if} \quad S_1^1(\langle n,q\rangle,p) \in K.$$

With p held constant $S_1^1(x,p)$ is a computable unary function, so K_0 *I, K.* To complete the argument that K is m-complete we need

Theorem 6.3. If ***A I, B*** and $B \leq_m$ C, then ***A I, C.***

Proof. Let $A = \{x \in N \mid f(x) \in B\}$ and $B = \{x \in N \mid g(x) \in C\}$. Then $A = \{x \in N \mid g(f(x)) \in C\}$, and $g(f(x))$ is computable. ■

As an immediate consequence we have

Corollary 6.4. If A is m-complete, B is r.e., and A I, B, then B is m-complete.

Proof. If C is r.e. then C $\leq_m$ A, and A I, B by assumption, so C I, B. ■

Thus, K is m-complete. Informally, testing membership in an m-complete set is "at least as difficult as" testing membership in any r.e. set. So an m-complete set is a good choice for showing by a reducibility argument that a given set is not computable. We expand on this subject in Chapter 8.

Actually, we have shown both K *I,* K_0 and K_0 *I, K, so* in a sense, testing membership in K and testing membership in K_0 are "equally difficult" problems.

Definition. A $\equiv_m B$ means that A $\leq_m B$ and $B \leq_m$ A.

In general, for sets A and B, if A $\equiv_m B$ then testing membership in A has the "same difficulty as" testing membership in B.

To summarize, we have proved

Theorem 6.5.

1. K and K_0 are m-complete.
2. $K \equiv_m$ *K,.*

We can also use reducibility arguments to show that certain sets are not r.e. Let

$$\text{EMPTY} = \{x \in N \mid W_x = 0).$$

Theorem 6.6. EMPTY is not r.e.

Proof. We will show that $\bar{K} \leq_m$ EMPTY. $\bar{K}$ is not r.e., so by Theorem 6.2, EMPTY is not r.e. Let $\mathscr{P}$ be the program

$$Y \leftarrow \Phi(X_2, X_2),$$

and let $p = \#(\mathscr{P})$. $\mathscr{P}$ ignores its first argument, so for a given z,

$$\psi_{\mathscr{P}}^{(2)}(x, z)\downarrow \text{ for all } x \quad \text{if and only if} \quad \Phi(z, z)\downarrow.$$

By the parameter theorem

$$\psi_{\mathscr{P}}^{(2)}(x_1, x_2) = \Phi^{(2)}(x_1, x_2, p) = \Phi^{(1)}(x_1, S_1^1(x_2, p)),$$

so, for any z,

$$\begin{aligned} z \in \bar{K} &\text{ if and only if } \Phi(z,z)\uparrow \\ &\text{ if and only if } \psi_{\mathscr{P}}^{(2)}(x,z)\uparrow \text{ for all } x \\ &\text{ if and only if } \Phi^{(1)}(x, S_1^1(z,p))\uparrow \text{ for all } x \\ &\text{ if and only if } W_{S_1^1(z,p)} = 0 \\ &\text{ if and only if } S_1^1(z,p) \in \text{ EMPTY.} \end{aligned}$$

$f(z) = S_1^1(z,p)$ is computable, so $\bar{K}$ I, EMPTY. ■

Exercises

1. Show that the proof of Theorem 4.7 is a diagonalization argument.
2. Prove by diagonalization that there is no enumeration f_0, f_1, f_2, . . . of all total unary (not necessarily computable) functions on N.
3. Let A = $\{x \in N \mid \Phi_x(x)\downarrow$ and $\Phi_x(x) >$ x$\}$.
   (a) Show that A is r.e.
   (b) Show by diagonalization that A is not recursive.
4. Show how the diagonalization argument in the proof of Theorem 6.1 fails for the set of all numbers p such that p is the number of a program that computes a partial function, i.e., the set N.
5. Let A, B be sets of numbers. Prove
   (a) **A $\leq_m$ A.**
   (b) $A \leq_m B$ if and only if $\bar{A}$ I, $\bar{B}$.
6. Prove that no m-complete set is recursive.
7. Let A, B be m-complete. Show that A $\equiv_m$ B.
8. Prove that $\bar{K} \not\leq_m$ K, i.e., $\bar{K}$ is not many-one reducible to K.
9. For every number n, let A, = $\{x \mid n \in W_x\}$.
   (a) Show that A_i is r.e. but not recursive, for all i.
   (b) Show that $A_i \equiv_m A_j$ for all i, j.
10. Define the predicate $P(x) \Leftrightarrow \Phi_x(x) = 1$. Show that $P(x)$ is not computable.
11. Define the predicate

$Q(x) \Leftrightarrow$ the variable Y assumes the value 1 sometime during the computation of $\psi_{\mathscr{P}}(x)$, where $\#(\mathscr{P})$ = x.

Show that $Q(x)$ is not computable. *[Hint:* Use the parameter theorem and a version of the universal program $\mathscr{U}_1$.]

12. Let INF = $\{x \in N \mid W_x$ is infinite$\}$. Show that INF $\equiv_m$ TOTAL.
13. Let FIN = $\{x \in N \mid W_x$ is finite$\}$. Show that $\bar{K}$ I, FIN.

14.* Let

$$\text{MONOTONE} = (y \in N \mid \Phi_y(x) \text{ is total and } \Phi_y(x) \leq \Phi_y(x+1) \text{ for all } x\}.$$

(a) Show by diagonalization that MONOTONE is not r.e.
(b) Show that MONOTONE $\equiv_m$ TOTAL.

7. Rice's Theorem

Using the reducibility method we can prove a theorem that gives us, at a single stroke, a wealth of interesting unsolvable problems concerning programs.

Let Γ be some collection of partially computable functions of one variable. We may associate with Γ the set (usually called an *index set)*

$$R_\Gamma = \{t \in N \mid \Phi_t \in \Gamma\}.$$

R_Γ is a recursive set just in case the predicate $g(t)$, defined $g(t) \Leftrightarrow \Phi_t \in \Gamma$, is computable. Consider the examples:

1. I' is the set of computable functions;
2. Γ is the set of primitive recursive functions;
3. Γ is the set of partially computable functions that are defined for all but a finite number of values of x.

These examples make it plain that it would be interesting to be able to show that R_Γ is computable for various collections Γ. Invoking Church's thesis, we can say that R_Γ is a recursive set just in case there is an algorithm that accepts *programs* $\mathscr{P}$ as input and returns the value TRUE or FALSE depending on whether or not the function $\psi_{\mathscr{P}}^{(1)}$ does or does not belong to Γ. In fact, those who work with computer programs would be very pleased to possess algorithms that accept a program as input and which return as output some useful property of the partial function computed by that program. Alas, such algorithms are not to be found! This dismal conclusion follows from Rice's theorem.

Theorem 7.1 (Rice's Theorem). Let Γ be a collection of partially computable functions of one variable. Let there be partially computable functions $f(x)$, $g(x)$ such that $f(x)$ belongs to Γ but $g(x)$ does not. Then R_Γ is not recursive.

Proof. Let $h(x)$ be the function such that $h(x)\uparrow$ for all x. We assume first that $h(x)$ does not belong to Γ. Let q be the number of

$$
\begin{aligned}
&Z \leftarrow \Phi(X_2, X_2)\\
&Y \leftarrow f(X_1)
\end{aligned}
$$

Then, for any i, $S_1^1(i,q)$ is the number of

$$
\begin{aligned}
&X_2 \leftarrow i\\
&Z \leftarrow \Phi(X_2, X_2)\\
&Y \leftarrow f(X_1)
\end{aligned}
$$

Now

$$
\begin{aligned}
i \in K \quad &\textit{implies} \quad \Phi(i,i)\downarrow\\
&\textit{implies} \quad \Phi_{S_1^1(i,q)}(x) = f(x) \text{ for all } x\\
&\textit{implies} \quad \Phi_{S_1^1(i,q)} \in \Gamma\\
&\textit{implies} \quad S_1^1(i,q) \in R_\Gamma,
\end{aligned}
$$

and

$$
\begin{aligned}
i \notin K \quad &\textit{implies} \quad \Phi(i,i)\uparrow\\
&\textit{implies} \quad \Phi_{S_1^1(i,q)}(x)\uparrow \text{ for all } x\\
&\textit{implies} \quad \Phi_{S_1^1(i,q)} = h\\
&\textit{implies} \quad \Phi_{S_1^1(i,q)} \notin \Gamma\\
&\textit{implies} \quad S_1^1(i,q) \notin R_\Gamma,
\end{aligned}
$$

so $K \leq_1 R_\Gamma$. By Theorem 6.2, R_Γ is not recursive.

If $h(x)$ does belong to Γ, then the same argument with Γ and $f(x)$ replaced by $\overline{\Gamma}$ and $g(x)$ shows that $R_{\overline{\Gamma}}$ is not recursive. But $R_{\overline{\Gamma}} = \overline{R_\Gamma}$, *so,* by Theorem 4.1, R_Γ is not recursive in this case either. ■

Corollary 7.2. There are no algorithms for testing a given program $\mathscr{P}$ of the language $\mathscr{S}$ to determine whether $\psi_{\mathscr{P}}^{(1)}(x)$ belongs to any of the classes described in Examples 1-3.

Proof. In each case we only need find the required functions $f(x)$, $g(x)$ to show that R_Γ is not recursive. The corollary then follows by Church's thesis. For 1, 2, or 3 we can take, for example, $f(x) = u_1^1(x)$ and g(x) = $1 - x$ [so that g(x) is defined only for $x = 0, 1$]. ■

Exercises

1. Show that Rice's theorem is false if the requirement for functions $f(x)$, g(x) is omitted.
2. Show there is no algorithm to determine of a given program $\mathscr{P}$ in the language $\mathscr{S}$ whether $\psi_{\mathscr{P}}(x) = x^2$ for all x.
3. Show that there is no algorithm to determine of a pair of numbers u, v whether $\Phi_u(x) = \Phi_v(x)$ for all x.
4. Show that the set A = $\{x \mid \Phi_x$ is defined for at least one input$\}$ is r.e. but not recursive.
5. Use Rice's theorem to show that the following sets are not recursive. [See Section 6 for the definitions of the sets.]
   (a) TOT;
   (b) EMPTY;
   (c) INF;
   (d) FIN;
   (e) MONOTONE;
   (f) $\{y \in \mathbf{N} \mid \Phi_y^{(1)}$ is a predicate$\}$.
6. Let Γ be a collection of partially computable functions of m variables, $m > 1$, and let $R_\Gamma^{(m)} = \{t \in N \mid \Phi_t^{(m)} \in \Gamma\}$. State and prove a version of Rice's theorem for collections of partially computable functions of m variables, $m > 1$.
7. Define the predicate

   PROPER(n) $\Leftrightarrow$ $\min_z [\Phi_n^{(2)}(x, z) = 3]$ is an application of proper minimalization to the predicate $\Phi_n^{(2)}(x, z) = 3$.

   Show that PROPER(x) is not computable.
8. Let Γ be a set of partially computable functions of one variable. Show that R_Γ is r.e. if and only if it is m-complete.

*8. The Recursion Theorem

In the proof that HALT(x, y) is not computable, we gave (assuming HALT(x, y) to be computable) a program $\mathscr{P}$ such that

$$\text{HALT}(\#(\mathscr{P}), \#(\mathscr{P})) \Leftrightarrow \sim \text{HALT}(\#(\mathscr{P}), \#(\mathscr{P})).$$

We get a contradiction when we consider the behavior of the program $\mathscr{P}$ on input $\#(\mathscr{P})$. The phenomenon of a program acting on its own description is sometimes called self-reference, and it is the source of many fundamental results in computability theory. Indeed, the whole point of diagonalization in the proof of Theorem 2.1 is to get a contradictory self-reference. We turn now to a theorem which packages, so to speak, a general technique for obtaining self-referential behavior. It is one of the most important applications of the parameter theorem.

Theorem 8.1 (Recursion Theorem). Let $g(z, x_1, \ldots, x_m)$ be a partially computable function of $m + 1$ variables. Then there is a number e such that

$$\Phi_e^{(m)}(x_1, \ldots, x_m) = g(e, x_1, \ldots, x_m).$$

Discussion. Let e = $\#(\mathscr{P})$, so that $\psi_{\mathscr{P}}^{(m)}(x_1, \ldots, x_m) = \Phi_e^{(m)}(x_1, \ldots, x_m)$. The equality in the theorem says that the m-ary function $\psi_{\mathscr{P}}^{(m)}(x_1, \ldots, x_m)$ is equal to $g(z, x_1, \ldots, x_m)$ when the first argument of g is held constant at e. That is, $\mathscr{P}$ is a program that, in effect, gets access to its own number, e, and computes the m-ary function $g(e$, xi,. $.., x_m)$. Note that since $x_1, \ldots, x_m$ can be arbitrary values, e generally does not appear among the inputs to $\psi_{\mathscr{P}}^{(m)}(x_1, \ldots, x_m)$, so $\mathscr{P}$ must somehow compute e. One might suppose that $\mathscr{P}$ might contain e copies of an instruction such as $Z \leftarrow$ Z + 1, that is, an expansion of the macro Z ← e, but if $\mathscr{P}$ has at least e instructions, then certainly $\#(\mathscr{P}) >$ e. The solution is to write $\mathscr{P}$ so that it computes e without having e "built in" to the program. In particular, we build into $\mathscr{P}$ a "partial description" of $\mathscr{P}$, and then have $\mathscr{P}$ compute e from the partial description. Let $\mathscr{Q}$ be the program

$$Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$$
$$Y \leftarrow g(Z, X_1, \ldots, X_m)$$

We prefix $\#(\mathscr{Q})$ copies of the instruction $X_{m+1} \leftarrow X_{m+1} + 1$ to get the program $\mathscr{R}$:

$$X_{m+1} \leftarrow X_{m+1} + 1$$
$$\vdots \qquad \vdots$$
$$X_{m+1} \leftarrow X_{m+1} + 1$$
$$Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$$
$$Y \leftarrow g(Z, X_1, \ldots, X_m)$$

After the first $\#(\mathscr{Q})$ instructions are executed, X_{m+1} holds the value $\#(\mathscr{Q})$, and $S_m^1(\#(\mathscr{Q}), \#(\mathscr{Q}))$, as defined in the proof of the parameter theorem, computes the number of the program consisting of $\#(\mathscr{Q})$ copies of $X_{m+1} \leftarrow X_{m+1} + 1$ followed by program $\mathscr{Q}$. ***But that program is*** $\mathscr{R}$. ***So*** $Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$ gives Z the value $\#(\mathscr{R})$, and Y $\leftarrow g(Z, X_1, \ldots, X_m)$ causes $\mathscr{R}$ to output $g(\#(\mathscr{R}), x_1, \ldots, x_m)$. We take e to be $\#(\mathscr{R})$ and we have

$$\Phi_e^{(m)}(x_1, \ldots, x_m) = \psi_{\mathscr{R}}^{(m)}(x_1, \ldots, x_m) = g(e, x_1, \ldots, x_m).$$

We now formalize this argument.

Proof. Consider the partially computable function

$$g(S_m^1(v, v), x_1, \ldots, x_m)$$

where S_m^1 is the function that occurs in the parameter theorem. Then we have for some number z_0,

$$\begin{aligned} g(S_m^1(v, v), x_1, \ldots, x_m) &= \Phi^{(m+1)}(x_1, \ldots, x_m, v, z_0) \\ &= \Phi^{(m)}(x_1, \ldots, x_m, S_m^1(v, z_0)), \end{aligned}$$

where we have used the parameter theorem. Setting v = z_0 and e = $S_m^1(z_0$, z,), we have

$$g(e, x_1, \ldots, x_m) = \Phi^{(m)}(x_1, \ldots, x_m, \mathrm{e}) = \Phi_e^{(m)}(x_1, \ldots, x_m). \qquad \blacksquare$$

We can use the recursion theorem to give another self-referential proof that HALT(x, y) is not computable. If HALT(x, y) were computable, then

$$f(x, y) \begin{cases} \uparrow & \text{if HALT}(y, x) \\ 0 & \text{otherwise} \end{cases}$$

would be partially computable, so by the recursion theorem there would be a number e such that

$$\Phi_e(y) = f(e, y) = \begin{cases} \uparrow & \text{if HALT}(y, \mathrm{e}) \\ 0 & \text{otherwise,} \end{cases}$$

that is,

$$\sim \text{HALT}(y, e) \Leftrightarrow \text{HALT}(y, e).$$

So HALT(x, y) is not computable. The self-reference occurs when Φ_e computes e, tests HALT(y, e), and then does the opposite of what HALT(y, e) says it does.

One of the many uses of the recursion theorem is to allow us to write down definitions of functions that involve the program used to compute the function as part of its definition. For a simple example we give

Corollary 8.2. There is a number e such that for all x

$$\Phi_e(x) = e.$$

Proof. We consider the computable function

$$g(z, x) = u_1^2(z, x) = z.$$

Applying the recursion theorem we obtain a number e such that

$$\Phi_e(x) = g(e, x) = e$$

and we are done. ■

It is tempting to be a little metaphorical about this result. The program with number e "consumes" its "environment" (i.e., the input x) and outputs a "copy" of itself. That is, it is, in miniature, a self-reproducing organism. This program has often been cited in considerations of the comparison between living organisms and machines.

For another example, let

$$f(x,t) = \begin{cases} k & \text{if } t = 0 \\ g(t \dot{-} 1, \Phi_x(t \dot{-} 1)) & \text{otherwise,} \end{cases}$$

where $g(x, y)$ is computable. It is clear that $f(x, t)$ is partially computable, so by the recursion theorem there is a number e such that

$$\Phi_e(t) = f(e, t) = \begin{cases} k & \text{if } t = 0 \\ g(t \dot{-} 1,\ \Phi_e(t \dot{-} 1)) & \text{otherwise.} \end{cases}$$

An easy induction argument on t shows that Φ_e is a total, and therefore computable, function. Now, Φ_e satisfies the equations

$$\Phi_e(0) = k$$

$$\Phi_e(t + 1) = g(t, \Phi_e(t)),$$

that is, Φ_e is obtained from g by primitive recursion of the form (2.1) in Chapter 3, so the recursion theorem gives us another proof of Theorem 2.1 in Chapter 3. In fact, the recursion theorem can be used to justify definitions based on much more general forms of recursion, which explains how it came by its name.[1] We give one more example, in which we wish to

[1] For more on this subject, see Part 5.

know if there are partially computable functions f, g that satisfy the equations

$$\begin{aligned} f(0) &= 1 \\ f(t+1) &= g(2t) + 1 \\ g(0) &= 3 \\ g(2t+2) &= f(t) + 2. \end{aligned} \tag{8.1}$$

Let $F(z,t)$ be the partially computable function

$$F(z,x) = \begin{cases} 1 & \text{if } x = \langle 0,0\rangle \\ \Phi_z(\langle 1, 2(r(x) \dot{-} 1)\rangle) + 1 & \text{if } (\exists y)_{\le x}(x = \langle 0, y+1\rangle) \\ 3 & \text{if } x = \langle 1,0\rangle \\ \Phi_z(\langle 0, \lfloor (r(x) \dot{-} 2)/2 \rfloor\rangle) + 2 & \text{if } (\exists y)_{\le x}(x = \langle 1, 2y+2\rangle). \end{cases}$$

By the recursion theorem there is a number e such that

$$\begin{aligned} \Phi_e(x) &= F(e,x) \\ &= \begin{cases} 1 & \text{if } x = \langle 0,0\rangle \\ \Phi_e(\langle 1, 2(r(x) \dot{-} 1)\rangle) + 1 & \text{if } (\exists y)_{\le x}(x = \langle 0, y+1\rangle) \\ 3 & \text{if } x = (1,0) \\ \Phi_e(\langle 0, \lfloor (r(x) \dot{-} 2)/2 \rfloor\rangle) + 2 & \text{if } (\exists y)_{\le x}(x = \langle 1, 2y+2\rangle). \end{cases} \end{aligned}$$

Now, setting

$$f(x) = \Phi_e(\langle 0, x\rangle) \quad \text{and} \quad g(x) = \Phi_e(\langle 1, x\rangle)$$

we have

$$\begin{aligned} f(0) &= \Phi_e(\langle 0,0\rangle) = 1 \\ f(t+1) &= \Phi_e(\langle 0, t+1)) = \Phi_e(\langle 1, 2t\rangle) + 1 = g(2t) + 1 \\ g(0) &= \Phi_e(\langle 1,0\rangle) = 3 \\ g(2t+2) &= \Phi_e(\langle 1, 2t+2)) = \Phi_e(\langle 0, t\rangle) + 2 = f(t) + 2, \end{aligned}$$

so f, g satisfy (8.1).

Another application of the recursion theorem is

Theorem 8.3 (Fixed Point Theorem). Let $f(z)$ be a computable function. Then there is a number e such that

$$\Phi_{f(e)}(x) = \Phi_e(x)$$

for all x.

Proof. Let $g(z, x) = \Phi_{f(z)}(x)$, a partially computable function. By the recursion theorem, there is a number e such that

$$\Phi_e(x) = g(e, x) = \Phi_{f(e)}(x). \qquad \blacksquare$$

Usually a number n is considered to be a fixed point of a function $f(x)$ if $f(n) = n$. Clearly there are computable functions that have no fixed point in this sense, e.g., $s(x)$. The fixed point theorem says that for every computable function $f(x)$, there is a number e of a program that ***computes the same function as*** the program with number $f(e)$.

For example, let $P(x)$ be a computable predicate, let g(x) be a computable function, and let while(n) = $\#(\mathscr{Q}_n)$, where $\mathscr{Q}_n$ is the program

```
      $X_2 \leftarrow n$
      $Y \leftarrow X$
[A]   IF $\sim P(Y)$ GOTO $E$
      $Y \leftarrow \Phi_{X_2}(g(Y))$
```

It should be clear that while(x) is a computable, in fact primitive recursive, function, so by the fixed point theorem there is a number e such that

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x).$$

It follows from the construction of while(e) that

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ \Phi_e(g(x)) & \text{otherwise.} \end{cases}$$

Moreover,

$$\Phi_e(g(x)) = \Phi_{\text{while}(e)}(g(x)) = \begin{cases} g(x) & \text{if } \sim P(g(x)) \\ \Phi_e(g(g(x))) & \text{otherwise,} \end{cases}$$

so

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ g(x) & \text{if } P(x) \,\&\, \sim P(g(x)) \\ \Phi_e(g(g(x))) & \text{otherwise,} \end{cases}$$

and continuing in this fashion we get

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ g(x) & \text{if } P(x) \,\&\, \sim P(g(x)) \\ g(g(x)) & \text{if } P(x) \,\&\, P(g(x)) \,\&\, \sim P(g(g(x))) \\ \vdots & \vdots \end{cases}$$

In other words, program e behaves like the pseudo-program

```
$Y \leftarrow X$
WHILE $P(Y)$ DO
    $Y \leftarrow g(Y)$
END
```

We end this discussion of the recursion theorem by giving another proof of Rice's theorem. Let Γ, f(x), g(x) be as in the statement of Theorem 7.1.

Alternative Proof of Rice's Theorem.[2] Suppose that R_Γ were computable. Let

$$P_\Gamma(t) = \begin{cases} 1 & \text{if } t \in R_\Gamma \\ 0 & \text{otherwise.} \end{cases}$$

That is, P_Γ is the characteristic function of R_Γ. Let

$$h(t,x) = \begin{cases} g(x) & \text{if } t \in R_\Gamma \\ f(x) & \text{otherwise.} \end{cases}$$

Then, since (as in the proof of Theorem 5.4, Chapter 3)

$$h(t,x) = g(x) \cdot P_\Gamma(t) + f(x) \cdot \alpha(P_\Gamma(t)),$$

$h(t,x)$ is partially computable. Thus, by the recursion theorem, there is a number e such that

$$\Phi_e(x) = h(e,x) = \begin{cases} g(x) & \text{if } \Phi_e \text{ belongs to } \Gamma \\ f(x) & \text{otherwise.} \end{cases}$$

[2] This elegant proof was called to our attention by John Case.

Does e belong to R_Γ? Recalling that $f(x)$ belongs to Γ but $g(x)$ does not, we have

$$\begin{aligned} e \in R_\Gamma \quad & \textit{implies} \quad \Phi_e(x) = g(x) \\ & \textit{implies} \quad \Phi_e \text{ is not in } \Gamma \\ & \textit{implies} \quad e \notin R_\Gamma. \end{aligned}$$

But likewise,

$$\begin{aligned} e \notin R_\Gamma \quad & \textit{implies} \quad \Phi_e(x) = f(x) \\ & \textit{implies} \quad \Phi_e \text{ is in } \Gamma \\ & \textit{implies} \quad e \in R_\Gamma. \end{aligned}$$

This contradiction proves the theorem. ■

Exercises

1. Use the proof of Corollary 8.2 and the discussion preceding the proof of the recursion theorem to write a program $\mathscr{P}$ such that $\psi_{\mathscr{P}}(x) = \#(\mathscr{P})$.
2. Let $A = \{x \in N \mid \Phi_x(x)\downarrow \text{ and } \Phi_x(x) > x\}$. Use the recursion theorem to show that A is not recursive.
3. Show that there is a number e such that $W_e = \{e\}$.
4. Show that there is a program $\mathscr{P}$ such that $\psi_{\mathscr{P}}(x)\downarrow$ if and only if $x = \#(\mathscr{P})$.
5. (a) Show that there is a partially computable function f that satisfies the equations

$$\begin{aligned} f(x,0) &= x + 2 \\ f(x,1) &= 2 \cdot f(x,2x) \\ f(x,2t+2) &= 3 \cdot f(x,2t) \\ f(x,2t+3) &= 4 \cdot f(x,2t+1). \end{aligned}$$

   What is $f(2,5)$?

   (b) Prove that f is total.

   (c) Prove that f is unique. (That is, only one function satisfies the given equations.)
6. Give two distinct partially computable functions f, g that satisfy the equations

$$\begin{aligned} f(0) &= 2 & g(0) &= 2 \\ f(2t+2) &= 3 \cdot f(2t) & g(2t+2) &= 3 \cdot g(2t). \end{aligned}$$

   For the specific functions f, g that you give, what are $f(1)$ and $g(1)$?

7. Let $f(x) = x + 1$. Use the proof of the fixed point theorem and the discussion preceding the proof of the recursion theorem to give a program $\mathscr{P}$ such that $\Phi_{\#(\mathscr{P})}(x) = \Phi_{f(\#(\mathscr{P}))}(x)$. What unary function does $\mathscr{P}$ compute?

8. Give a function f(y) such that, for all y, f(y) > y and $\Phi_y(x) = \Phi_{f(y)}(x)$.

9. Give a function f(y) such that, for all y, if $\Phi_y(x) = \Phi_{f(y)}(x)$, then $\Phi_y(x)$ is not total.

10. Show that the function while(x) defined following the fixed point theorem is primitive recursive. [*Hint:* Use the parameter theorem.]

11. **(a)** Prove that the recursion theorem can be strengthened to read: There are infinitely many numbers e such that

$$\Phi_e^{(m)}(x_1, \ldots, x_m) = g(e, x_1, \ldots, x_m).$$

(b) Prove that the fixed point theorem can be strengthened to read: There are infinitely many numbers e such that

$$\Phi_{f(e)}(x) = \Phi_e(x).$$

12. Prove the following version of the recursion theorem: There is a primitive recursive function self(x) such that for all z

$$\Phi_{\text{self}(z)}(x) = \Phi_z^{(2)}(\text{self}(z), x).$$

13. Prove the following version of the fixed point theorem: There is a primitive recursive function $\text{fix}(u)$ such that for all x, u,

$$\Phi_{\text{fix}(u)}(x) = \Phi_{\Phi_u(\text{fix}(u))}(x).$$

14.* Let S be an acceptable programming system with universal functions $\Psi^{(m)}$. Prove the following: For every partially computable function $g(z, x_1, \ldots, x_m)$ there is a number e such that

$$\Psi_e^{(m)}(x_1, \ldots, x_m) = g(e, x_1, \ldots, x_m).$$

That is, a version of the recursion theorem holds for S. [See Exercise 5.4 for the definition of acceptable programming systems.]

*9. A Computable Function That Is Not Primitive Recursive

In Chapter 3 we showed that all primitive recursive functions are computable, but we did not settle the question of whether all computable functions are primitive recursive. We shall deal with this matter by showing how to obtain a function h(x) that is computable but is not primitive recursive. Our method will be to construct a computable function $\phi(t,x)$ that enumerates all of the unary primitive recursive functions. That is, it will be the case that

1. for each fixed value $t = t_0$, the function $\phi(t_0, x)$ will be primitive recursive;
2. for each unary primitive recursive function $f(x)$, there will be a number t_0 such that $f(x) = \phi(t_0, x)$.

Once we have this function ϕ at our disposal, we can diagonalize, obtaining the unary computable function $\phi(x, x) + 1$ which must be different from all primitive recursive functions. (If it were primitive recursive, we would have

$$\phi(x,x) + 1 = \phi(t_0, x)$$

for some fixed t_0, and setting $x = t_0$ would lead to a contradiction.)

We will obtain our enumerating function by giving a new characterization of the unary primitive recursive functions. However, we begin by showing how to reduce the number of parameters needed in the operation of primitive recursion which, as defined in Chapter 3 (Eq. (2.2)), proceeds from the total n-ary function f and the total $n + 2$-ary function g to yield the $n + 1$-ary function h such that

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

If $n > 1$ we can reduce the number of parameters needed from n to $n - 1$ by using the pairing functions. That is, let

$$\tilde{f}(x_1, \ldots . x_{n-1}) = f(x_1, \ldots, x_{n-2}, l(x_{n-1}), r(x_{n-1})),$$
$$\tilde{g}(t, u, x_1, \ldots, x_{n-1}) = g(t, u, x_1, \ldots, x_{n-2}, l(x_{n-1}), r(x_{n-1})),$$
$$\tilde{h}(x_1, \ldots, x_{n-1}, t) = h(x_1, \ldots, x_{n-2}, l(x_{n-1}), r(x_{n-1}), t).$$

Then, we have

$$\tilde{h}(x_1, \ldots, x_{n-1}, 0) = \tilde{f}(x_1, \ldots, x_{n-1})$$
$$\tilde{h}(x_1, \ldots, x_{n-1}, t+1) = \tilde{g}\big(t, \tilde{h}(x_1, \ldots, x_{n-1}, t), x_1, \ldots, x_{n-1}\big).$$

Finally, we can retrieve the original function h from the equation

$$h(x_1, \ldots, x_n, t) = \tilde{h}(x_1, \ldots, x_{n-2}, \langle x_{n-1}, x_n \rangle, t).$$

By iterating this process we can reduce the number of parameters to 1, that is, to recursions of the form

$$\begin{aligned} h(x,0) &= f(x) \\ h(x,t+1) &= g(t, h(x,t), x) \end{aligned} \tag{9.1}$$

Recursions with no parameters, as in Eq. (2.1) in Chapter 3, can also readily be put into the form (9.1). Namely, to deal with

$$\begin{aligned} \psi(0) &= k \\ \psi(t+1) &= \theta(t, \psi(t)), \end{aligned}$$

we set $f(x) = k$ (which can be obtained by k compositions with $s(x)$ beginning with $n(x)$) and

$$g(x_1, x_2, x_3) = \theta(u_1^3(x_1, x_2, x_3),\ u_2^3(x_1, x_2, x_3))$$

in the recursion (9.1). Then, $\psi(t) = h(x,t)$ for all x. In particular, $\psi(t) = h(u_1^1(t), u_1^1(t))$.

We can simplify recursions of the form (9.1) even further by using the pairing functions to combine arguments. Namely, we set

$$\tilde{h}(x,t) = \langle h(x,t), \langle x,t\rangle\rangle.$$

Then, we have

$$\begin{aligned} \tilde{h}(x,0) &= \langle f(x), \langle x,0\rangle\rangle \\ \tilde{h}(x,t+1) &= \langle h(x,t+1), \langle x, t+1\rangle\rangle \\ &= \langle g(t, h(x,t), x), \langle x, t+1\rangle\rangle \\ &= \tilde{g}(\tilde{h}(x,t)), \end{aligned}$$

where

$$\tilde{g}(u) = \langle g(r(r(u)), l(u), l(r(u)), \langle l(r(u)), r(r(u)) + 1\rangle\rangle.$$

Once again, the original function h can be retrieved from $\tilde{h}$; we can use the equation

$$h(x,t) = l(\tilde{h}(x,t)).$$

Now this reduction in the complexity of recursions was only possible using the pairing functions. Nevertheless, we can use it to get a simplified characterization of the class of primitive recursive functions by adding the pairing functions to our initial functions. We may state the result as a theorem.

Theorem 9.1. The primitive recursive functions are precisely the functions obtainable from the initial functions

$$s(x),\ n(x),\ Z(z),\ r(z),\ \langle x, y\rangle \quad \text{and} \quad u_i^n,\quad 1 \le i \le n$$

using the operations of composition and primitive recursion of the particular form

$$h(x, 0) = f(x)$$

$$h(x, t+1) = g(h(x,t)).$$

The promised characterization of the unary primitive recursive functions is as follows.

Theorem 9.2. The unary primitive recursive functions are precisely those obtained from the initial functions $s(x) = x + 1$, $n(x) = 0$, $E(x)$, $r(x)$ by applying the following three operations on unary functions:

1. to go from f(x) and g(x) to $f(g(x))$;
2. to go from $f(x)$ and g(x) to $\langle f(x), g(x)\rangle$;
3. to go from $f(x)$ and g(x) to the function defined by the recursion

$$h(0) = 0$$

$$h(t+1) = \begin{cases} f\left(\dfrac{t}{2}\right) & \text{if } t+1 \text{ is odd,} \\ g\left(h\left(\dfrac{t+1}{2}\right)\right) & \text{if } t+1 \text{ is even.} \end{cases}$$

Proof. Let us write **PR** for the set of all functions obtained from the initial functions listed in the theorem using operations 1 through 3. We will show that **PR** is precisely the set of unary primitive recursive functions.

To see that all the functions in **PR** are primitive recursive, it is necessary only to consider operation **3.** That is, we need to show that if f and g are primitive recursive, and h is obtained using operation 3, then h is also primitive recursive. What is different about operation 3 is that $h(t+1)$ is computed, not from $h(t)$ but rather from $h(t/2)$ or $h((t+1)/2)$, depending on whether t is even or odd. To deal with this we make use of Gödel numbering, setting

$$\tilde{h}(0) = 0,$$

$$\tilde{h}(n) = [h(0), \ldots, h(n-1)] \text{ if } n > 0.$$

We will show that $\tilde{h}$ is primitive recursive and then conclude that the same is true of h by using the equation[3]

$$h(n) = (\tilde{h}(n+1))_{n+1}.$$

Then (recalling that p_n is the nth prime number) we have

$$\tilde{h}(n+1) = \tilde{h}(n) \cdot p_{n+1}^{h(n)}$$

$$= \begin{cases} \tilde{h}(n) \cdot p_{n+1}^{f(\lfloor n/2 \rfloor)} & \text{if } n \text{ is odd,} \\ \text{i;(n)} \cdot p_{n+1}^{g((\tilde{h}(n))_{\lfloor n/2 \rfloor})} & \text{if } n \text{ is even.} \end{cases}$$

Here, we have used $\lfloor n/2 \rfloor$ because it gives the correct value whether n is even or odd and because we know from Chapter 3 that it is primitive recursive.

Next we will show that every unary primitive recursive function belongs to **PR.** For this purpose we will call a function $g(x_1, \ldots, x_n)$ *satisfactory* if it has the property that for any unary functions $h_1(t), \ldots, h_n(t)$ that belong to PR, the function $g(h_1(t), \ldots, h_n(t))$ also belongs to PR. Note that a *unary* function $g(t)$ that is satisfactory must belong to PR because $g(t) = g(u_1^1(t))$ and $u_1^1(t) = \langle l(t), r(t) \rangle$ belongs to PR. Thus, we can obtain our desired result by proving that all primitive recursive functions are satisfactory.[4]

We shall use the characterization of the primitive recursive functions of Theorem 9.1. Among the initial functions, we need consider only the pairing function $(x_1, x_2 \rangle$ and the projection functions u_i^n where $1 \le i \le n$. If $h_1(t)$ and $h_2(t)$ are in PR, then using operation 2 in the definition of PR, we see that $\langle h_1(t), h_2(t) \rangle$ is also in PR. Hence, $\langle x_1, x_2 \rangle$ is satisfactory. And evidently, if $h_1(t), \ldots,$ *h,(t)* belong to PR, then $u_i^n(h_1(t), \ldots,$ *h,(t)),* which is simply equal to $h_i(t)$, certainly belongs to PR, so u_i^n is satisfactory.

To deal with composition, let

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

where $g_1, \ldots, g_k$ and f are satisfactory. Let *h,(t), . . . , h,(t)* be given functions that belong to PR. Then, setting

$$\tilde{g}_i(t) = g_i(h_1(t), \ldots, h_n(t))$$

[3] This is a general technique for dealing with recursive definitions for a given value in terms of smaller values, so-called ***course-of-value recursions. See*** Exercise 8.5 in Chapter 3.

[4] This is an example of what was called an induction ***loading device*** in Chapter 1.

for $1 \le i \le k$ we see that each $\tilde{g}_i$ belongs to **PR.** Hence

$$h(h_1(t), \ldots, h_n(t)) = f(\tilde{g}_1(t), \ldots, \tilde{g}_n(t))$$

belongs to **PR,** and so, h is satisfactory.

Finally, let

$$h(x, 0) = f(x)$$
$$h(x, t + 1) = g(h(x, t))$$

where f and g are satisfactory. Let $\psi(0) = 0$ and let $\psi(t + 1) = h(r(t), Z(t))$. Recalling that

$$\langle a, b\rangle = 2^a(2b + 1) - 1,$$

we consider two cases according to whether $t + 1 = 2^a(2b + 1)$ is even or odd. If $t + 1$ is even, then $a > 0$ and

$$\begin{aligned}\psi(t + 1) &= h(b, a)\\ &= g(h(b, a - 1))\\ &= g(\psi(2^{a-1}(2b + 1)))\\ &= g(\psi((t + 1)/2)).\end{aligned}$$

On the other hand, if $t + 1$ is odd, then $a = 0$ and

$$\begin{aligned}\psi(t + 1) &= h(b, 0)\\ &= f(b)\\ &= f(t/2).\end{aligned}$$

In other words,

$$\psi(0) = 0$$
$$\psi(t + 1) = \begin{cases} f\left(\dfrac{t}{2}\right) & \text{if } t + 1 \text{ is odd,}\\ g\left(\psi\left(\dfrac{t + 1}{2}\right)\right) & \text{if } t + 1 \text{ is even.}\end{cases}$$

Now f and g are satisfactory, and, being unary, they are therefore in **PR.** Since ψ is obtained from f and g using operation 3, ψ also belongs to **PR.** To retrieve h from ψ we can use $h(x, y) = \psi(\langle x, y\rangle + 1)$. So,

$$h(h_1(t), h_2(t)) = \psi(s(\langle h_1(t), h_2(t)\rangle))$$

from which we see that if h_1 and h_2 both belong to PR, then so does $h(h_1(t), h_2(t))$. Hence h is satisfactory. ■

Now we are ready to define the function $\phi(t, x)$, which we shall also write as $\phi_t(x)$, that will enumerate the unary primitive recursive functions:

$$\phi_t(x) = \begin{cases} x + 1 & \text{if } t = 0 \\ 0 & \text{if } t = 1 \\ l(x) & \text{if } t = 2 \\ r(x) & \text{if } t = 3 \\ \phi_{l(n)}(\phi_{r(n)}(x)) & \text{if } t = 3n + 1, n > 0 \\ \langle \phi_{l(n)}(x), \phi_{r(n)}(x) \rangle & \text{if } t = 3n + 2, n > 0 \\ 0 & \text{if } t = 3n + 3, n > 0 \text{ and } x = 0 \\ \phi_{l(n)}((x - 1)/2) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is odd} \\ \phi_{r(n)}(\phi_t(x/2)) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is even} \end{cases}$$

Here $\phi_0(x)$, $\phi_1(x)$, $\phi_2(x)$, $\phi_3(x)$ are the four initial functions. For $t > 3$, t is represented as $3n + i$ where $n > 0$ and $i = 1$, 2 or 3; the three operations of Theorem 9.2 are then dealt with for values of t with the corresponding value of i. The pairing functions are used to guarantee all functions obtained for any value of t are eventually used in applying each of the operations. It should be clear from the definition that $\phi(t, x)$ is a total function and that it does enumerate all the unary primitive recursive functions. Although it is pretty clear that the definition provides an algorithm for computing the values of ϕ for any given inputs, for a rigorous proof more is needed. Fortunately, the recursion theorem makes it easy to provide such a proof. Namely, we set

$$g(z, t, x)$$

$$= \begin{cases} x + 1 & \text{if } t = 0 \\ 0 & \text{if } t = 1 \\ l(x) & \text{if } t = 2 \\ r(x) & \text{if } t = 3 \\ \Phi_z^{(2)}(l(n), \Phi_z^{(2)}(r(n), x)) & \text{if } t = 3n + 1, n > 0 \\ \langle \Phi_z^{(2)}(l(n), x), \Phi_z^{(2)}(r(n), x) \rangle & \text{if } t = 3n + 2, n > 0 \\ 0 & \text{if } t = 3n + 3, n > 0 \text{ and } x = 0 \\ \Phi_z^{(2)}(l(n), \lfloor x/2 \rfloor) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is odd} \\ \Phi_z^{(2)}(r(n), \Phi_z^{(2)}(t, \lfloor x/2 \rfloor)) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is even} \end{cases}$$

Then, $g(z,t,x)$ is partially computable, and by the recursion theorem, there is a number e such that

$$g(e,t,x) = \Phi_e^{(2)}(t,x).$$

Then, since $g(e,t,x)$ satisfies the definition of $\phi(t,x)$ and that definition determines ϕ uniquely as a total function, we must have

$$\phi(t,x) = g(e,t,x),$$

so that ϕ is computable.

The discussion at the beginning of this section now applies and we have our desired result.

Theorem 9.3. The function $\phi(x,x) + 1$ is a computable function that is not primitive recursive.

Exercises

1. Show that $\phi(t,x)$ is not primitive recursive.
2. Give a direct proof that $\phi(t,x)$ is computable by showing how to obtain an $\mathscr{S}$ program that computes ϕ. [*Hint:* Use the pairing functions to construct a stack for handling recursions.]