

Computability, Complexity, and Languages

**Fundamentals of Theoretical
Computer Science**

**Martin D. Davis
Elaine J. Weyuker**

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



1983

ACADEMIC PRESS

A Subsidiary of Harcourt Brace Jovanovich, Publishers

New York London

Paris San Diego San Francisco São Paulo Sydney Tokyo Toronto

**COPYRIGHT © 1983, BY ACADEMIC PRESS, INC.
ALL RIGHTS RESERVED.**

**NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR
TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC
OR MECHANICAL, INCLUDING PHOTOCOPY, RECORDING, OR ANY
INFORMATION STORAGE AND RETRIEVAL SYSTEM, WITHOUT
PERMISSION IN WRITING FROM THE PUBLISHER.**

**ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003**

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1 7DX

Library of Congress Cataloging in Publication Data

Davis, Martin, Date

Computability, complexity, and languages.

(Computer science and applied mathematics)

Includes index.

1. Machine theory. 2. Computational complexity.

3. Formal languages. I. Weyuker, Elaine J.

II. Title. III. Series.

QA267.D38 1983 001.64'01 83-2727

ISBN 0-12-206380-5

PRINTED IN THE UNITED STATES OF AMERICA

83 84 85 86 9 8 7 6 5 4 3 2 1

To

Helen Davis, Marx Weyuker, Sylvia Weyuker

Virginia Davis, Thomas Ostrand

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.

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, and Worcester Polytechnic Institute.

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–Schützenberger representation theorem for context-free languages is 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 three chapters of Part 4 introduces an important theory of computational complexity, concluding with the theory of **NP**-completeness. Part 5 contains an introduction to advanced recursion theory, includes a number of topics that have had fruitful analogs in the theory of polynomial-time computability, and concludes with an introduction to priority constructions for recursively enumerable Turing degrees. The anomalies revealed by these constructions must be taken into account in efforts to understand the underlying nature of algorithms, even though there is no reason to believe that the specific algorithms generated will prove useful in practice.

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 8 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, 8, and 9. A semester or quarter course on *logic for computer scientists* could be based on selections from Parts 1 and 3. Many other arrangements and courses are possible, as should be apparent from the dependency graph, which follows. 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

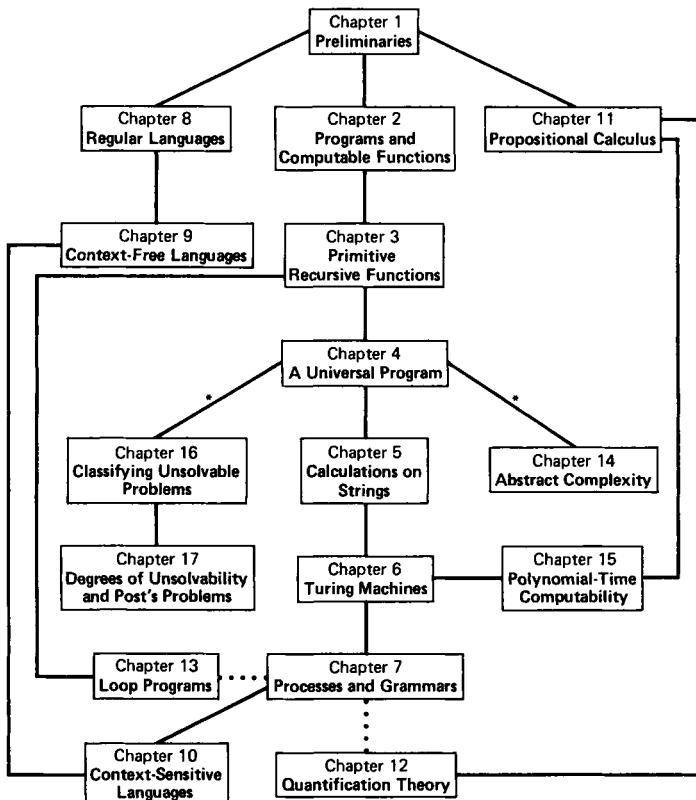
Many readers will wish to begin with Chapter 2, using the material of Chapter 1 for reference as required. Readers who enjoy skipping around will find the *dependency graph* useful.

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.

Dependency Graph



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.

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* $0, 1, 2, 3, \dots$. 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 \emptyset , 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 , $\emptyset \subseteq R$ and $R \subseteq R$. We write $R \subset S$ to indicate that $R \subseteq S$ but $R \neq S$. In this case R is called a *proper subset* of S . If R and S are sets, we write $R \cup S$ for the *union* of R and S , that is the collection of all objects which are members of either R or S or both. $R \cap S$, the *intersection* of R and S , is the set of all objects which belong to both R and S . $R - S$, the set of all objects which belong to R and do not belong to S , is the *difference* between R and S . S may contain objects not in R . Thus $R - S = R - (R \cap S)$. 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

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

for the set consisting of the n objects a_1, a_2, \dots, a_n . Sets which can be written in this form as well as the empty set are called *finite*. Sets which 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:

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

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

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

if and only if

$$a_1 = b_1, \quad a_2 = b_2, \quad \dots, \quad \text{and} \quad a_n = b_n.$$

If S_1, S_2, \dots, S_n are given sets, then we write $S_1 \times S_2 \times \dots \times S_n$ for the set of all n -tuples (a_1, a_2, \dots, a_n) such that $a_1 \in S_1, a_2 \in S_2, \dots, a_n \in S_n$. $S_1 \times S_2 \times \dots \times S_n$ is sometimes called the Cartesian product of S_1, S_2, \dots, S_n . In case $S_1 = S_2 = \dots = S_n = S$ we write S^n for the Cartesian product $S_1 \times S_2 \times \dots \times S_n$.

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 which has the special property

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

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* which 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 which 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 $f(a_1, \dots, a_n)$ rather than $f((a_1, \dots, a_n))$. 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 $f(x_1, \dots, x_n)$ instead of f as a way of showing explicitly that f is n -ary.

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_1, a_2, \dots, a_n) we write simply $a_1 a_2 \cdots a_n$. If $u = a_1 a_2 \cdots a_n$, 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

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

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

$$u0 = 0u = u,$$

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 $a_1, \dots, a_n \in A$, 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

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

where TRUE and FALSE are a pair of distinct objects called *truth values*. We often say $P(a)$ is true for $P(a) = \text{TRUE}$, and $P(a)$ is false for $P(a) = \text{FALSE}$. For our purposes it is useful to identify the truth values with specific numbers, so we set

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

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.

Table 4.1

p	$\sim p$	p	q	$p \& q$	$p \vee q$
0	1	1	1	1	1
1	0	0	1	0	1
		1	0	0	1
		0	0	0	0

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 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

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

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,

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

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

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

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

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

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

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

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

We write

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

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

The following generalized De Morgan identities are sometimes useful:

$$\begin{aligned}\sim(\exists t)_{\leq y} P(t, x_1, \dots, x_n) &\Leftrightarrow (\forall t)_{\leq y} \sim P(t, x_1, \dots, x_n), \\ \sim(\exists t)P(t, x_1, \dots, x_n) &\Leftrightarrow (\forall t)\sim P(t, x_1, \dots, x_n).\end{aligned}$$

The reader may easily verify the following examples:

$$\begin{aligned}(\exists y)(x + y = 4) &\Leftrightarrow x \leq 4, \\ (\exists y)(x + y = 4) &\Leftrightarrow (\exists y)_{\leq 4}(x + y = 4), \\ (\forall y)(xy = 0) &\Leftrightarrow x = 0, \\ (\exists y)_{\leq z}(x + y = 4) &\Leftrightarrow (x + z \geq 4 \ \& \ x \leq 4).\end{aligned}$$

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 $n \leq 40$. 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 11 and 12 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

$$2 = (m/n)^2 \quad (6.1)$$

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

$$(\forall n)(\text{If } P(n) \text{ then } P(n + 1)). \quad (7.1)$$

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

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

$$\begin{aligned} &\text{If } P(0) \text{ then } P(1), \\ &\text{If } P(1) \text{ then } P(2), \\ &\text{If } P(2) \text{ then } P(3), \dots \end{aligned}$$

Since we have proved $P(0)$, we can infer $P(1)$. Having now proved $P(1)$ 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

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

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 proved 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 which 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

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

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

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

But the “induction hypothesis” $(\forall m)_{m < 0} P(m)$ is entirely vacuous because there are 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) which 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:

$$P(0),$$

$$\text{If } P(0) \text{ then } P(1),$$

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

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

⋮

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:

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

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

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 which 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.

Programs and Computable Functions

1. A Programming Language

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

$$X_1 \ X_2 \ X_3 \ \dots$$

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

$$Z_1 \ Z_2 \ Z_3 \ \dots$$

will be called the *local variables* of \mathcal{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, \mathcal{S} must be regarded as a purely theoretical entity. Nevertheless, readers having programming experience will find working with \mathcal{S} very easy.

In \mathcal{S} we will be able to write “instructions” of various sorts; a “program” of \mathcal{S} will then consist of a *list* (i.e., a finite sequence) of instructions. For example, for each variable V there will be an instruction:

$$V \leftarrow V + 1$$

A simple example of a program of \mathcal{S} is

$$\begin{aligned} X &\leftarrow X + 1 \\ X &\leftarrow X + 1 \end{aligned}$$

“Execution” of this program has the effect of increasing the value of X by 2. In addition to variables we will need “labels.” In \mathcal{S} these are

$$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ C_2 \ D_2 \ E_2 \ A_3 \ \dots$$

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.

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

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,

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

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

2. Some Examples of Programs

(a) Our first example is the program

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

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 $X \neq 0$ 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 above program is a perfectly well-defined program of our language \mathcal{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 <i>B</i> $Z \leftarrow Z + 1$ IF $Z \neq 0$ GOTO <i>E</i>
[B]	$X \leftarrow X - 1$ $Y \leftarrow Y + 1$ $Z \leftarrow Z + 1$ IF $Z \neq 0$ GOTO <i>A</i>

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

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

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 $Z \neq 0$ 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 \mathcal{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 GOTO L several times to shorten the program. Of course, if challenged, we could produce a legal program of \mathcal{S} by replacing each GOTO 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 $X \neq 0$ GOTO B
	GOTO C
[B]	$X \leftarrow X - 1$
	$Y \leftarrow Y + 1$
	$Z \leftarrow Z + 1$
	GOTO A
[C]	IF $Z \neq 0$ GOTO D
	GOTO E
[D]	$Z \leftarrow Z - 1$
	$X \leftarrow X + 1$
	GOTO 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 ← X1
Z ← X2
[B]   IF Z ≠ 0 GOTO A
      GOTO E
[A]   Z ← Z - 1
      Y ← Y + 1
      GOTO 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”

```

Z2 ← X2
[B]   IF Z2 ≠ 0 GOTO A
      GOTO E
[A]   Z2 ← Z2 - 1
      Z1 ← X1 + Y
      Y ← Z1
      GOTO B

```

Of course, the “instruction” $Z_1 \leftarrow X_1 + Y$ is not permitted in the language \mathcal{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

```

Z1 ← X1 + Y
Y ← Z1

```

was used in this program rather than the single instruction

```

Y ← X1 + 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:

```

 $Y \leftarrow X_1$ 
 $Z \leftarrow Y$ 
[B] IF  $Z \neq 0$  GOTO A
    GOTO E
[A]   $Z \leftarrow Z - 1$ 
     $Y \leftarrow Y + 1$ 
    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 $Z_1 \leftarrow X_1 + Y$:

[B] $Z_2 \leftarrow X_2$ [A] $Z_2 \leftarrow Z_2 - 1$ [B ₂] $Z_3 \leftarrow Y$ [A ₂] $Z_3 \leftarrow Z_3 - 1$ [E ₂] $Z \leftarrow Z_1$	$Z_1 \leftarrow X_1$ IF $Z_2 \neq 0$ GOTO A GOTO E $Z_1 \leftarrow X_1$ IF $Z_3 \neq 0$ GOTO A ₂ GOTO E ₂ $Z_1 \leftarrow Z_1 + 1$ GOTO B ₂ GOTO B	Macro Expansion of $Z_1 \leftarrow X_1 + Y$
--	---	--

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_2 , B_2 , E_2 instead.
- (3) The instruction GOTO 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

```

 $Y \leftarrow X_1$ 
 $Z \leftarrow X_2$ 
[C] IF  $Z \neq 0$  GOTO A
      GOTO E
[A] IF  $Y \neq 0$  GOTO B
      GOTO A
[B]  $Y \leftarrow Y - 1$ 
       $Z \leftarrow Z - 1$ 
      GOTO 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  $Y \neq 0$  GOTO B
      GOTO 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 \mathcal{S} (using macros freely) which computes the function $f(x) = 3x$.
2. Write a program in \mathcal{S} which 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 \mathcal{S} which computes f .
4. Let $f(x) = 1$ if x is even; $f(x)$ undefined if x is odd. Write a program in \mathcal{S} which computes f .

3. Syntax

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

The symbols

$$X_1 \ X_2 \ X_3 \ \dots$$

are called *input variables*,

$$Z_1 \ Z_2 \ Z_3 \ \dots$$

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

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

are called *labels* of S . (As already indicated, in practice the subscript 1 is often omitted.) A *statement* is one of the following:

```

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

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

Note that we have included among the statements of \mathcal{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.¹ 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* \mathcal{P} is a list of equations of the form $V = m$, where V is a variable and m is a number, including exactly one equation for each variable that occurs in \mathcal{P} . As an example, let \mathcal{P} be the program of (b) from Section 2, which contains the variables $X \ Y \ Z$. The list

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

¹ Readers familiar with FORTRAN will recall that the CONTINUE command is a similar “dummy.”

is thus a state of \mathcal{P} . (The definition of state does not require that the state can actually be “attained” from some initial state.) The list

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

is also a state of \mathcal{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 \mathcal{P} .) The list

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

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

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

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

Let σ be a state of \mathcal{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

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

is 4.

Suppose we have a program \mathcal{P} and a state σ of \mathcal{P} . In order to say what happens “next,” we also need to know which instruction of \mathcal{P} is about to be executed. We therefore define a *snapshot* or *instantaneous description* of a program \mathcal{P} of length n to be a pair (i, σ) where $1 \leq i \leq n + 1$, and σ is a state of \mathcal{P} . (Intuitively the number i indicates that it is the i th instruction which is about to be executed; $i = n + 1$ corresponds to a “stop” instruction.)

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

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

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

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

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

Case 4b. σ contains the equation $V = m$ where $m \neq 0$. Then, if there

is an instruction of \mathcal{P} labeled L , j is the *least number* such that the j th instruction of \mathcal{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

$$X = 4, \quad Y = 0, \quad Z = 0$$

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

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

$$X = 4, \quad Y = 0, \quad Z = 1.$$

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

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

4. Computable Functions

We have been speaking of the function computed by a program \mathcal{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, \dots, 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 \mathcal{P} of the language \mathcal{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 \mathcal{P} be any program in the language \mathcal{S} and let r_1, \dots, r_m be m given numbers. We form the state σ of \mathcal{P} which consists of the equations

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

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

Case 1. There is a computation s_1, s_2, \dots, s_k of \mathcal{P} beginning with the initial snapshot. Then we write $\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \dots, r_m)$ for the value of the variable Y at the (terminal) snapshot s_k .

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

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

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

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

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

$$\begin{aligned} [A] \quad &\text{IF } X \neq 0 \text{ GOTO } B & (1) \\ &Z \leftarrow Z + 1 & (2) \\ &\text{IF } Z \neq 0 \text{ GOTO } E & (3) \\ [B] \quad &X \leftarrow X - 1 & (4) \\ &Y \leftarrow Y + 1 & (5) \\ &Z \leftarrow Z + 1 & (6) \\ &\text{IF } Z \neq 0 \text{ GOTO } A & (7) \end{aligned}$$

For other examples of Section 2 we have

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

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

- (c) $\psi_{\mathcal{P}}^{(2)}(r_1, r_2) = r_1,$
- (d) $\psi_{\mathcal{P}}^{(1)}(r_1) = r_1 + 0 = r_1,$
- $\psi_{\mathcal{P}}^{(3)}(r_1, r_2, r_3) = r_1 + r_2.$

For any program \mathcal{P} and any positive integer m , the function $\psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m)$ is said to be *computed* by \mathcal{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 \mathcal{P} such that

$$g(r_1, \dots, r_m) = \psi_{\mathcal{P}}^{(m)}(r_1, \dots, r_m)$$

for all r_1, \dots, 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 $g(r_1, \dots, r_m)$ is defined for all r_1, \dots, 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 \mathcal{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 \mathcal{S} :

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

For this program \mathcal{P} , $\psi_{\mathcal{P}}^{(1)}(x)$ 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 \mathcal{P} be the program of (b), Section 2. Write out a computation of \mathcal{P} beginning with the snapshot $(1, \sigma)$, where σ consists of the equations $X = 2, Y = 0, Z = 0$.

2. Let \mathcal{P} be the program

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

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

3. The same as Exercise 2 for the program

[B] IF $X \neq 0$ GOTO A
 $Z \leftarrow Z + 1$
 IF $Z \neq 0$ GOTO B
 [A] $X \leftarrow X$

4. The same as Exercise 2 for the empty program.

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 $f(x_1, \dots, x_n)$ be some partially computable function computed by the program \mathcal{P} . We shall assume that the variables that occur in \mathcal{P} are all

included in the list $Y, X_1, \dots, X_n, Z_1, \dots, Z_k$ and that the labels that occur in \mathcal{P} are all included in the list E, A_1, \dots, A_l . We also assume that for each instruction of \mathcal{P} of the form

IF $V \neq 0$ GOTO A_i

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

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

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

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

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

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

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

$$\begin{aligned} & Z_m \leftarrow 0 \\ & Z_{m+1} \leftarrow V_1 \\ & Z_{m+2} \leftarrow V_2 \\ & \vdots \\ & Z_{m+n} \leftarrow V_n \\ & Z_{m+n+1} \leftarrow 0 \\ & Z_{m+n+2} \leftarrow 0 \\ & \vdots \\ & Z_{m+n+k} \leftarrow 0 \\ & \mathcal{Q}_m \\ [E_m] \quad & W \leftarrow Z_m \end{aligned}$$

Here it is understood that the number m is chosen so large that none of the variables or labels used in \mathcal{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 \mathcal{P} equal to 0 and those corresponding to X_1, \dots, X_n equal to the values of V_1, \dots, 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 \mathcal{Q}_m terminates, the value of Z_m is $f(V_1, \dots, V_n)$, so that W finally does get the value $f(V_1, \dots, V_n)$.

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

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

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

Here is an example:

$$\begin{aligned} Z &\leftarrow X_1 - X_2 \\ Y &\leftarrow Z + X_3 \end{aligned}$$

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 \mathcal{S} to permit the use of macros which allow assignment statements of the form

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

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

IF $V \neq 0$ GOTO L

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

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

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

TRUE = 1, 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 $P(x_1, \dots, x_n)$ be any computable predicate. Then the appropriate macro expansion of

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

is simply

```
Z ← P(V1, ..., Vn)
IF Z ≠ 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

```
IF V = 0 GOTO L
```

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

```
IF X ≠ 0 GOTO E
Y ← 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 $V ← V + 1$, $V ← V - 1$ are now augmented by general assignment statements of the form $W ← f(V₁, ..., V_n)$ for any partially computable function f . Also, the original conditional branch statements $\text{IF } V ≠ 0 \text{ GOTO } L$ are now augmented by general conditional branch statements of the form $\text{IF } P(V₁, ..., V_n) \text{ GOTO } L$ 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. Let $f(x)$, $g(x)$ be computable functions and let $h(x) = f(g(x))$. Show that h is computable.
2. Show by constructing a program that the predicate $x_1 \leq x_2$ is computable.

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, \dots, g_k be functions of n variables. Let

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

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

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

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

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 *recursion*.

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

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

Hence we have available the macro $Y \leftarrow k$. The following is a program which 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 $h(0), h(1), \dots, h(x)$ and then terminates. ■

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

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

Here the function h of $n + 1$ variables is said to be obtained by *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, \dots, 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 $h(x_1, \dots, x_n, x_{n+1})$:

```
[A]   Y ← f(X1, ..., Xn)
      IF Xn+1 = 0 GOTO E
      Y ← g(Z, Y, X1, ..., Xn)
      Z ← Z + 1
      Xn+1 ← Xn+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*

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

[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 \mathcal{C} is called a *PRC¹ class* if

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

¹ This is an abbreviation for “primitive recursively closed.”

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 $u_i^n(x_1, \dots, x_n)$ is computed by the program

$$Y \leftarrow X_i$$

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

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 \mathcal{C} be some PRC class. We want to show that f belongs to \mathcal{C} . Since f is a primitive recursive function, there is a list f_1, f_2, \dots, 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 \mathcal{C} . Moreover the result of applying composition or recursion to functions in \mathcal{C} is again a function belonging to \mathcal{C} . Hence each function in the list f_1, \dots, f_n belongs to \mathcal{C} . Since $f_n = f$, f belongs to \mathcal{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 13 we shall show how to obtain a computable function which 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.

4. Some Primitive Recursive Functions

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

$$1. \quad x + y.$$

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

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

We can rewrite these equations as

$$\begin{aligned} f(x, 0) &= u_1^1(x), \\ f(x, y + 1) &= s(u_2^3(y, f(x, y), x)). \end{aligned}$$

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, the function $s(u_2^3(x_1, x_2, x_3))$ is a primitive recursive function, since it is obtained by composition of primitive recursive functions. Thus, the above 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.

$$2. \quad x \cdot y.$$

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

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

This can be rewritten

$$\begin{aligned} h(x, 0) &= n(x), \\ h(x, y + 1) &= f(u_2^3(y, h(x, y), x), u_3^3(y, h(x, y), x)). \end{aligned}$$

Here

$n(x)$ is the zero function,

$f(x_1, x_2)$ 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. Thus, the function

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

is primitive recursive since it is obtained by composition of primitive recursive functions. 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 $X \leftarrow X - 1$.

The recursion equations for $p(x)$ are simply

$$p(0) = 0,$$

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

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

6. $x \dashv y$.

The function $x \dashv y$ is defined as follows:

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

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

We show that $x \dashv y$ is primitive recursive by displaying the recursion equations:

$$x \dashv 0 = x,$$

$$x \dashv (t + 1) = p(x \dashv t).$$

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

$$|x - y| = (x \dashv y) + (y \dashv x)$$

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

$$\alpha(x) = 1 \dashv x.$$

Or we can simply write the recursion equations:

$$\alpha(0) = 1,$$

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

Exercises

1. Show that for each k , the function $f(x) = k$ is primitive recursive.
2. Prove that if $f(x)$ and $g(x)$ are primitive recursive functions, so is $f(x) + g(x)$.
- 3.* (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 is primitive recursive.
- 4.* 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.
- 5.* 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.

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.

$$9. \quad x = y.$$

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

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

$$10. \quad x \leq y.$$

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

Theorem 5.1. Let \mathcal{C} be a PRC class. If P, Q are predicates which belong to \mathcal{C} , then so are $\sim P$, $P \vee Q$, and $P \& Q$.²

² See Chapter 1, Section 4.

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

Also, we have

$$P \& Q = P \cdot Q,$$

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

Finally, the De Morgan law,

$$P \vee Q \Leftrightarrow \sim(\sim P \& \sim Q)$$

shows, using what we have already done, that $P \vee Q$ belongs to \mathcal{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 \mathcal{C} to be the class of all primitive recursive functions, we have

Corollary 5.2. If P, Q are primitive recursive predicates, then so are $\sim P$, $P \vee Q$, and $P \& Q$.

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

Corollary 5.3. If P, Q are computable predicates, then so are $\sim P$, $P \vee Q$, and $P \& Q$.

As a simple example we have

$$11. \quad x < y.$$

We can write

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

or more simply

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

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

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

Then f belongs to \mathcal{C} .

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

Proof. The result is obvious because

$$\begin{aligned} f(x_1, \dots, x_n) &= \\ g(x_1, \dots, x_n) \cdot P(x_1, \dots, x_n) + h(x_1, \dots, x_n) \cdot \alpha(P(x_1, \dots, x_n)). & \end{aligned} \quad ■$$

6. Iterated Operations and Bounded Quantifiers

Theorem 6.1 Let \mathcal{C} be a PRC class. If $f(t, x_1, \dots, x_n)$ belongs to \mathcal{C} , then so do the functions

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

and

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

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

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

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

We proceed with the correct proof.

Proof. We note the recursion equations

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

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

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

Similarly,

$$h(0, x_1, \dots, x_n) = f(0, x_1, \dots, x_n),$$

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

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

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

or

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

Then the initial recursion equations can be taken to be

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

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

with the equations for $g(t + 1, x_1, \dots, x_n)$ and $h(t + 1, x_1, \dots, x_n)$ as in the above 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 $f(t, x_1, \dots, x_n)$ belongs to the PRC class \mathcal{C} , then so do the functions

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

and

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

We have

Theorem 6.3. If the predicate $P(t, x_1, \dots, x_n)$ belongs to some PRC class \mathcal{C} , then so do the predicates³

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

Proof. We need only observe that

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

and

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

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

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

Sometimes in applying Theorem 6.3 we want to use the quantifier

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

That the theorem is still valid is clear from the relations

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

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

We continue our list of examples:

³ See Chapter 1, Section 5.

12. $y \mid x$.

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

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

while

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

The predicate is primitive recursive since

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

13. $\text{Prime}(x)$.

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., $\sigma(6) = 1 + 2 + 3 + 6 = 12$]. Show that $\sigma(x)$ 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 $\text{SQSM}(x)$ be true if x is the sum of two perfect squares; false otherwise. Show that $\text{SQSM}(x)$ is primitive recursive.

7. Minimalization

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

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

also belongs to \mathcal{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 $t_0 \leq y$,

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

but

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

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

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

Hence,

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

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

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

Thus, $\min_{t \leq y} P(t, x_1, \dots, x_n)$ is the least value of $t \leq y$ for which $P(t, x_1, \dots, x_n)$ 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 $P(t, x_1, \dots, x_n)$ belongs to some PRC class \mathcal{C} and $f(y, x_1, \dots, x_n) = \min_{t \leq y} P(t, x_1, \dots, x_n)$, then f also belongs to \mathcal{C} .

The operation “ $\min_{t \leq y}$ ” is called *bounded minimization*.

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

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

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

$$R(x, y) = x - (y \cdot \lfloor x/y \rfloor),$$

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 n th 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

$$p_0 = 0,$$

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

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

$$p_{n+1} \leq (p_n)! + 1. \quad (7.1)$$

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, \dots, 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 \mathcal{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

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

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 $P(x_1, \dots, x_n, y)$ is true, then $\min_y P(x_1, \dots, x_n, y)$ 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 $P(x_1, \dots, x_n, y)$ is a computable predicate and if

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

then g is a partially computable function.

Proof. The following program obviously computes g :

```
[A]    IF P(X1, ..., Xn, Y) GOTO E
          Y ← Y + 1
          GOTO A
```

■

Exercises

1. Let $h(x)$ be the integer n such that $n \leq \sqrt{2}x < n + 1$. Show that $h(x)$ is primitive recursive.

2. Do the same when $h(x)$ is the integer n such that

$$n \leq (1 + \sqrt{2})x < n + 1.$$

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 n th 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 n th 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

$$g(x, y) = \max_{t \leq y} R(x, t),$$

i.e., $g(x, y)$ is the largest value of $t \leq y$ for which $R(x, t)$ is true; if there is none, $g(x, y) = 0$. Prove that $g(x, y)$ is primitive recursive.

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

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

Note that $2^x(2y + 1) \neq 0$ so we can just as well omit the “dot” and write

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

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

$$\langle x, y \rangle = z, \quad (8.1)$$

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

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

this last equation has a (unique) solution because $(z + 1)/2^x$ must be odd. (The 2's have been “divided out.”) Equation (8.1) thus defines functions

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

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

$$l(z) \leq z, \quad r(z) \leq z.$$

Hence we can write

$$\begin{aligned} l(z) &= \min_{x \leq z} [(\exists y)_{\leq z} (z = \langle x, y \rangle)], \\ r(z) &= \min_{y \leq z} [(\exists x)_{\leq z} (z = \langle x, y \rangle)], \end{aligned}$$

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. $l(z), r(z) \leq z$.

We next obtain primitive recursive functions which 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, \dots, a_n) to be the number

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

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 $[a_1, \dots, a_n]$ is clearly primitive recursive.

Gödel numbering satisfies the following uniqueness property:

Theorem 8.2. If $[a_1, \dots, a_n] = [b_1, \dots, b_n]$, then

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

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

However, note that

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

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

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

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 zero to the left end of a sequence, the Gödel number of the new sequence will not be the same as the Gödel 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

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

then $(x)_i = a_i$. We set

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

Note that $(x)_0 = 0$.

We shall also use the primitive recursive function

$$\text{Lt}(x) = \min_{i \leq x} ((x)_i \neq 0 \ \& \ (\forall j)_{\leq x} (j \leq i \vee (x)_j = 0)).$$

Thus, if $x = 20 = 2^2 \cdot 5^1 = [2, 0, 1]$, then $(x)_3 = 1$, but $(x)_4 = (x)_5 = \dots = (x)_{20} = 0$. So, $\text{Lt}(20) = 3$. Also, $\text{Lt}(0) = \text{Lt}(1) = 0$. If $x > 1$, and $\text{Lt}(x) = n$, then p_n divides x but no prime greater than p_n divides x . Note that $\text{Lt}([a_1, \dots, a_n]) = n$ 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. $([a_1, \dots, a_n])_i = \begin{cases} a_i & \text{if } 1 \leq i \leq n \\ 0 & \text{otherwise.} \end{cases}$
- b. $[(x)_1, \dots, (x)_n] = x \quad \text{if } n \geq \text{Lt}(x).$

Exercises

1. Let $F(0) = 0, F(1) = 1, F(n + 2) = F(n + 1) + F(n)$. [$F(n)$ is the n th so-called Fibonacci number.] Prove that $F(n)$ is primitive recursive.

2. Let

$$h_1(x, 0) = f_1(x),$$

$$h_2(x, 0) = f_2(x),$$

$$h_1(x, t + 1) = g_1(x, h_1(x, t), h_2(x, t)),$$

$$h_2(x, t + 1) = g_2(x, h_1(x, t), h_2(x, t)).$$

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

3.* (Course-of-Values Recursion) (a) For $f(n)$ any function, we write

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

Let

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

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

(b) Let

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

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

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

CHAPTER 4

A Universal Program

1. Coding Programs by Numbers

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

$Y \ X_1 \ Z_1 \ X_2 \ Z_2 \ X_3 \ Z_3 \dots$

Next we do the same for the labels:

$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ C_2 \ D_2 \ E_2 \ A_3 \dots$

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 \mathcal{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

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

then $b = 0$ or 1 or 2 , respectively;

4. if the statement in I is

IF $V \neq 0$ GOTO 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] $X \leftarrow X + 1$

is

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

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

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

and L is the j th label in our list.

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

$$\#(\mathcal{P}) = [\#(I_1), \#(I_2), \dots, \#(I_k)] - 1. \quad (1.1)$$

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:

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

The reader will recognize this as the example given in Chapter 2 of a program which 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,

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

Thus, finally, the number of this short program is

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

Note that the number of the unlabeled instruction $Y \leftarrow Y$ 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 $Y \leftarrow Y$ 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 \mathcal{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 $\#(\mathcal{P}) = 199$, \mathcal{P} consists of 3 instructions, the second of which is the unlabeled statement $Y \leftarrow Y$. We have

$$3 = \langle 2, 0 \rangle = \langle 2, \langle 0, 0 \rangle \rangle$$

and

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

Thus, the program is

$$\begin{aligned} [B] \quad & Y \leftarrow Y \\ & Y \leftarrow Y \\ & Y \leftarrow Y + 1 \end{aligned}$$

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

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

Exercises

1. Compute $\#(\mathcal{P})$ for \mathcal{P} the programs of Exercises 4.2, 4.3, Chapter 2.
2. Find \mathcal{P} such that $\#(\mathcal{P}) = 575$.

2. The Halting Problem

In this section we want to discuss a predicate $\text{HALT}(x, y)$, which we now define. For given y , let \mathcal{P} be the program such that $\#(\mathcal{P}) = y$. Then $\text{HALT}(x, y)$ is true if $\psi_{\mathcal{P}}^{(1)}(x)$ is defined and false if $\psi_{\mathcal{P}}^{(1)}(x)$ is undefined. To put it succinctly:

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

We now prove the remarkable:

Theorem 2.1. $\text{HALT}(x, y)$ is not a computable predicate.

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

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

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

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

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

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

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 which is not computable by any program in the language \mathcal{S} . But we would like to go further; we would like to conclude the following:

There is no algorithm which, given a program of \mathcal{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 halting problem*. We reason as follows: if there were such an algorithm, we could use it to check the truth or falsity of $\text{HALT}(x, y)$ for given x, y by first obtaining program \mathcal{Q} with $\#(\mathcal{Q}) = y$ and then checking whether \mathcal{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 \mathcal{S}* . Hence this would contradict the fact that $\text{HALT}(x, y)$ 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 \mathcal{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 \mathcal{S}) such that nobody is in a position to tell whether they will ever halt. For example, it is easy to write a program \mathcal{P} of \mathcal{S} that will search through all positive integers x, y, z and numbers $n > 2$ for a solution to the equation

$$x^n + y^n = z^n, \quad (2.1)$$

and that would halt if and when such a solution were found. The statement that program \mathcal{P} never halts is equivalent to the statement that Eq. (2.1) has no solution in positive x, y, z and $n > 2$, i.e., to what is known as *Fermat's last theorem*. Despite the word "theorem" this 300-year-old problem is still very much open. Thus, nobody knows whether this program \mathcal{P} will eventually halt.

Exercise

1. Show that $\text{HALT}(x, x)$ is not computable.

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

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n), \quad \text{where } \#(\mathcal{P}) = y.$$

One of the key tools in computability theory is

Theorem 3.1 (Universality Theorem). For each $n > 0$, the function $\Phi^{(n)}(x_1, \dots, x_n, y)$ is partially computable.

We shall prove this theorem by showing how to construct, for each $n > 0$, a program \mathcal{U}_n which computes $\Phi^{(n)}$. That is, we shall have for each $n > 0$,

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

The programs \mathcal{U}_n are called *universal*. For example, \mathcal{U}_1 can be used to compute *any* partially computable function of one variable, namely: if

$f(x)$ is computed by a program \mathcal{P} and $y = \#(\mathcal{P})$, then $f(x) = \Phi^{(1)}(x, y) = \psi_{\mathcal{U}_1}^{(2)}(x, y)$. The program \mathcal{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 \mathcal{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 \mathcal{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 i th variable in our list has the value a_i and all variables after the m th have the value 0, by the Gödel number $[a_1, \dots, a_m]$. For example, the state

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

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 K th instruction is about to be executed;
 S will store the current state coded in the manner just explained.

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

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

We begin by exhibiting \mathcal{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 $X_{n+1} = \#(\mathcal{P})$, where \mathcal{P} consists of the instructions I_1, \dots, I_m , then Z gets the value $[\#(I_1), \dots, \#(I_m)]$ [see Eq. (1.1)]. S is initialized as $[0, X_1, 0, X_2, \dots, 0, X_n]$, 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:

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

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:

$$U \leftarrow r((Z)_K)$$

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

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

```
IF  $l(U) = 0$  GOTO  $N$ 
IF  $l(U) = 1$  GOTO  $A$ 
IF  $\sim(P|S)$  GOTO  $N$ 
IF  $l(U) = 2$  GOTO  $M$ 
```

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 GOTO A (for *Add*). If $l(U) \neq 0, 1$, then the current instruction is either of the form $V \leftarrow V - 1$ or IF $V \neq 0$ GOTO L . In either case, if P is not a divisor of S , i.e., if the current value of V is 0, the computation need do nothing to S . If $P|S$ and $l(U) = 2$, then the computation executes a GOTO M (for *Minus*), so that 1 can be subtracted from the exponent to which P divides S . To continue:

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

If $l(U) > 2$ and $P|S$, the current instruction is of the form IF $V \neq 0$ GOTO 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 $l((Z)_i) = l(U) - 2$. 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 GOTO C takes the computation back for the next instruction (if any) to be processed. Continuing:

$[M]$	$S \leftarrow \lfloor S/P \rfloor$
	GOTO N
$[A]$	$S \leftarrow S \cdot P$
$[N]$	$K \leftarrow K + 1$
	GOTO 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 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 \mathcal{U}_n and we put the pieces together (Fig. 3.1):

```

 $Z \leftarrow X_{n+1} + 1$ 
 $S \leftarrow \prod_{i=1}^n (p_{2i})^{X_i}$ 
 $K \leftarrow 1$ 
[C] IF  $K = \text{Lt}(Z) + 1 \vee K = 0$  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  $\sim(P|S)$  GOTO N
    IF  $l(U) = 2$  GOTO M
     $K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$ 
    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]  $Y \leftarrow (S)_1$ 
```

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

For each $n > 0$, the sequence

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

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

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

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 \mathcal{U}_n will enable us to deal with the predicates:

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

We want to prove

Theorem 3.2 (Step-Counter Theorem). The predicates $\text{STP}^{(n)}(x_1, \dots, x_n, y, t)$ are computable.

Intuitively, it is easy to see why one expects this to be true. One simply “runs” program number y for up to t steps. If the program halts, the predicate is true; otherwise it is false.

The idea of the proof is to modify the universal programs \mathcal{U}_n by adding a variable Q which functions as a step counter. Then each time through the

$Z \leftarrow X_{n+1} + 1$ $S \leftarrow \prod_{i=1}^n (p_{2i})^{x_i}$ $K \leftarrow 1$ [C] $Q \leftarrow Q + 1$ [C] IF $Q > X_{n+2} + 1$ GOTO <i>E</i> [C] IF $K = \text{Lt}(Z) + 1 \vee K = 0$ GOTO <i>F</i> $U \leftarrow r((Z)_K)$ $P \leftarrow p_{r(U)+1}$ [M] IF $l(U) = 0$ GOTO <i>N</i> [M] IF $l(U) = 1$ GOTO <i>A</i> [M] IF $\sim (P S)$ GOTO <i>N</i> [M] IF $l(U) = 2$ GOTO <i>M</i> $K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$ [M] GOTO <i>C</i> [M] $S \leftarrow [S/P]$ [M] GOTO <i>N</i> [A] $S \leftarrow S \cdot P$ [N] $K \leftarrow K + 1$ [N] GOTO <i>C</i> [F] $Y \leftarrow 1$	<i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i> <i>(*)</i>
---	--

Fig. 3.2. Program which computes $Y = \text{STP}^{(n)}(X_1, \dots, X_n, X_{n+1}, X_{n+2})$.

main loop, Q is increased by 1. Thus the program will “know” when a given number of steps has been exceeded. The program is given in Fig. 3.2. The instructions marked with an (*) are those which are not present in \mathcal{U}_n .

This program will exit with $Y = 0$ if program number X_{n+1} fails to terminate after X_{n+2} steps; otherwise it will exit with $Y = 1$.

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.* Show that for each n , $\text{STP}^{(n)}$ is primitive recursive.

4. Recursively Enumerable Sets

As was explained in Chapter 1, to say that a set B , $B \subseteq N$, belongs to some class of *functions* means that the predicate $P(x)$ belongs to the class in question, where

$$P(x) \Leftrightarrow x \in B,$$

that is, where,

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

The relation between B and $P(x)$ can also be expressed by the equation

$$B = \{x \in N \mid P(x)\}.$$

Thus, in particular, to say that the set B is computable or recursive is just to say that $P(x)$ is a computable function. Likewise, B is a primitive recursive set if $P(x)$ is a primitive recursive predicate.

We have, for example,

Theorem 4.1. Let the sets B, C belong to some PRC class \mathcal{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. ■

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

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

[Recall from Chapter 1, Section 2, that $g(r) \downarrow$ means that $g(r)$ is defined.]

The term *recursively enumerable* is usually abbreviated *r.e.* If \mathcal{P} is a program which computes the function g in (4.1), then B is simply the set of all inputs to \mathcal{P} for which \mathcal{P} eventually halts. If we think of \mathcal{P} as providing an algorithm for testing for membership in B , we see that for numbers which do belong to B , the algorithm will provide a “yes” answer; but for numbers which 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 \mathcal{S} .

We have

Theorem 4.2. If B is a recursive set, then B is r.e.

Proof. Consider the program \mathcal{P} :

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

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

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

If B and \bar{B} are both r.e., we have a pair of algorithms which 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 which will always terminate and which 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 4.3. 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 4.2, 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 \mathcal{P} and h be computed by program \mathcal{Q} , and let $p = \#(\mathcal{P})$, $q = \#(\mathcal{Q})$. Then

the program given below computes B . (That is, the program computes the characteristic function of B .)

```
[A]    IF STP(1)(X, p, T) GOTO C
        IF STP(1)(X, q, T) GOTO E
        T ← T + 1
        GOTO A
[C]    Y ← 1
```

■

Theorem 4.4. If B and C are r.e. sets so are $B \cup C$ 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

$$\begin{aligned} Y &\leftarrow g(X) \\ Y &\leftarrow h(X) \end{aligned}$$

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 \mathcal{P} and \mathcal{Q} , respectively, and let $\#(\mathcal{P}) = p$, $\#(\mathcal{Q}) = q$. Let $k(x)$ be the function computed by the program

```
[A]    IF STP(1)(X, p, T) GOTO E
        IF STP(1)(X, q, T) GOTO E
        T ← T + 1
        GOTO 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 4.5 (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 $\Phi(x, n)$. ■

■

The theorem gets its name from the fact that the sequence

$$W_0, W_1, W_2, \dots$$

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 4.6. 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.)

Exercises

1. (a) Let f be a computable function. Let $B = \{f(n) \mid n \in N\} = \{f(0), f(1), \dots\}$. Prove that B is r.e.
 (b) Now let f be strictly increasing [i.e., $f(n + 1) > f(n)$ for all n]. Prove that B is computable.
2. Show that every infinite r.e. set has an infinite recursive subset.
3. Show that there is no computable function $f(x)$ such that $f(x) = \Phi(x, x) + 1$ whenever $\Phi(x, x) \downarrow$.
4. (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?

*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 which relates the various functions $\Phi^{(n)}(x_1, x_2, \dots, x_n, y)$ for different values of n .

Theorem 5.1 (Parameter Theorem). For each $n, m > 0$, there is a primitive recursive function $S_m^n(u_1, u_2, \dots, u_n, y)$ such that

$$\Phi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi^{(m)}(x_1, \dots, x_m, S_m^n(u_1, \dots, u_n, y)). \quad (5.1)$$

Suppose that values for variables u_1, \dots, 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, \dots, x_m . Letting q be the number of a program that computes this function of m variables, we have

$$\Phi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi^{(m)}(x_1, \dots, x_m, q).$$

The parameter theorem tells us that not only does there exist such a number q , but that it can be obtained from u_1, \dots, u_n, y in a computable (in fact, primitive recursive) way.

Proof. 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

$$\Phi^{(m+1)}(x_1, \dots, x_m, u, y) = \Phi^{(m)}(x_1, \dots, x_m, S_m^1(u, y)).$$

Here $S_m^1(u, y)$ must be the number of a program which, given m inputs x_1, \dots, x_m , computes the same value as program number y does when given the $m + 1$ inputs x_1, \dots, x_m, u . Let \mathcal{P} be the program such that $\#(\mathcal{P}) = y$. 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 \mathcal{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

$$S_m^1(u, y) = \left[\left(\prod_{i=1}^u p_i \right)^{16m+10} \cdot \prod_{j=1}^{\text{Lt}(y+1)} p_{u+j}^{(y+1)_j} \right] \doteq 1,$$

a primitive recursive function. Here the numbers of the instructions of \mathcal{P} which appear as exponents in the prime power factorization of $y + 1$ have been shifted to the primes $p_{u+1}, p_{u+2}, \dots, p_{u+\text{Lt}(y+1)}$.

To complete the proof, suppose the result known for $n = k$. Then we have

$$\begin{aligned} \Phi^{(m+k+1)}(x_1, \dots, x_m, u_1, \dots, u_k, u_{k+1}, y) \\ = \Phi^{(m+k)}(x_1, \dots, x_m, u_1, \dots, u_k, S_m^1(u_{k+1}, y)) \\ = \Phi^{(m)}(x_1, \dots, x_m, S_m^k(u_1, \dots, u_k, S_m^1(u_{k+1}, y))), \end{aligned}$$

using first the result for $n = 1$, and then the induction hypothesis. But now, if we define

$$S_m^{k+1}(u_1, \dots, u_k, u_{k+1}, y) = S_m^k(u_1, \dots, u_k, S_m^1(u_{k+1}, y)),$$

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 3 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.]

*6. The Recursion Theorem

One of the most important uses of the parameter theorem is in proving a result called the recursion theorem.

Theorem 6.1 (Recursive Theorem). Let $g(z, x_1, \dots, x_m)$ be a partially computable function of $m + 1$ variables. Then there is a number e such that

$$\Phi_e^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m).$$

Proof. Consider the partially computable function

$$g(S_m^1(v, v), x_1, \dots, x_m)$$

where S_m^1 is the function that occurs in the parameter theorem. Then we have for some number z_0 ,

$$\begin{aligned} g(S_m^1(v, v), x_1, \dots, x_m) &= \Phi^{(m+1)}(x_1, \dots, x_m, v, z_0) \\ &= \Phi^{(m)}(x_1, \dots, x_m, S_m^1(v, z_0)), \end{aligned}$$

where we have used the parameter theorem. Setting $v = z_0$ and $e = S_m^1(z_0, z_0)$, we have

$$g(e, x_1, \dots, x_m) = \Phi^{(m)}(x_1, \dots, x_m, e) = \Phi_e^{(m)}(x_1, \dots, x_m). \quad \blacksquare$$

One of the many uses of the recursion theorem is that it allows one to write down definitions of functions that involve the program used to compute the function as part of its definition.

Here we give

Corollary 6.2. 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.

Another application of the recursion theorem is

Theorem 6.3 (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

$$\Phi_e(x) = g(e, x) = \Phi_{f(e)}(x).$$
 ■

Exercise

1. Prove that there is a primitive recursive function $h(u)$ such that for all x, u ,

$$\Phi(x, h(u)) = \Phi(x, \Phi(h(u), u)).$$

[Hint: Apply the recursion theorem to $g(z, x, u) = \Phi(x, \Phi(S_1^1(u, z), u))$ and then use the parameter theorem.]

*7. Rice's Theorem

Let Γ be some collection of partially computable functions of one variable. We may associate with Γ the set (usually called an *index set*)

$$R_\Gamma = \{t \in N \mid \Phi_t \in \Gamma\}.$$

Then, invoking Church's thesis, R_Γ will be a recursive set just in case there is an algorithm which accepts as inputs numbers t of programs and returns

the value TRUE or FALSE depending on whether or not the function Φ_t , computed by program number t does or does not belong to Γ . Some examples are

- (1) Γ is the set of computable functions;
- (2) Γ is the set of primitive recursive functions;
- (3) Γ is the set of partially computable functions which 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 . In fact, those who work with computer programs would be very pleased to possess algorithms which accept a program (or equivalently a program number) 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.*¹ Our proof will use the recursion theorem. 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

$$\Phi_e(x) = h(e, x) = \begin{cases} g(x) & \text{if } \Phi_e \text{ belongs to } \Gamma \\ f(x) & \text{otherwise.} \end{cases}$$

¹ 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

$$\begin{aligned} e \in R_\Gamma &\text{ implies } \Phi_e(x) = g(x) \\ &\text{implies } \Phi_e \text{ is not in } \Gamma \\ &\text{implies } e \notin R_\Gamma. \end{aligned}$$

But likewise,

$$\begin{aligned} e \notin R_\Gamma &\text{ implies } \Phi_e(x) = f(x) \\ &\text{implies } \Phi_e \text{ is in } \Gamma \\ &\text{implies } e \in R_\Gamma. \end{aligned}$$

This contradiction proves the theorem. ■

Corollary 7.2. There are no algorithms for testing a given program \mathcal{P} of the language \mathcal{S} to determine whether $\psi_{\mathcal{P}}^{(1)}(x)$ belongs to any of the classes described in (1)–(3) above.

Proof. In each case we need only find the required functions $f(x), g(x)$. For (1), (2), or (3) we can take $f(x)$ to be, say, $f(x) = u_1^1(x)$ and $g(x) = 1 - x$ [so that $g(x)$ is defined only for $x = 0, 1$]. ■

Exercises

1. Show there is no algorithm to determine of a given program \mathcal{P} in the language \mathcal{S} whether $\psi_{\mathcal{P}}^1(x) = x^2$ for all x .
2. 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 .

CHAPTER 5

Calculations on Strings

1. Numerical Representation of Strings

So far we have been dealing exclusively with computations on numbers. Now we want to extend our point of view to include computations on strings of symbols on a given alphabet. In order to extend computability theory to strings on an alphabet A , we wish to associate numbers with elements of A^* in a one-one manner. We now describe one convenient way of doing this: Let A be some given alphabet. Since A is a set, there is no order implied among the symbols. However, *we will assume in this chapter that the elements of A have been placed in some definite order*. In particular, when we write $A = \{s_1, \dots, s_n\}$, we think of the sequence s_1, \dots, s_n as corresponding to this given order. Now, let $w = s_{i_k} s_{i_{k-1}} \cdots s_{i_1} s_{i_0}$. Then we associate with w the integer

$$x = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \cdots + i_1 \cdot n + i_0. \quad (1.1)$$

With $w = 0$, we associate the number 0. (It is for this reason that we use the same symbol for both.) For example, let A consist of the symbols a, b, c given in the order shown, and let $w = baacb$. Then, the corresponding integer is

$$x = 2 \cdot 3^4 + 1 \cdot 3^3 + 1 \cdot 3^2 + 3 \cdot 3^1 + 2 = 209.$$

In order to see that the representation (1.1) is unique, we show how to retrieve the subscripts i_0, i_1, \dots, i_k from x assuming that $x \neq 0$. We define the primitive recursive functions:

$$\begin{aligned} R^+(x, y) &= \begin{cases} R(x, y) & \text{if } \sim(y|x) \\ y & \text{otherwise,} \end{cases} \\ Q^+(x, y) &= \begin{cases} \lfloor x/y \rfloor & \text{if } \sim(y|x) \\ \lfloor x/y \rfloor - 1 & \text{otherwise,} \end{cases} \end{aligned}$$

where the functions $R(x, y)$ and $\lfloor x/y \rfloor$ are as defined in Chapter 3, Section 7. Then, as we shall easily show, for $y \neq 0$,

$$\frac{x}{y} = Q^+(x, y) + \frac{R^+(x, y)}{y}, \quad 0 < R^+(x, y) \leq y.$$

This equation expresses ordinary division with quotient and remainder:

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

as long as y is not a divisor of x . If y is a divisor of x we have

$$\frac{x}{y} = \lfloor x/y \rfloor = (\lfloor x/y \rfloor - 1) + \frac{y}{y} = Q^+(x, y) + \frac{R^+(x, y)}{y}.$$

Thus, what we are doing differs from ordinary division with remainders in that “remainders” are permitted to take on values between 1 and y rather than between 0 and $y - 1$.

Now, let us set

$$u_0 = x, \quad u_{m+1} = Q^+(u_m, n). \quad (1.2)$$

Thus, by (1.1)

$$\begin{aligned} u_0 &= i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \cdots + i_1 \cdot n + i_0, \\ u_1 &= i_k \cdot n^{k-1} + i_{k-1} \cdot n^{k-2} + \cdots + i_1, \\ &\vdots \\ u_k &= i_k. \end{aligned} \quad (1.3)$$

Therefore,

$$i_m = R^+(u_m, n), \quad m = 0, 1, \dots, k. \quad (1.4)$$

Hence, for any number x satisfying (1.1), the string w can be retrieved. It is worth noting that this can be accomplished using primitive recursive functions. If we write

$$\begin{aligned} g(0, n, x) &= x, \\ g(m + 1, n, x) &= Q^+(g(m, n, x), n), \end{aligned}$$

then

$$g(m, n, x) = u_m \quad (1.5)$$

as defined by (1.2), where, of course, g is primitive recursive. Moreover, if we let $h(m, n, x) = R^+(g(m, n, x), n)$, then h is also primitive recursive, and by (1.4)

$$i_m = h(m, n, x), \quad m = 0, 1, \dots, k. \quad (1.6)$$

This method of representing strings by numbers is clearly related to the usual base n notation for numbers. To explore the connection, it is instructive to consider the alphabet

$$D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, X\}$$

in the order shown. Then the number associated with the string 45 is

$$4 \cdot 10 + 5 = 45.$$

On the other hand, the number associated with 2X is

$$2 \cdot 10 + 10 = 30.$$

(Perhaps we should read 2X as “twenty-ten”!) Clearly a string on D that does not include X is simply the usual decimal notation for the number it represents. It is numbers whose decimal representation includes a 0 which now require an X .

Thus, in the general case of an alphabet A consisting of s_1, \dots, s_n , ordered as shown, we see that we are simply using a base n representation in which the “digits” range from 1 to n instead of the usual 0 to $n - 1$. We are proceeding in this manner simply to avoid the lack of uniqueness of the usual base n representation:

$$79 = 079 = 0079 = 00079 = \text{etc.}$$

This lack of uniqueness is of course caused by the fact that leading 0s do not change the number being represented.

It is interesting to observe that the rules of elementary arithmetic (including the use of “carries”) work perfectly well with our representation. Here are a few examples:

$$\begin{array}{r} 1 \ 7 \\ +1 \ X \ 3 \\ \hline 2 \ 1 \ X \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 17 \\ +203 \\ \hline 220 \end{array}$$

$$\begin{array}{r} 29 \\ -1 \ X \\ \hline 9 \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 29 \\ -20 \\ \hline 9 \end{array}$$

$$\begin{array}{r} X \ 5 \\ \times 2 \ X \\ \hline X \ 4 \ X \\ 1 \ X \ X \\ \hline 3 \ 1 \ 4 \ X \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 105 \\ \times 30 \\ \hline 3150 \end{array}$$

(Incidentally, this shows that the common belief that the modern rules of calculation required the introduction of a digit for 0 is unjustified.) Note in particular the following examples of adding 1:

$$\begin{array}{r} X1 \\ + 1 \\ \hline X2 \end{array} \quad \begin{array}{r} 3X \\ + 1 \\ \hline 41 \end{array} \quad \begin{array}{r} 3XX \\ + 1 \\ \hline 411 \end{array} \quad \begin{array}{r} 73XX \\ + 1 \\ \hline 7411 \end{array} \quad \begin{array}{r} 49 \\ + 1 \\ \hline 4X \end{array}$$

Adding 1 to X gives a result of 1 with a carry of 1. If the string ends in more than one X , the carry propagates. Subtracting 1 is similar, with a propagating carry produced by a string ending in 1:

$$\begin{array}{r} 1X \\ - 1 \\ \hline 19 \end{array} \quad \begin{array}{r} X1 \\ - 1 \\ \hline 9X \end{array} \quad \begin{array}{r} 711 \\ - 1 \\ \hline 6XX \end{array}$$

Now we return to the general case. Given the alphabet A consisting of s_1, \dots, s_n in the order shown, the string $w = s_{i_k}s_{i_{k-1}} \cdots s_{i_1}s_{i_0}$ is called the *base n notation for the number x* defined by (1.1). (0 is the base n notation for the null string 0 for every n .) Thus when n is fixed we can regard a partial function of one or more variables on A^* as a function of the corresponding numbers. (That is, the numbers are just those which the given strings represent in base n notation.) It now makes perfect sense to speak of an m -ary partial function on A^* with values in A^* as being *partially computable*, or when it is total, as being *computable*. Similarly we can say that an m -ary function on A^* is *primitive recursive*. Note that for any alphabet $A = \{s_1, \dots, s_n\}$ with the symbols ordered as shown, s_1 denotes 1 in base n . Thus an m -ary predicate on A^* is simply a total m -ary function on A^* all of whose values are either s_1 or 0. And it now makes sense as well to speak of an m -ary predicate on A^* as being *computable*.

As was stated in Chapter 1, for a given alphabet A , any subset of A^* is called a *language on A*. Once again, by associating with the elements of A^* the corresponding numbers, we can speak of a language on A as being r.e., or recursive, or primitive recursive.

It is important to observe that whereas the usual base n notation using a 0 digit works only for $n \geq 2$, the representation (1.1) is valid even for $n = 1$. For an alphabet consisting of the single symbol 1, the string $1^{[x]}$ of length x is the base 1 notation for the number $\sum_{i=0}^{x-1} 1 \cdot (1)^i = \sum_{i=0}^{x-1} 1 = x$. That is, the base 1 (or *unary*) representation of the number x is simply a string of 1s of length x .

In thinking of numbers (that is, elements of N) as inputs to and outputs from programs written in our language \mathcal{S} , no particular representation of these numbers was specified or required. Numbers occur in the theory as purely abstract entities, just as they do in ordinary mathematics. However,

when we wish to refer to particular numbers, we do so in the manner familiar to all of us, by writing their decimal representations. These representations are, of course, really strings on the alphabet which consists of the decimal digits:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

But it is essential to avoid confusing such strings with the numbers they represent. For this reason, for the remainder of this chapter we shall avoid the use of decimal digits as symbols in our alphabets. Thus, a string of decimal digits will *always be meant to refer to a number*.

Now, let A be some fixed alphabet containing exactly n symbols, say $A = \{s_1, s_2, \dots, s_n\}$. For each $m \geq 1$, we define $\text{CONCAT}_n^{(m)}$ as follows:

$$\begin{aligned}\text{CONCAT}_n^{(1)}(u) &= u, \\ \text{CONCAT}_n^{(m+1)}(u_1, \dots, u_m, u_{m+1}) &= zu_{m+1},\end{aligned}\tag{1.7}$$

where

$$z = \text{CONCAT}_n^{(m)}(u_1, \dots, u_m).$$

Thus, for given strings $u_1, \dots, u_m \in A^*$, $\text{CONCAT}_n^{(m)}(u_1, \dots, u_m)$ is simply the string obtained by placing the strings u_1, \dots, u_m one after the other, or, as is usually said, by *concatenating* them. We will usually omit the superscript, so that, for example we may write

$$\text{CONCAT}_2(s_2s_1, s_1s_1s_2) = s_2s_1s_1s_1s_2.$$

Likewise,

$$\text{CONCAT}_6(s_2s_1, s_1s_1s_2) = s_2s_1s_1s_1s_2.$$

However, the string s_2s_1 represents the number 5 in base 2 and the number 13 in base 6. Also, the string $s_1s_1s_2$ represents the number 8 in base 2 and the number 44 in base 6. Finally, the string $s_2s_1s_1s_1s_2$ represents 48 in base 2 and 2852 in base 6. If we wish to think of CONCAT as defining functions on N (as will be necessary, for example, in showing that the functions (1.7) are primitive recursive), then the example we have been considering becomes

$$\text{CONCAT}_2(5, 8) = 48 \quad \text{and} \quad \text{CONCAT}_6(13, 44) = 2852.$$

The same example in base 10 gives

$$\text{CONCAT}_{10}(21, 112) = 21112.$$

Bearing this discussion in mind, we now proceed to give a list of primitive recursive functions on A^* which we will need later.

1. $f(u) = |u|$. This “length” function is defined on A^* , but has values in N . For each x , the number $\sum_{j=0}^x n^j$ has the base n representation $s_1^{[x+1]}$; hence

this number is the smallest number whose base n representation contains $x + 1$ digits. Thus,

$$|u| = \min_{x \leq u} \left[\sum_{j=0}^x n^j > u \right].$$

2. $g(u, v) = \text{CONCAT}_n(u, v)$. The primitive recursiveness of this function follows from the equation

$$\text{CONCAT}_n(u, v) = u \cdot n^{|v|} + v.$$

3. $\text{CONCAT}_n^{(m)}(u_1, \dots, u_m)$, as defined in (1.7) above, is primitive recursive for each $m, n \geq 1$. This follows at once from the previous example using composition.

4. $\text{RTEND}_n(w) = h(0, n, w)$, where h is as in (1.6). As a function of A^* , RTEND_n gives the right-most symbol of a given word, as is clear from (1.3) and (1.6).

5. $\text{LTEND}_n(w) = h(|x| - 1, n, w)$. LTEND_n gives the leftmost symbol of a given word.

6. $\text{RTRUNC}_n(w) = g(1, n, w)$. RTRUNC_n gives the result of removing the rightmost symbol from a given nonempty word, as is clear from (1.3) and (1.5). When we can omit reference to the base n , we often write w^- for $\text{RTRUNC}_n(w)$. Note that $0^- = 0$.

7. $\text{LTRUNC}_n(w) = w - (\text{LTEND}_n(w) \cdot n^{|w|-1})$. In the notation of (1.3), for a given nonempty word w , $\text{LTRUNC}_n(w) = w - u_k \cdot n^k$, i.e., $\text{LTRUNC}_n(w)$ is the result of removing the leftmost symbol from w .

We will now use the list of primitive recursive functions that we have just given to prove the computability of a pair of functions that can be used in changing base. Thus, let $1 \leq n < l$. Let $A \subset \tilde{A}$, where A is an alphabet of n symbols and \tilde{A} is an alphabet of l symbols. Thus a string which belongs to A^* also belongs to \tilde{A}^* . For any $x \in N$, let w be the word in A^* that represents x in base n . Then, we write $\text{UPCHANGE}_{n,l}(x)$ for the number which w represents in base l . E.g., referring to our previous example, we have $\text{UPCHANGE}_{2,6}(5) = 13$, $\text{UPCHANGE}_{2,6}(8) = 44$, $\text{UPCHANGE}_{2,6}(48) = 2852$. Also $\text{UPCHANGE}_{2,10}(5) = 21$ and $\text{UPCHANGE}_{6,10}(13) = 21$.

Next, for $x \in N$, let w be the string in \tilde{A}^* which represents x in base l , and let w' be obtained from w by crossing out all of the symbols that belong to $\tilde{A} - A$. Then, $w' \in A^*$, and we write $\text{DOWNCHANGE}_{n,l}(x)$ for the number which w' represents in base n . For example, the string $s_2 s_6 s_1$ represents the number 109 in base 6. To obtain $\text{DOWNCHANGE}_{2,6}(109)$ we cross out the s_6 , obtaining the string $s_2 s_1$, which represents 5 in base 2; thus $\text{DOWNCHANGE}_{2,6}(109) = 5$.

Although $\text{UPCHANGE}_{n,l}$ and $\text{DOWNCHANGE}_{n,l}$ are actually primitive recursive functions, we will content ourselves with proving that they are computable:

Theorem 1.1. Let $0 < n < l$. Then the functions $\text{UPCHANGE}_{n,l}$ and $\text{DOWNCHANGE}_{n,l}$ are computable.

Proof. We begin with $\text{UPCHANGE}_{n,l}$. We write a program which extracts the successive symbols of the word that the given number represents in base n and uses them in computing the number that the given word represents in base l :

[A] IF $X = 0$ GOTO E
 $Z \leftarrow \text{LTEND}_n(X)$
 $X \leftarrow \text{LTRUNC}_n(X)$
 $Y \leftarrow l \cdot Y + Z$
GOTO A

$\text{DOWNCHANGE}_{n,l}$ is handled similarly. Our program will extract the successive symbols of the word that the given number represents in base l . However, these symbols will only be used if they belong to the smaller alphabet, i.e., if as numbers they are $\leq n$:

[A] IF $X = 0$ GOTO E
 $Z \leftarrow \text{LTEND}_l(X)$
 $X \leftarrow \text{LTRUNC}_l(X)$
IF $Z > n$ GOTO A
 $Y \leftarrow n \cdot Y + Z$
GOTO A

■

Exercises

1. (a) Write the numbers 40 and 12 in base 3 notation using the “digits” {1, 2, 3}.
(b) Work out the multiplication $40 \cdot 12 = 480$ in base 3.
(c) Compute $\text{CONCAT}_n(12, 15)$ for $n = 3, 5$, and 10. Why is no calculation required in the last case?
(d) Compute the following: $\text{UPCHANGE}_{3,7}(15)$, $\text{UPCHANGE}_{2,7}(15)$, $\text{UPCHANGE}_{2,10}(15)$, $\text{DOWNCHANGE}_{3,7}(15)$, $\text{DOWNCHANGE}_{2,7}(15)$, $\text{DOWNCHANGE}_{2,10}(20)$.
2. Show that the function f whose value is the string formed of the symbols occurring in the odd-numbered places in the input [i.e., $f(a_1a_2a_3\cdots a_n) = a_1a_3\cdots$] is computable.
3. Let $\#(u, v)$ be the number of occurrences of u as a part of v [e.g., $\#(bab, ababab) = 2$]. Prove that $\#(u, v)$ is primitive recursive.

2. A Programming Language for String Computations

From the point of view of string computations, the language \mathcal{S} seems quite artificial. For example, the instruction

$$V \leftarrow V + 1$$

which is so basic for integers, seems entirely unnatural as a basic instruction for string calculations. Thus, for the alphabet $\{a, b, c\}$, applying this instruction to $bacc$ produces baa because a carry is propagated. (This will perhaps seem more evident if, temporarily ignoring our promise to avoid the decimal digits as symbols in our alphabets, we use the alphabet $\{1, 2, 3\}$ and write

$$2133 + 1 = 2211.)$$

We are now going to introduce, for each $n > 0$, a programming language \mathcal{S}_n , which is specifically designed for string calculations on an alphabet of n symbols. The languages \mathcal{S}_n will be supplied with the same input, output, and local variables as \mathcal{S} , except that we now think of them as having values in the set A^* , where A is an n symbol alphabet. Variables not otherwise initialized are to be initialized to 0. We use the same symbols as labels in \mathcal{S}_n as in \mathcal{S} and the same conventions regarding their use. The instruction types are

Instruction	Interpretation
$V \leftarrow \sigma V$ for each symbol σ in the alphabet A	Place the symbol σ to the left of the string which is the value of V .
$V \leftarrow V^-$	Delete the final symbol of the string which is the value of V . If the value of V is 0, leave it unchanged.
If V ENDS σ GOTO L for each symbol σ in the alphabet A and each label L	If the value of V ends in the symbol σ , execute next the first instruction labeled L ; otherwise proceed to the next instruction.

The formal rules of syntax in \mathcal{S}_n are entirely analogous to those for \mathcal{S} , and we omit them. Similarly, we use macro expansions quite freely. An m -ary partial function on A^* which is computed by a program in \mathcal{S}_n is said to be *partially computable* in \mathcal{S}_n . If the function is total and partially computable in \mathcal{S}_n , it is called *computable* in \mathcal{S}_n .

Although the instructions of \mathcal{S}_n refer to strings, we can just as well think of them as referring to the numbers that the corresponding strings represent in base n . For example, the numerical effect of the instruction

$$X \leftarrow s_i X$$

in the n symbol alphabet $\{s_1, \dots, s_n\}$ ordered as shown, is to replace the numerical value x by $i \cdot n^{|x|} + x$. Just as the instructions of \mathcal{S} are natural as basic numerical operations, but complex as string operations, so the instructions of \mathcal{S}_n are natural as basic string operations, but complex as numerical operations.

We now give some macros for use in \mathcal{S}_n with the corresponding expansions.

1. The macro IF $V \neq 0$ GOTO L has the expansion

```
IF V ENDS s1 GOTO L
IF V ENDS s2 GOTO L
⋮
IF V ENDS sn GOTO L
```

2. The macro $V \leftarrow 0$ has the expansion

```
[A]      V ← V-
          IF V ≠ 0 GOTO A
```

3. The macro GOTO L has the expansion

```
Z ← 0
Z ← s1Z
IF Z ENDS s1 GOTO L
```

4. The macro $V' \leftarrow V$ has the expansion shown in Fig. 2.1.

```
Z ← 0
V' ← 0
[A]  IF V ENDS s1 GOTO B1
      IF V ENDS s2 GOTO B2
      ⋮
      IF V ENDS sn GOTO Bn
      GOTO C
[Bi] V ← V-
        V' ← siV'  

        Z ← siZ
        GOTO A } i = 1, 2, ..., n
[C]   IF Z ENDS s1 GOTO D1
      IF Z ENDS s2 GOTO D2
      ⋮
      IF Z ENDS sn GOTO Dn
      GOTO E
[Di] Z ← Z-
        V ← siV  

        GOTO C } i = 1, 2, ..., n
```

Fig. 2.1. Macro expansion of $V' \leftarrow V$ in \mathcal{S}_n .

The macro expansion of $V' \leftarrow V$ in \mathcal{S}_n is quite similar to that in \mathcal{S} .
The block of instructions

```
IF  $V$  ENDS  $s_1$  GOTO  $B_1$ 
IF  $V$  ENDS  $s_2$  GOTO  $B_2$ 
:
IF  $V$  ENDS  $s_n$  GOTO  $B_n$ 
```

is usually written simply

$$\text{IF } V \text{ ENDS } s_i \text{ GOTO } B_i \quad (1 \leq i \leq n)$$

Such a block of instructions is referred to as a *filter* for obvious reasons. Note that at the point in the computation when the first "GOTO C " is executed, V' and Z will both have the original value of V , whereas V will have the value 0. On exiting, Z has the value 0, while V' retains the original value of V and V has been restored to its original value.

If $f(x_1, \dots, x_m)$ is any function which is partially computable in \mathcal{S}_n , we permit the use in \mathcal{S}_n of macros of the form

$$V \leftarrow f(V_1, \dots, V_m)$$

The corresponding expansions are carried out in a manner entirely analogous to that discussed in Chapter 2, Section 6.

We conclude this section with two examples of functions which are computable in \mathcal{S}_n for every n . The general results in the next section will make it clear that these two examples are the only bit of programming in \mathcal{S}_n that we shall need to carry out explicitly.

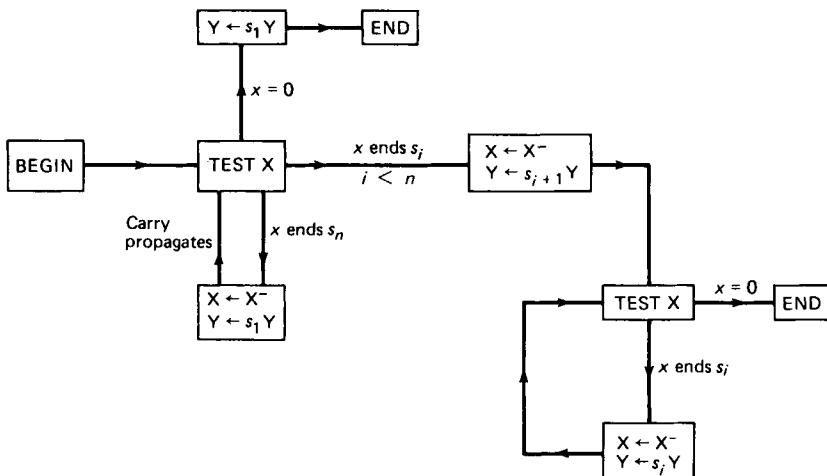


Fig. 2.2. Flow chart for computing $x + 1$ in \mathcal{S}_n .

We want to show that the function $x + 1$ is computable in \mathcal{L}_n . We let our alphabet consist of the symbols s_1, s_2, \dots, s_n ordered as shown. The desired program is exhibited in Fig. 2.3; a flow chart that shows how the program works is shown in Fig. 2.2.

[B]	IF X ENDS s_i GOTO A_i ($1 \leq i \leq n$)
	$Y \leftarrow s_1 Y$
	GOTO E
[A_i]	$X \leftarrow X^-$
	$Y \leftarrow s_{i+1} Y$
	$\left. \begin{array}{l} \\ \end{array} \right\} 1 \leq i < n$
	GOTO C
[A_n]	$X \leftarrow X^-$
	$Y \leftarrow s_1 Y$
	GOTO B
[C]	IF X ENDS s_i GOTO D_i ($1 \leq i \leq n$)
	GOTO E
[D_i]	$X \leftarrow X^-$
	$Y \leftarrow s_i Y$
	$\left. \begin{array}{l} \\ \end{array} \right\} 1 \leq i \leq n$
	GOTO C

Fig. 2.3. Program which computes $x + 1$ in \mathcal{L}_n .

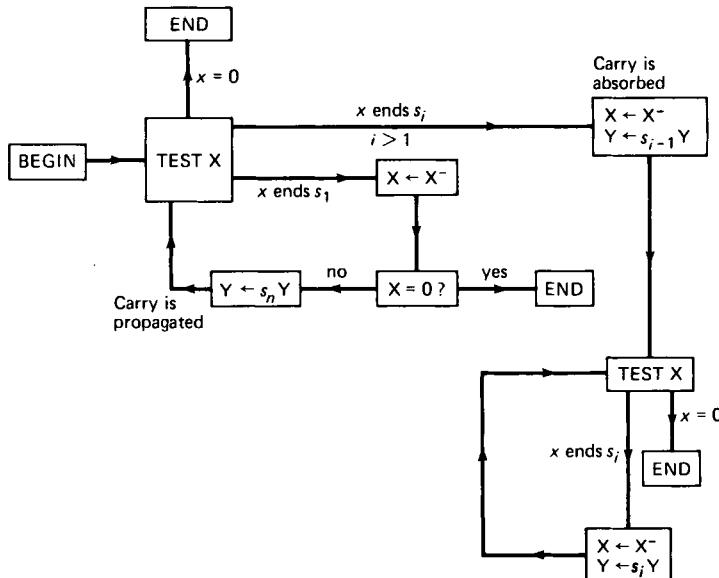


Fig. 2.4. Flow chart for computing $x + 1$ in \mathcal{L}_n .

[B]	IF X ENDS s_i GOTO A_i ($1 \leq i \leq n$) GOTO E
[A _i]	$X \leftarrow X^-$ $Y \leftarrow s_{i-1}Y$ GOTO C
[A ₁]	$X \leftarrow X^-$ IF $X \neq 0$ GOTO C_2 GOTO E
[C ₂]	$Y \leftarrow s_n Y$ GOTO B
[C]	IF X ENDS s_i GOTO D_i ($1 \leq i \leq n$) GOTO E
[D _i]	$X \leftarrow X^-$ $Y \leftarrow s_i Y$ GOTO C

$\left. \begin{array}{l} X \leftarrow X^- \\ Y \leftarrow s_i Y \end{array} \right\} 1 \leq i \leq n$

Fig. 2.5. Program which computes $x \div 1$ in \mathcal{S}_n .

Our final example is a program that computes $x \div 1$ base n . A flow chart is given in Fig. 2.4 and the actual program in \mathcal{S}_n is exhibited in Fig. 2.5. The reader should check both of these programs with some examples.

Exercises

1. Write a program in \mathcal{S}_n to compute $\#(u, v)$ as defined in Exercise 1.3.
2. Show that $f(x) = x^R$ is computable in \mathcal{S}_n . (x^R is defined in Chapter 1, Section 3.)

3. The Languages \mathcal{S} and \mathcal{S}_n

We now want to compare the functions that can be computed in the various languages we've been considering, namely, \mathcal{S} and the different \mathcal{S}_n . For the purpose of making this comparison, we take the point of view that in all of the languages, computations are "really" dealing with numbers, and that strings on an n letter alphabet are simply data objects being used to represent numbers (using base n of course).

We shall see that in fact all of these languages are equivalent. That is, a function f is partially computable if and only if it is partially computable in each \mathcal{S}_n and therefore, also, f is partially computable in any one \mathcal{S}_n if and only if it is partially computable in all of them.

To begin with we have

Theorem 3.1. A function is partially computable if and only if it is partially computable in \mathcal{S}_1 .

Proof. It is easy to see that the languages \mathcal{S} and \mathcal{S}_1 are really the same. That is, the numerical effect of the instructions

$$V \leftarrow s_1 V \quad \text{and} \quad V \leftarrow V^-$$

in \mathcal{S}_1 is the same as that of the corresponding instructions in \mathcal{S} :

$$V \leftarrow V + 1 \quad \text{and} \quad V \leftarrow V - 1.$$

Furthermore, the condition “ V ENDS s_1 ” in \mathcal{S}_1 is equivalent to the condition $V \neq 0$ in \mathcal{S} . (Since s_1 is the only symbol, ending in s_1 is equivalent to being different from the null string.) ■

This theorem shows that results we obtain about the languages \mathcal{S}_n can always be specialized to give results about \mathcal{S} by setting $n = 1$.

Next we shall prove

Theorem 3.2. If a function is partially computable, then it is also partially computable in \mathcal{S}_n for each n .

Proof. Let the function f be computed by a program \mathcal{P} in the language \mathcal{S} . We translate \mathcal{P} into a program in \mathcal{S}_n by replacing each instruction of \mathcal{P} by a macro in \mathcal{S}_n as follows:

We replace each instruction $V \leftarrow V + 1$ by the macro $V \leftarrow V + 1$, each instruction $V \leftarrow V - 1$ by the macro $V \leftarrow V - 1$, and each instruction IF $V \neq 0$ GOTO L by the macro IF $V \neq 0$ GOTO L . Here we are using the fact, proved at the end of the preceding section, that $x + 1$ and $x - 1$ are both computable in base n , and hence can each be used to define a macro in \mathcal{S}_n .

It is then obvious that the new program computes in \mathcal{S}_n the same function f that \mathcal{P} computes in \mathcal{S} . ■

This is the first of many proofs by the method of *simulation*: A program in one language is “simulated” step by step by a corresponding program in a different language.

We could now prove directly that if a function is partially computable in \mathcal{S}_n for any particular n , then it is in fact partially computable in our original sense. But it will be easier to delay doing so since the result will be an automatic consequence of our work on Post-Turing programs.

4. Post-Turing Programs

In this section, we will study yet another programming language for string manipulation, the Post-Turing language \mathcal{T} . Unlike \mathcal{S}_n , the language \mathcal{T} has no variables. All of the information being processed is placed on one linear tape. We can conveniently think of the tape as ruled into squares each

--	--	--	--	--	--	--

Fig. 4.1

of which can carry a single symbol. See Fig. 4.1. The tape is thought of as infinite in both directions. Each step of a computation is sensitive to just one symbol on the tape, the symbol on the square being “scanned.” We can think of the tape passing through a device (like a tape recorder), or we can think of the computer as a tapehead that moves along the tape and is at each moment on one definite square (or we might say “tile”). With this simple scheme, there are not many steps we can imagine. The symbol being scanned can be altered. (That is, a new symbol can be “printed” in its place.) Or which instruction of a program is to be executed next can depend on which symbol is currently being scanned. Or, finally, the head can move one square to the left or right of the square presently scanned. We are led to the following language:

Instruction	Interpretation
PRINT σ	Replace the symbol on the square being scanned by σ .
IF σ GOTO L	GOTO the first instruction labeled L if the symbol currently scanned is σ ; otherwise, continue to the next instruction.
RIGHT	Scan the square immediately to the <i>right</i> of the square presently scanned.
LEFT	Scan the square immediately to the <i>left</i> of the square presently scanned.

Although the formulation of \mathcal{T} we have presented is closer in spirit to that originally given by Emil Post, it was Turing’s analysis of the computation process that has made this formulation seem so appropriate. This language has played a fundamental role in theoretical computer science.

Turing’s analysis was obtained by abstracting from the process carried out by a human being engaged in calculating according to a mechanical deterministic algorithm. Turing reasoned that there was no loss of generality in assuming that the person used a linear paper (like the paper tape in an old-fashioned adding machine or a printing calculator) instead of two-dimensional sheets of paper. Such a calculator is then engaged in observing symbols and writing symbols. Again without loss of generality, we can assume that only one symbol at a time is observed, since any finite group of symbols can be regarded as a single “megasymbol.” Finally, we can assume that when the calculator shifts attention it is to an immediately adjacent symbol. For, to look, say, 3 symbols to the left is equivalent to moving one symbol to the left 3 successive times. And now we have arrived at precisely the Post–Turing language.

In order to speak of a function being computed by a Post-Turing program, we will need to deal with input and output. Let us suppose that we are dealing with string functions on the alphabet $A = \{s_1, s_2, \dots, s_n\}$. We will use an additional symbol, written s_0 , which we call the *blank* and use as a punctuation mark. Often we write B for the blank instead of s_0 . All of our computations will be arranged so that *all but a finite number of squares on the tape are blank*, i.e., *contain the symbol B*. We show the contents of a tape by exhibiting a finite section containing all of the nonblank squares. We indicate the square currently being scanned by an arrow pointing up, just below the scanned square.

For example we can write

$$\begin{array}{cccccc} s_1 & s_2 & B & s_2 & s_1 \\ & & \uparrow & & \end{array}$$

to indicate that the tape consists of $s_1 s_2 B s_2 s_1$ with blank squares to the left and right, and that the square currently scanned contains the s_2 furthest to the right. We speak of a *tape configuration* as consisting of the tape contents together with a specification of one square as being currently *scanned*.

Now, to compute a partial function $f(x_1, \dots, x_m)$ of m variables on A^* , we need to place the m strings x_1, \dots, x_m on the tape initially. We do this using the initial tape configuration:

$$\begin{array}{ccccccc} B & x_1 & B & x_2 & \dots & B & x_m \\ & \uparrow & & & & & \end{array}$$

That is, the inputs are separated by single blanks, and the symbol initially scanned is the blank immediately to the left of x_1 . Here are a few examples:

1. $n = 1$, so the alphabet is $\{s_1\}$. We want to compute a function $f(x_1, x_2)$ and the initial values are $x_1 = s_1 s_1$, $x_2 = s_1$. Then the tape configuration initially will be

$$\begin{array}{cccccc} B & s_1 & s_1 & B & s_1 \\ & \uparrow & & & \end{array}$$

Of course, there are infinitely many blank squares to the left and right of the finite section we have shown:

$$\dots B \ B \ B \ B \ s_1 \ s_1 \ B \ s_1 \ B \ B \ B \ \dots$$

2. $n = 2$, $x_1 = s_1 s_2$, $x_2 = s_2 s_1$, $x_3 = s_2 s_2$. Then the tape configuration is initially

$$\begin{array}{cccccc} B & s_1 & s_2 & B & s_2 & s_1 & B & s_2 & s_2 \\ & \uparrow & & & & & & & \end{array}$$

3. $n = 2$, $x_1 = 0$, $x_2 = s_2 s_1$, $x_3 = s_2$. Then the tape configuration is initially

$$\begin{array}{cccccc} B & B & s_2 & s_1 & B & s_2. \\ & \uparrow & & & & \end{array}$$

4. $n = 2$, $x_1 = s_1 s_2$, $x_2 = s_2 s_1$, $x_3 = 0$. Then the tape configuration is initially

$$\begin{array}{cccccc} B & s_1 & s_2 & B & s_2 & s_1 & B. \\ & \uparrow & & & & & \end{array}$$

Note there is no way to distinguish this initial tape configuration from that for which there are only 2 inputs $x_1 = s_1 s_2$ and $x_2 = s_2 s_1$. In other words, with this method of placing inputs on the tape, the number of arguments must be provided externally. It cannot be read from the tape.

A simple example of a Post-Turing program is given in Fig. 4.2.

```
PRINT s1
LEFT
PRINT s2
LEFT
```

Fig. 4.2

Beginning with input x , this program outputs $s_2 s_1 x$. More explicitly, beginning with a tape configuration

$$\begin{array}{c} B & x \\ & \uparrow \end{array}$$

this program halts with the tape configuration

$$\begin{array}{c} B & s_2 & s_1 & x. \\ & \uparrow & & \end{array}$$

Next, for a slightly more complicated example, we consider Fig. 4.3. Here we are assuming that the alphabet is $\{s_1, s_2, s_3\}$. Beginning with a tape configuration

$$\begin{array}{c} B & x \\ & \uparrow \end{array}$$

this program halts with the tape configuration

$$\begin{array}{c} B & x & s_1 & s_1. \\ & \uparrow & & \end{array}$$

```

[A]    RIGHT
      IF  $s_1$  GOTO A
      IF  $s_2$  GOTO A
      IF  $s_3$  GOTO A
      PRINT  $s_1$ 
      RIGHT
      PRINT  $s_1$ 
[C]    LEFT
      IF  $s_1$  GOTO C
      IF  $s_2$  GOTO C
      IF  $s_3$  GOTO C

```

Fig. 4.3

The computation proceeds by first moving right until the blank to the right of x is located. The symbol s_1 is then printed twice and then the computation proceeds by moving left until the blank to the left of x is again located.

Figure 4.4 exhibits another example, this time with the alphabet $\{s_1, s_2\}$. The effect of this program is to “erase” all of the occurrences of s_2 in the input string, that is to replace each s_2 by B . For the purpose of reading output values off the tape, these additional B s are ignored. Thus, if $f(x)$ is the function which this last program computes, we have, for example,

$$\begin{aligned}
 f(s_2 s_1 s_2) &= s_1, \\
 f(s_1 s_2 s_1) &= s_1 s_1, \\
 f(0) &= 0.
 \end{aligned}$$

Of course, the initial tape configuration

$$\begin{array}{ccccccc}
 B & s_1 & s_2 & s_1 & s_2 & s_1 & s_2 \\
 & & \uparrow & & & &
 \end{array}$$

leads to the final tape configuration

$$\begin{array}{ccccccc}
 B & s_1 & B & s_1 & B & s_1 & s_2 \\
 & & & & & \uparrow &
 \end{array}$$

but the blanks are ignored in reading the output.

```

[C]    RIGHT
      IF B GOTO E
      IF  $s_2$  GOTO A
      IF  $s_1$  GOTO C
[A]    PRINT B
      IF B GOTO C

```

Fig. 4.4

[A]	RIGHT
	IF B GOTO E
	PRINT M
[B]	RIGHT
	IF s_1 GOTO B
[C]	RIGHT
	IF s_1 GOTO C
	PRINT s_1
[D]	LEFT
	IF s_1 GOTO D
	IF B GOTO D
	PRINT s_1
	IF s_1 GOTO A

Fig. 4.5

For our final example we are computing a string function on the alphabet $\{s_1\}$. However the program uses three symbols, B , s_1 , and M . The symbol M is a *marker* to keep track of a symbol being copied. The program is given in Fig. 4.5. Beginning with the tape configuration

$$\begin{array}{c} B \ u \\ \uparrow \end{array}$$

where u is a string in which only the symbol s_1 occurs, this program will terminate with the tape configuration

$$\begin{array}{c} B \ u \ B \ u. \\ \uparrow \end{array}$$

(Thus we can say that this program computes the function $2x$ using unary notation.) The computation proceeds by replacing each successive s_1 (going from left to right) by the marker M and then copying the s_1 on the right.

We conclude this section with some definitions. Let $f(x_1, \dots, x_m)$ be an m -ary partial function on the alphabet $\{s_1, \dots, s_n\}$. Then the program \mathcal{P} in the Post-Turing language \mathcal{T} is said to *compute* f if when started in the tape configuration

$$\begin{array}{c} B \ x_1 \ B \ \dots \ B \ x_m \\ \uparrow \end{array}$$

it eventually halts if and only if $f(x_1, \dots, x_m)$ is defined and if, on halting, the string $f(x_1, \dots, x_m)$ can be read off the tape by ignoring all symbols other than s_1, \dots, s_n . (That is, any “markers” left on the tape as well as blanks are to be ignored.) Note that we are thus permitting \mathcal{P} to contain instructions that mention symbols other than s_1, \dots, s_n .

The program \mathcal{P} will be said to compute f strictly if two additional conditions are met:

1. No instruction in \mathcal{P} mentions any symbol other than s_0, s_1, \dots, s_n .
2. Whenever \mathcal{P} halts, the tape configuration is of the form

$$\dots B \ B \ B \ B \ y \ B \ B \dots,$$

↑

where the string y contains no blanks.

Thus when P computes f strictly, the output is available in a consecutive block of squares on the tape.

Exercises

1. Construct a Post-Turing program which computes $f(x) = x^R$ strictly. (See Exercise 2.2.)
2. (a) Construct a Post-Turing program using only the symbols s_0, s_1 which computes the function $f(x) = 2x$ in base 1 strictly.
 (b) The same as (a) for $f(x, y) = x \div y$.

5. Simulation of \mathcal{S}_n in \mathcal{T}

In this section we will prove

Theorem 5.1. If $f(x_1, \dots, x_m)$ is partially computable in \mathcal{S}_n , then there is a Post-Turing program which computes f strictly.

Let \mathcal{P} be a program in \mathcal{S}_n which computes f . We assume that in addition to the input variables X_1, \dots, X_m and the output variable Y , \mathcal{P} uses the local variables Z_1, \dots, Z_k . Thus, altogether \mathcal{P} uses $m + k + 1$ variables:

$$X_1, \dots, X_m, Z_1, \dots, Z_k, Y.$$

We set $l = m + k + 1$ and write these variables, in the same order, as

$$V_1, \dots, V_l.$$

We shall construct a Post-Turing program \mathcal{Q} which simulates \mathcal{P} step by step. Since all of the information available to \mathcal{Q} will be on the tape, we must allocate space on the tape to contain the values of the variables V_1, \dots, V_l .

Our scheme is simply that at the beginning of each simulated step, the tape configuration will be as follows:

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m \ B \ z_1 \ B \ \dots \ B \ z_k \ B \ y;$$

↑

where $x_1, x_2, \dots, x_m, z_1, \dots, z_k, y$ are the current values computed for the variables $X_1, X_2, \dots, X_m, Z_1, \dots, Z_k, Y$. This scheme is especially convenient in that the initial tape configuration

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m$$

↑

is already in the correct form, since the remaining variables are initialized to be 0. So we must show how to program the effect of each instruction type of \mathcal{L}_n in the language \mathcal{T} . Various macros in \mathcal{T} will be useful in doing this, and we now present them.

The macro

GOTO L

has the expansion

$$\begin{aligned} & \text{IF } s_0 \text{ GOTO } L \\ & \text{IF } s_1 \text{ GOTO } L \\ & \vdots \\ & \text{IF } s_n \text{ GOTO } L \end{aligned}$$

The macro

RIGHT TO NEXT BLANK

has the expansion

$$\begin{aligned} [A] \quad & \text{RIGHT} \\ & \text{IF } B \text{ GOTO } E \\ & \text{GOTO } A \end{aligned}$$

Similarly the macro

LEFT TO NEXT BLANK

has the expansion:

$$\begin{aligned} [A] \quad & \text{LEFT} \\ & \text{IF } B \text{ GOTO } E \\ & \text{GOTO } A \end{aligned}$$

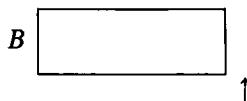
The macro

MOVE BLOCK RIGHT

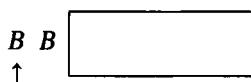
has the expansion

[C]	LEFT IF s_0 GOTO A_0 IF s_1 GOTO A_1 ⋮ IF s_n GOTO A_n
[A_i]	RIGHT PRINT s_i LEFT GOTO C
[A_0]	RIGHT PRINT B LEFT

The effect of the macro MOVE BLOCK RIGHT beginning with a tape configuration



in which the string in the rectangular box contains no blanks, is to terminate with the tape configuration



Finally we will use the macro

ERASE A BLOCK

whose expansion is

[A]	RIGHT IF B GOTO E PRINT B GOTO A
-----	---

This program causes the head to move to the right, with everything erased between the square at which it begins and the first blank to its right.

We adopt the convention that a number ≥ 0 in square brackets after the name of a macro indicates that the macro is to be repeated that number of

times. For example,

RIGHT TO NEXT BLANK [3]

is short for

RIGHT TO NEXT BLANK
RIGHT TO NEXT BLANK
RIGHT TO NEXT BLANK

We are now ready to show how to simulate the three instruction types in the language \mathcal{S}_n by Post-Turing programs. We begin with

$$V_j \leftarrow s_i V_j$$

In order to place the symbol s_i to the left of the j th variable on the tape, the values of the variables V_j, \dots, V_l must all be moved over one square to the right to make room. After the s_i has been inserted, we must remember to go back to the blank at the left of the value of V_1 in order to be ready for the next simulated instruction. The program is

RIGHT TO NEXT BLANK [!]
MOVE BLOCK RIGHT [$l - j + 1$]
RIGHT
PRINT s_i
LEFT TO NEXT BLANK [j]

Next we must show how to simulate

$$V_j \leftarrow V_j^-$$

The complication is that if the value of V_j is the null word, we want it left unchanged. So we move to the blank immediately to the right of the value of V_j . By moving one square to the left we can detect whether the value of V_j is null (if it is, there are two consecutive blanks). Here is the program:

RIGHT TO NEXT BLANK [j]
LEFT
IF B GOTO C
MOVE BLOCK RIGHT [j]
RIGHT
GOTO E
[C] LEFT TO NEXT BLANK [$j - 1$]

The final instruction type in \mathcal{S}_n is

IF V_j ENDS s_i GOTO L

and the corresponding Post-Turing program is

```

RIGHT TO NEXT BLANK [j]
LEFT
IF  $s_i$  GOTO C
GOTO D
[C]  LEFT TO NEXT BLANK [j]
GOTO L
[D]  RIGHT
LEFT TO NEXT BLANK [j]

```

This completes the simulation of the three instruction types of \mathcal{S}_n . Thus, given our program \mathcal{P} in the language \mathcal{S}_n , we can compile a corresponding program of \mathcal{T} . When this corresponding program terminates, the tape configuration will be

$$\dots B \ B \ B \ x_1 \ B \dots B \ x_m \ B \ z_1 \ B \dots B \ z_k \ B \ y \ B \ B \ B \dots,$$

↑

where the values between blanks are those of the variables of \mathcal{P} on its termination. However, we wish only y to remain as output. Hence to obtain our program \mathcal{Q} in the language \mathcal{T} we put at the end of the compiled Post-Turing program the following:

ERASE A BLOCK [$l - 1$]

After this last has been executed, all but the last block will have been erased and the tape configuration will be

$$\dots B \ B \ B \ B \ y \ B \ B \ B \dots$$

↑

Thus, the output is in precisely the form required for us to be able to assert that our Post-Turing program computes f strictly.

6. Simulation of \mathcal{T} in \mathcal{S}

In this section we will prove

Theorem 6.1. If there is a Post-Turing program which computes the partial function $f(x_1, \dots, x_m)$, then f is partially computable.

What this theorem asserts is that if the m -ary partial function f on A^* is computed by a program of \mathcal{T} , then there is a program of \mathcal{S} which computes f (regarded as an m -ary partial function on the base n numerical values of

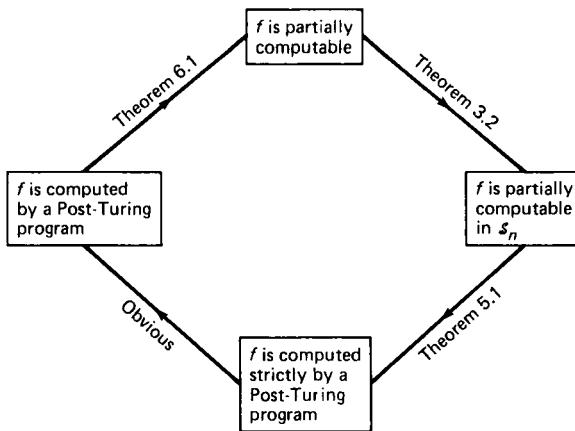


Fig. 6.1

the strings). Before giving the proof we observe some of the consequences of this theorem. As shown in Fig. 6.1, the theorem completes a “circle” of implications. Thus all of the conditions in the figure are equivalent. To summarize:

Theorem 6.2. Let f be an m -ary partial function on A^* , where A is an alphabet of n symbols. Then the following conditions are all equivalent:

1. f is partially computable;
2. f is partially computable in \mathcal{S}_n ;
3. f is computed strictly by a Post-Turing program;
4. f is computed by a Post-Turing program.

The equivalence of so many different notions of computability constitutes important evidence for the correctness of our identification of intuitive computability with these notions, i.e., for the correctness of Church’s thesis.

Shifting our point of view to that of an m -ary partial function on N , we have

Corollary 6.3. For any $n, l \geq 1$, an m -ary partial function f on N is partially computable in \mathcal{S}_n if and only if it is also partially computable in \mathcal{S}_l .

Proof. Each of these conditions is equivalent to the function f being partially computable. ■

By considering the language \mathcal{S}_1 we have

Corollary 6.4. Every partially computable function is computed strictly by some Post-Turing program which uses only the symbols s_0, s_1 .

Now we return to the proof of Theorem 6.1. Let \mathcal{P} be a Post-Turing program which computes f . We want to construct a program \mathcal{Q} in the language \mathcal{S} which computes f . \mathcal{Q} will consist of three sections:

BEGINNING
MIDDLE
END

The MIDDLE section will simulate \mathcal{P} in a step-by-step “interpretive” manner. The task of BEGINNING is to arrange the input to \mathcal{Q} in the appropriate format for MIDDLE, and the task of END is to extract the output.

Let us suppose that f is an m -ary partial function on A^* , where $A = \{s_1, \dots, s_n\}$. The Post-Turing program \mathcal{P} will also use the blank B and perhaps additional symbols (we are not assuming that the computation is strict!) s_{n+1}, \dots, s_r . We write the symbols which \mathcal{P} uses in the order

$$s_1, \dots, s_n, s_{n+1}, \dots, s_r, B.$$

The program \mathcal{Q} will simulate \mathcal{P} by using the numbers that strings on this alphabet represent in base $r + 1$ as “codes” for the corresponding strings. Note that as we have arranged the symbols, the blank B represents the number $r + 1$. For this reason we will write the blank as s_{r+1} instead of s_0 . The tape configuration at a given stage in the computation by \mathcal{P} will be kept track of by \mathcal{Q} using three numbers stored in the variables L , H , and R . The value of H will be the numerical value of the symbol currently being scanned by the *head*. The value of L will be a number which represents in base $r + 1$ a string of symbols w such that the tape contents to the left of the head consists of infinitely many blanks followed by w . The value of R represents in a similar manner the string of symbols to the right of the head. For example, consider the tape configuration

$$\dots B \ B \ B \ B \ s_2 \ s_1 \ B \ s_3 \ s_1 \ s_2 \ B \ B \ B \ \dots$$

↑

Here $r = 3$, so we will use the base 4. Then we would have

$$H = 3.$$

We might have

$$L = 2 \cdot 4^2 + 1 \cdot 4 + 4 = 40,$$

$$R = 1 \cdot 4 + 2 = 6.$$

An alternative representation could show some of the blanks on the left or right explicitly. For example, recalling that B represents $r + 1 = 4$,

$$L = 4 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4 + 4 = 296,$$

$$R = 1 \cdot 4^3 + 2 \cdot 4^2 + 4 \cdot 4 + 4 = 116.$$

Now it is easy to simulate the instruction types of \mathcal{T} by programs of \mathcal{S} . An instruction PRINT s_i is simulated by

$$H \leftarrow i$$

An instruction IF s_i GOTO L is simulated by

$$\text{IF } H = i \text{ GOTO } L$$

An instruction RIGHT is simulated by

$$\begin{aligned} L &\leftarrow \text{CONCAT}_{r+1}(L, H) \\ H &\leftarrow \text{LTEND}_{r+1}(R) \\ R &\leftarrow \text{LTRUNC}_{r+1}(R) \end{aligned}$$

Similarly an instruction LEFT is simulated by

$$\begin{aligned} R &\leftarrow \text{CONCAT}_{r+1}(H, R) \\ H &\leftarrow \text{RTEND}_{r+1}(L) \\ L &\leftarrow \text{RTRUNC}_{r+1}(L) \end{aligned}$$

Now the section MIDDLE of \mathcal{Q} can be assembled simply by replacing each instruction of \mathcal{P} by its simulation.

In writing BEGINNING and END we must deal with the fact that f is an m -ary function on $\{s_1, \dots, s_n\}^*$. Thus the initial values of X_1, \dots, X_m for \mathcal{Q} will be numbers which represent the input strings in base n . Theorem 1.1 will enable us to change base as required. The section BEGINNING has the task of calculating the initial values of L, H, R , that is, the values corresponding to the tape configuration

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m,$$

↑

where the numbers x_1, \dots, x_m are represented in base n notation. Thus the section BEGINNING of \mathcal{Q} can simply be taken to be

$$\begin{aligned} L &\leftarrow 0 \\ H &\leftarrow r + 1 \\ Z_1 &\leftarrow \text{UPCHANGE}_{n,r+1}(X_1) \\ Z_2 &\leftarrow \text{UPCHANGE}_{n,r+1}(X_2) \\ &\vdots \\ Z_m &\leftarrow \text{UPCHANGE}_{n,r+1}(X_m) \\ R &\leftarrow \text{CONCAT}_{r+1}(Z_1, r+1, Z_2, r+1, \dots, r+1, Z_m) \end{aligned}$$

Finally, the section END of \mathcal{Q} can be taken simply to be

$Z \leftarrow \text{CONCAT}_{r+1}(L, H, R)$

$Y \leftarrow \text{DOWNCHANGE}_{n,r+1}(Z)$

We have now completed the description of the program \mathcal{Q} which simulates \mathcal{P} and our proof is complete. ■

Turing Machines

1. Internal States

Now we turn to a variant of the Post–Turing language which is closer to Turing’s original formulation. Instead of thinking of a list of instructions, we imagine a device capable of various internal states. The device is, at any particular instant, scanning a square on a linear tape just like the one used by Post–Turing programs. The combination of the current internal state with the symbol on the square currently scanned is then supposed to determine the next “action” of the device. As suggested by Turing’s analysis of the computation process (see Chapter 5, Section 4), we can take the next action to be either “printing” a symbol on the scanned square, or moving one square to the right or left. Finally, the device must be permitted to enter a new state.

We use the symbols q_1, q_2, q_3, \dots to represent states and we write s_0, s_1, s_2, \dots to represent symbols which can appear on the tape, where as usual $s_0 = B$ is the “blank.” By a *quadruple* we mean an expression of one of the following forms consisting of four symbols:

- (1) $q_i \ s_j \ s_k \ q_l,$
- (2) $q_i \ s_j \ R \ q_l,$
- (3) $q_i \ s_j \ L \ q_l.$

We intend a quadruple of type (1) to signify that in state q_i scanning symbol s_j , the device will print s_k and go into state q_l . Similarly, a quadruple of type (2) signifies that in state q_i scanning s_j the device will move one square to the right and then go into state q_l . Finally, a quadruple of type (3) is like one of type (2) except that the motion is to the left.

We now define a *Turing machine* to be a finite set of quadruples, no two of which begin with the same pair $q_i s_j$. Actually, any finite set of quadruples is called a *nondeterministic Turing machine*. But for the present we will deal only with *deterministic Turing machines* which satisfy the additional “consistency” condition forbidding two quadruples of a given machine to begin with the same pair $q_i s_j$, thereby guaranteeing that at any stage a Turing machine is capable of only one action. Nondeterministic Turing machines are discussed in Section 5.

The *alphabet* of a given Turing machine \mathcal{M} consists of all of the symbols s_i which occur in quadruples of \mathcal{M} except s_0 .

We stipulate that a Turing machine always begins in state q_1 . Moreover, a Turing machine will *halt* if it is in state q_i scanning s_j and *there is no quadruple of the machine which begins* $q_i s_j$. With these understandings, and using the same conventions concerning input and output that were employed in connection with Post–Turing programs, it should be clear what it means to say that some given Turing machine \mathcal{M} *computes* a partial function f on A^* for a given alphabet A .

Just as for Post–Turing programs, we may speak of a Turing machine \mathcal{M} which computes a function *strictly*, namely: assuming that \mathcal{M} computes f where f is a partial function on A^* , we say that \mathcal{M} *computes f strictly* if

1. the alphabet of \mathcal{M} is A ;
2. whenever \mathcal{M} halts, the final configuration has the form

$$\begin{array}{c} By \\ \uparrow \\ q_i \end{array}$$

where y contains no blanks.

Writing $s_0 = B$, $s_1 = 1$ consider the Turing machine with alphabet {1}:

$$\begin{array}{cccccc} q_1 & B & R & q_2 \\ q_2 & 1 & R & q_2 \\ q_2 & B & 1 & q_3 \\ q_3 & 1 & R & q_3 \\ q_3 & B & 1 & q_1. \end{array}$$

We can check the computation:

$$\begin{array}{ccccccccccc} B111, & B111, & \dots, & B111B, & B1111, & B1111B, & B11111 \\ \uparrow & \uparrow & & \uparrow & \uparrow & \uparrow & \uparrow \\ q_1 & q_2 & & q_2 & q_3 & q_3 & q_1 \end{array}$$

Table 1.1

Symbol	State		
	q_1	q_2	q_3
B	$R \ q_2$	$1 \ q_3$	$1 \ q_1$
1		$R \ q_2$	$R \ q_3$

The computation halts because there is no quadruple beginning $q_1 1$. Clearly, this Turing machine computes (but not strictly) the function $f(x) = x + 2$, where we are using unary (base 1) notation. The steps of the computation, which explicitly exhibit the state of the machine, the string of symbols on the tape, as well as the individual square on the tape being scanned, are called *configurations*.

It is sometimes helpful to exhibit a Turing machine by giving a state versus symbol table. Thus, for example the above Turing machine could be represented as shown in Table 1.1.

Another useful representation is by a state transition diagram. The Turing machine being discussed thus could be represented by the diagram shown in Fig. 1.1.

We now prove

Theorem 1.1. Any partial function which can be computed by a Post-Turing program can be computed by a Turing machine using the same alphabet.

Proof. Let \mathcal{P} be a given Post-Turing program consisting of the instructions I_1, \dots, I_k , and let s_0, s_1, \dots, s_n be a list which includes all of the symbols mentioned in \mathcal{P} . We shall construct a Turing machine \mathcal{M} that simulates \mathcal{P} .

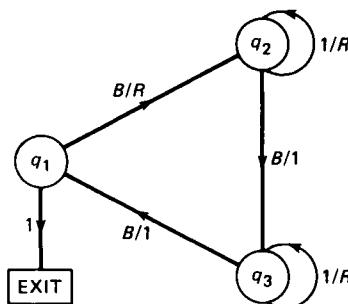


Fig. 1.1

The idea is that \mathcal{M} will be in state q_i precisely when \mathcal{P} is about to execute instruction I_i . Thus, if I_i is “PRINT s_k ,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ s_k \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

If I_i is “RIGHT,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ R \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

If I_i is “LEFT,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ L \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

Finally, if I_i is “IF s_k GOTO L ,” let m be the least number such that I_m is labeled L if there is an instruction of \mathcal{P} labeled L ; otherwise let $m = K + 1$. We place in \mathcal{M} the quadruple

$$q_i \ s_k \ s_k \ q_m$$

as well as all of the quadruples:

$$q_i \ s_j \ s_j \ q_{i+1}, \quad j = 0, 1, \dots, n; \quad j \neq k.$$

It is clear that the actions of \mathcal{M} correspond precisely to the instructions of \mathcal{P} , so we are done. ■

Using Corollary 6.4 from Chapter 5 and the *proof* of Theorem 1.1, we have

Theorem 1.2. Let f be an m -ary partially computable functions on A^* for a given alphabet A . Then there is a Turing machine \mathcal{M} which computes f strictly.

It is particularly interesting to apply this theorem to the case $A = \{1\}$. Thus, if $f(x_1, \dots, x_m)$ is any partially computable function on N , there is a Turing machine that computes f using only the symbols B and 1. The initial configuration corresponding to inputs x_1, \dots, x_m is

$$\begin{array}{c} B \ 1^{[x_1]} \ B \ \dots \ B \ 1^{[x_m]} \\ \uparrow \\ q_1 \end{array}$$

and the final configuration when $f(x_1, \dots, x_m) \downarrow$ will be

$$\begin{array}{c} B \ 1^{[f(x_1, \dots, x_m)]}. \\ \uparrow \\ q_{K+1} \end{array}$$

Next we shall consider a variant notion of Turing machines: machines which consist of quintuples instead of quadruples. There are two kinds of quintuples:

$$\begin{aligned} q_i & s_j & s_k & R & q_l, \\ q_i & s_j & s_k & L & q_l. \end{aligned}$$

The first quintuple signifies that when the machine is in state q_i scanning s_j it will print s_k and *then* move one square to the right and go into state q_l . And naturally, the second quintuple is the same, except that the motion is to the left. A finite set of quintuples no two of which begin with the same pair $q_i s_j$ is called a *quintuple Turing machine*. We can easily prove

Theorem 1.3. Any partial function which can be computed by a Turing machine can be computed by a quintuple Turing machine using the same alphabet.

Proof. Let \mathcal{M} be a Turing machine with states q_1, \dots, q_K and alphabet $\{s_1, \dots, s_n\}$. We construct a quintuple Turing machine $\bar{\mathcal{M}}$ to simulate \mathcal{M} . The states of $\bar{\mathcal{M}}$ will be $q_1, \dots, q_K, q_{K+1}, \dots, q_{2K}$.

For each quadruple of \mathcal{M} of the form $q_i s_j R q_l$ we place the corresponding quintuple $q_i s_j s_j R q_l$ in $\bar{\mathcal{M}}$. Similarly, for each quadruple $q_i s_j L q_l$ in \mathcal{M} , we place the quintuple $q_i s_j s_j L q_l$ in $\bar{\mathcal{M}}$. And, for each quadruple $q_i s_j s_k q_l$ in \mathcal{M} , we place in $\bar{\mathcal{M}}$ the quintuple $q_i s_j s_k R q_{K+i}$. Finally we place in $\bar{\mathcal{M}}$ all quintuples of the form

$$q_{K+i} s_j s_j L q_i, \quad i = 1, \dots, K; \quad j = 0, 1, \dots, n.$$

Quadruples requiring motion are simulated easily by quintuples. But a quadruple requiring a “print” necessitates using a quintuple which causes a motion after the “print” has taken place. The final list of quintuples undoes the effect of this unwanted motion. The extra states q_{K+1}, \dots, q_{2K} serve to “remember” that we have gone a square too far to the right. ■

Finally, we will complete another circle by proving

Theorem 1.4. Any partial function which can be computed by a quintuple Turing machine can be computed by a Post–Turing program using the same alphabet.

Combining Theorems 1.1, 1.3, and 1.4, we will have

Corollary 1.5. For a given partial function f , the following are equivalent:

1. f can be computed by a Post–Turing program;
2. f can be computed by a Turing machine;
3. f can be computed by a quintuple Turing machine.

Proof of Theorem 1.4. Let \mathcal{M} be a given quintuple Turing machine with states q_1, \dots, q_K and alphabet $\{s_1, \dots, s_n\}$. We associate with each state q_i a label A_i and with each pair $q_i s_j$ a label B_{ij} . Each label A_i is to be placed next to the first instruction in the filter:

$[A_i]$	IF s_0 GOTO B_{i0} ,
	IF s_1 GOTO B_{i1}
	⋮
	IF s_n GOTO B_{in}

If \mathcal{M} contains the quintuple $q_i \ s_j \ s_k \ R \ q_l$, then we introduce the block of instructions

$[B_{ij}]$	PRINT s_k
	RIGHT
	GOTO A_l

Similarly, if \mathcal{M} contains the quintuple $q_i \ s_j \ s_k \ L \ q_l$, then we introduce the block of instructions:

$[B_{ij}]$	PRINT s_k
	LEFT
	GOTO A_l

Finally, if there is no quintuple in \mathcal{M} beginning $q_i s_j$, we introduce the block

$[B_{ij}]$	GOTO E
------------	----------

Then we can easily construct a Post-Turing program which simulates \mathcal{M} simply by putting all of these blocks and filters one under the other. The order is irrelevant except for one restriction: The filter labeled A_1 must begin the program. The entire program is listed in Figure 1.2. ■

$[A_1]$	IF s_0 GOTO B_{10}
	⋮
	IF s_n GOTO B_{1n}
$[A_2]$	IF s_0 GOTO B_{20}
	⋮
	IF s_n GOTO B_{2n}
$[A_K]$	IF s_0 GOTO B_{K0}
	⋮
	IF s_n GOTO B_{Kn}
$[B_{i_1 j_1}]$	PRINT s_{k_1}
	RIGHT
	GOTO A_{l_1}
$[B_{i_2 j_2}]$	PRINT s_{k_2}
	⋮

Fig. 1.2

Exercises

1. Let T be the Turing machine consisting of the quadruples

$$\begin{aligned} q_1 & B \ R \ q_2 \\ q_2 & 1 \ R \ q_3 \\ q_3 & B \ R \ q_4 \\ q_4 & 1 \ B \ q_1 \\ q_4 & B \ R \ q_4. \end{aligned}$$

For each integer x , let $g(x)$ be the number of occurrences of 1 on the tape when and if T halts when started with the read–write head one square to the left of the initial 1, with input $1^{[x]}$. What is the function $g(x)$?

2. Write out the quadruples constituting a Turing machine which computes the function

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$$

in base 1. Exhibit the state transition diagram for your machine.

3. Give precise definitions of *configuration*, *computation*, and *Turing Machine \mathcal{M} computes the function f* . (Compare Chapter 2, Section 3.)

2. A Universal Turing Machine

Let us now recall the partially computable function $\Phi(x, z)$ from Chapter 4. For fixed z , $\Phi(x, z)$ is the unary partial function computed by the program whose number is z . Let \mathcal{M} be a Turing machine (in either quadruple or quintuple form) which computes this function with alphabet $\{1\}$. For reasons that we will explain, it is appropriate to call this machine \mathcal{M} *universal*.

Let $g(x)$ be any partially computable function of one variable and let z_0 be the number of some program in the language \mathcal{S} which computes g . Then if we begin with a configuration

$$\begin{array}{c} B \ x \ B \ z_0 \\ \uparrow \\ q_1 \end{array}$$

(where x and z_0 are written as blocks of 1s, i.e., in unary notation), and let \mathcal{M} proceed to compute, \mathcal{M} will compute $\Phi(x, z_0)$, i.e., $g(x)$. Thus, \mathcal{M} can be used to compute any partially computable function of one variable.

\mathcal{M} provides a suggestive model of an all-purpose computer, in which data and programs are stored together in a single “memory.” We can think of z_0 as a coded version of the program for computing g and x as the input to that program. Turing’s construction of a universal computer in 1936 provided reason to believe that, at least in principle, an all-purpose computer would be possible, and was thus an anticipation of the modern digital computer.

3. The Languages Accepted by Turing Machines

Given a Turing machine \mathcal{M} with alphabet $A = \{s_1, \dots, s_n\}$, a word $u \in A^*$ is said to be *accepted* by \mathcal{M} if when \mathcal{M} begins with the configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

it will eventually halt. The set of all words $u \in A^*$ that \mathcal{M} accepts is called the *language accepted by \mathcal{M}* . An important problem in the theory of computation involves characterizing the languages accepted by various kinds of computing devices. It is easy for us to solve this problem for Turing machines.

Theorem 3.1. A language is accepted by some Turing machine if and only if the language is r.e.

Proof. Let L be the language accepted by a Turing machine \mathcal{M} with alphabet A . Let $g(x)$ be the unary function on A^* that \mathcal{M} computes. Then g is a partially computable function (by Corollary 1.5 and by Theorem 6.2 in Chapter 5). Now,

$$L = \{x \in A^* | g(x) \downarrow\}. \quad (3.1)$$

Hence L is r.e.

Conversely, let L be r.e. Then there is a partially computable function $g(x)$ such that (3.1) holds. Using Theorem 1.2, let \mathcal{M} be a Turing machine with alphabet $\{s_1, \dots, s_n\}$ which computes $g(x)$ strictly. Then \mathcal{M} accepts L . ■

Naturally Theorem 3.1 is also true for quintuple Turing machines.

Let us consider the special case $A = \{1\}$. Then we have

Theorem 3.2. A set U of numbers is r.e. if and only if there is a Turing machine \mathcal{M} with alphabet $\{1\}$ that accepts $1^{[x]}$ if and only if $x \in U$.

Proof. This follows immediately from Theorem 3.1 and the fact that the base 1 representation of the number x is the string $1^{[x]}$. ■

This is an appropriate place to consider some annoying ambiguities in our notation of r.e. language. Thus, for example, consider the language

$$L_0 = \{a^{[n]} \mid n > 0\},$$

on the alphabet $\{a, b\}$. According to our definitions, to say that L_0 is an r.e. language is to say that the set of numbers which the strings in L_0 represent in base 2 is an r.e. set of numbers. But, *this set of numbers is not determined until an order is specified for the letters of the alphabet*. If we take a, b in the order shown, then the set of numbers which represent strings in L_0 is evidently

$$Q_1 = \{2^n - 1 \mid n > 0\},$$

while if we take the letters in the order b, a , the set of numbers which represents strings in L_0 is

$$Q_2 = \{2x \mid x \in Q_1\} = \{2^{n+1} - 2 \mid n > 0\}.$$

Now, although there is no difficulty whatever in showing that Q_1 and Q_2 are both r.e. sets, it is nevertheless a thoroughly unsatisfactory state of affairs to be forced to be concerned with such matters in asserting that L_0 is an r.e. language. Here Theorem 3.1 comes to our rescue. The notion of a given string being accepted by a Turing machine does not involve imposing any order on the symbols of the alphabet. Hence, Theorem 3.1 implies immediately that whether a particular language on a given alphabet is r.e. is *independent of how the symbols of the alphabet are ordered*. The same is clearly true of a language L on a given alphabet A being recursive since this is equivalent to L and $A^* - L$ both being r.e.

Another ambiguity arises from the fact that a particular language may be considered with respect to more than one alphabet. Thus, let A be an n -letter alphabet and let \tilde{A} be an m -letter alphabet containing A , so that $m > n$. Then a language L on the alphabet A is simply some subset of A^* , so that L is also a language on the larger alphabet \tilde{A} . Thus, depending on whether we are thinking of L as a language on A or as a language on \tilde{A} , we will have to read the strings in L as being the notation for integers in *base n or in base m, respectively*. Hence, we are led to the unpleasant possibility that whether L is r.e. might actually depend on which alphabet we are considering. As an example, we may take $A = \{a\}$ and $\tilde{A} = \{a, b\}$, and consider the language L_0 above, where

$$L_0 \subseteq A^* \subseteq \tilde{A}^*.$$

We have already seen that our original definition of L_0 's being r.e. as a language on the alphabet \tilde{A} amounts to requiring that the set of numbers Q_1 or Q_2 (depending on the order of the symbols a, b) be r.e. However if we take our alphabet to be A , then the relevant set of numbers is

$$Q_3 = \{n \in N \mid n > 0\}.$$

We remove all such ambiguities by proving

Theorem 3.3. Let $A \subseteq \tilde{A}$ where A and \tilde{A} are alphabets and let $L \subseteq A^*$. Then L is an r.e. language on the alphabet A if and only if L is an r.e. language on \tilde{A} .

Proof. Let L be r.e. considered as a language on A and let \mathcal{M} be a Turing machine with alphabet A that accepts L . Let $\tilde{\mathcal{M}}$ be obtained from \mathcal{M} by adjoining to it the quadruples $q s s q$ for each symbol $s \in \tilde{A} - A$, and each state q of \mathcal{M} . Thus $\tilde{\mathcal{M}}$ will enter an “infinite loop” if it ever encounters a symbol in $\tilde{A} - A$. Since $\tilde{\mathcal{M}}$ has alphabet \tilde{A} and accepts the language L , we conclude from Theorem 3.1 that L is an r.e. language on \tilde{A} .

Conversely, let L be r.e. as a language on \tilde{A} , and let \mathcal{M} be a Turing machine with alphabet \tilde{A} that accepts L . Let $g(x)$ be the function on A^* that \mathcal{M} computes. (The symbols belonging to $\tilde{A} - A$ thus serve as “markers.”) Since $L \subseteq A^*$, we have

$$L = \{x \in A^* \mid g(x) \downarrow\}.$$

Since $g(x)$ is partially computable, it follows that L is an r.e. language on A . ■

Corollary 3.4. Let A, \tilde{A}, L be as in Theorem 3.3. Then L is a recursive language on A if and only if L is a recursive language on \tilde{A} .

Proof. First let L be a recursive language on A . Then L and $A^* - L$ are r.e. languages on A and therefore on \tilde{A} . Moreover, since

$$\tilde{A}^* - L = (\tilde{A}^* - A^*) \cup (A^* - L),$$

and since $\tilde{A}^* - A^*$ is r.e., as the reader can easily show (see Exercise 2), it follows from Theorem 4.4 in Chapter 4 that $\tilde{A}^* - L$ is r.e. Hence, L is a recursive language on \tilde{A} .

Conversely, if L is a recursive language on \tilde{A} , then L and $\tilde{A}^* - L$ are r.e. languages on \tilde{A} and therefore L is an r.e. language on A . Moreover, since

$$A^* - L = (\tilde{A}^* - L) \cap A^*,$$

and since A^* is obviously r.e. (as a language on A and therefore on \tilde{A}), it follows from Theorem 4.4 in Chapter 4 that $A^* - L$ is an r.e. language on \tilde{A} and hence on A . Thus, L is a recursive language on A . ■

Exercises

1. Write out the quadruples constituting a Turing machine which accepts the language consisting of all words on the alphabet $\{a, b\}$ of the form $a^{[i]}ba^{[i]}$.
2. Complete the proof of Corollary 3.4 by showing that $\tilde{A}^* - A^*$ is an r.e. language.

4. The Halting Problem for Turing Machines

We can use the results of the previous section to obtain a sharpened form of the unsolvability of the halting problem.

By the halting problem for a *fixed* given Turing machine \mathcal{M} we mean the problem of finding an algorithm to determine whether \mathcal{M} will eventually halt starting with a given configuration. We have

Theorem 4.1. There is a Turing machine \mathcal{K} with alphabet $\{1\}$ which has an unsolvable halting problem.

Proof. Take for the set U in Theorem 3.2, some r.e. set which is not recursive (e.g., the set K from Chapter 4). Let \mathcal{K} be the corresponding Turing machine. Thus \mathcal{K} accepts a string of 1s if and only if its length belongs to U . Hence, $x \in U$ if and only if \mathcal{K} eventually halts when started with the configuration

$$\begin{array}{c} B \ 1^{[x]} \\ \uparrow \\ q_1 \end{array}$$

Thus, if there were an algorithm for solving the halting problem for \mathcal{K} , it could be used to test a given number x for membership in U . Since U is not recursive, such an algorithm is impossible. ■

This is really a stronger result than was obtained in Chapter 4. What we can prove about Turing machines just using Theorem 2.1 in Chapter 4 is that there is no algorithm which can be used, given a Turing machine and an initial configuration, to determine whether the Turing machine will ever halt. Our present result gives a *fixed* Turing machine whose halting problem is unsolvable. Actually, this result could also have been easily obtained from the earlier one by using a universal Turing machine.

Next, we show how the unsolvability of the halting problem can be used to obtain another unsolvable problem concerning Turing machines. We begin with a Turing machine \mathcal{K} with alphabet $\{1\}$ that has an unsolvable

halting problem. Let the states of \mathcal{K} be q_1, \dots, q_k . We will construct a Turing machine $\tilde{\mathcal{K}}$ by adjoining to the quadruples of \mathcal{K} the following quadruples:

$$q_i \ B \ B \ q_{k+1}$$

for $i = 1, 2, \dots, k$ for which no quadruple of \mathcal{K} begins $q_i B$, and

$$q_i \ 1 \ 1 \ q_{k+1}$$

for $i = 1, 2, \dots, k$ when no quadruple of \mathcal{K} begins $q_i 1$. Thus, \mathcal{K} eventually halts beginning with a given configuration if and only if $\tilde{\mathcal{K}}$ eventually is in state q_{k+1} . We conclude

Theorem 4.2. There is a Turing machine $\tilde{\mathcal{K}}$ with alphabet $\{1\}$ and a state q_m such that there is no algorithm which can determine whether $\tilde{\mathcal{K}}$ will ever arrive at state q_m when it begins at a given configuration.

Exercises

1. Prove that there is a Turing machine \mathcal{M} such that there is no algorithm which can determine of a given configuration whether \mathcal{M} will eventually halt with a completely blank tape when started with the given tape configuration.
2. Prove that there is a Turing machine \mathcal{M} with alphabet $\{s_1, s_2\}$ such that there is no algorithm which can determine whether \mathcal{M} starting with a given configuration will ever print the symbol s_2 .
3. Jack and Jill have been working as programmers for a year. They are discussing their work. We listen in:

JACK: We are working on a wonderful program, AUTOCHECK. AUTOCHECK will accept PL/I programs as inputs and will return the values OK or LOOPS depending on whether the given program is or is not free of infinite loops.

JILL: Big deal! We have a mad mathematician in our firm who has developed an algorithm so complicated that no program can be written to execute it no matter how much space and time is allowed.

Comment on and criticize Jack and Jill's statements.

5. Nondeterministic Turing Machines

As already mentioned, a *nondeterministic Turing machine* is simply an arbitrary finite set of quadruples. Thus, what we have been calling a Turing machine is simply a special kind of nondeterministic Turing machine. For

emphasis, we will sometimes refer to ordinary Turing machines as *deterministic*.

A configuration

$$\begin{array}{c} \dots s_j \dots \\ \uparrow \\ q_i \end{array}$$

is called *terminal* with respect to a given nondeterministic Turing machine (and the machine is said to *halt*) if it contains no quadruple beginning $q_i \ s_j$. (This, of course, is exactly the same as for deterministic Turing machines.) We use the symbol \vdash (borrowed from logic) placed between a pair of configurations to indicate that the transition from the configuration on the left to the one on the right is permitted by one of the quadruples of the machine under consideration.

As an example, consider the nondeterministic Turing machine given by the quadruples

q_1	B	R	q_2
q_2	1	R	q_3
q_2	B	B	q_4
q_3	1	R	q_2
q_3	B	B	q_3
q_4	B	R	q_4
q_4	B	B	q_5

Then we have

So far the computation has been entirely determined; however, at this point the nondeterminism plays a role. We have

$$B \ 1 \ 1 \ 1 \ 1 \ B \vdash B \ 1 \ 1 \ 1 \ 1 \ B,$$

\uparrow

q_4

\uparrow

q_5

at which the machine halts. But we also have

$$\begin{array}{ccccccccc} B & 1 & 1 & 1 & 1 & B \vdash B & 1 & 1 & 1 & 1 & B & B \vdash B & 1 & 1 & 1 & 1 & B & B \vdash \dots \\ & \uparrow & & & & \uparrow & & & & & \uparrow & & & & & & & \\ & q_4 & & & & q_4 & & & & & q_4 & & & & & & & \end{array}$$

Let $A = \{s_1, \dots, s_n\}$ be a given alphabet and let $u \in A^*$. Then the nondeterministic Turing machine \mathcal{M} is said to *accept* u if there exists a sequence of configurations $\gamma_1, \gamma_2, \dots, \gamma_m$ such that γ_1 is the configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

γ_m is terminal with respect to \mathcal{M} , and $\gamma_1 \vdash \gamma_2 \vdash \gamma_3 \vdash \dots \vdash \gamma_m$. In this case, the sequence $\gamma_1, \gamma_2, \dots, \gamma_m$ is called an *accepting computation by \mathcal{M} for u* . If A is the alphabet of \mathcal{M} , then the *language accepted by \mathcal{M}* is the set of all $u \in A^*$ that are accepted by \mathcal{M} .

Of course, for deterministic Turing machines, this definition gives nothing new. However, it is important to keep in mind the distinctive feature of acceptance by nondeterministic Turing machines. It is perfectly possible to have an infinite sequence

$$\gamma_1 \vdash \gamma_2 \vdash \gamma_3 \vdash \dots$$

of configurations, where γ_1 is

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

even though u is accepted by \mathcal{M} . It is only necessary that there be *some* sequence of transitions leading to a terminal configuration. One sometimes expresses this by saying, “The machine is always permitted to guess the correct next step.”

Thus in the example given above, taking the alphabet $A = \{1\}$, we have that \mathcal{M} accepts 1111. In fact the language accepted by \mathcal{M} is $\{1^{[2^n]}\}$. (See Exercise 2.)

Since a Turing machine is also a nondeterministic Turing machine, Theorem 3.1 can be weakened to give

Theorem 5.1. For every r.e. language L , there is a nondeterministic Turing machine \mathcal{M} which accepts L .

The converse is also true: the language accepted by a nondeterministic Turing machine must be r.e. By Church’s thesis, it is clear that this should be true. It is only necessary to “run” a nondeterministic Turing machine \mathcal{M}

on a given input u , *following all alternatives at each step*, and giving the value (say) 0, if termination is reached along any branch. This defines a function which is *intuitively* partially computable and whose domain is the language accepted by \mathcal{M} . However, a detailed proof along these lines would be rather messy.

Fortunately the converse of Theorem 5.1 will be an easy consequence of the methods we will develop in the next chapter.

Exercises

1. Let L be the set of all words on the alphabet $\{a, b\}$ which contain at least two consecutive occurrences of b . Construct a nondeterministic Turing machine *which never moves left* and accepts L .
2. Show that the nondeterministic Turing machine \mathcal{M} used as an example in this section accepts the set $\{1^{2^n}\}$.

6. Variations on the Turing Machine Theme

So far we have three somewhat different formulations of Turing's conception of computation: the Post-Turing programming language, Turing machines as made up of quadruples, and quintuple Turing machines. The proof that these formulations are equivalent was quite simple. This is true in part because all three involved a single tapehead on a single two-way infinite tape. But it is easy to imagine other arrangements. In fact, Turing's original formulation was in terms of a tape which was infinite in only one direction, that is, with a first or leftmost square. See Fig. 6.1. We can also think of permitting several tapes, each of which can be one-way or two-way infinite and each with its own tapehead. There might even be several tapeheads per tape. As one would expect, programs can be shorter when several tapes are available. But, if we believe Church's thesis, we certainly would expect all of these formulations to be equivalent. In this section we will indicate briefly how this equivalence can be demonstrated.

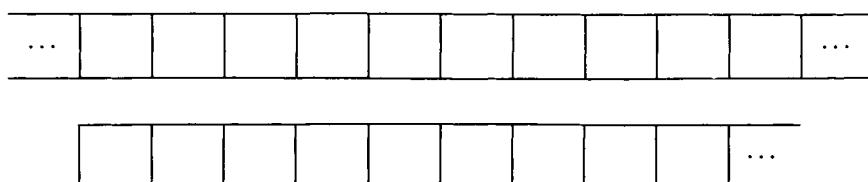


Fig. 6.1. Two-way infinite versus one-way infinite tape.

Let us begin by considering one-way infinite tapes. To make matters definite, we assume that we are representing a Turing machine as a set of quadruples. It is necessary to make a decision about the effect of a quadruple $q_i s_j L q_k$ in case the tapehead is already at the left end of the tape. There are various possibilities, and it really does not matter very much which we adopt. For definiteness we assume that an instruction to move left will be interpreted as a *halt* in case the tapehead is already at the leftmost square. Now it is pretty obvious that anything that a Turing machine could do on a one-way infinite tape could also be done on a two-way infinite tape, and we leave details to the reader.

How can we see that any partially computable function can be computed by a Turing machine on a one-way infinite tape? One way is by simply examining the proof of Theorem 5.1 in Chapter 5, which shows how a computation in any of the languages \mathcal{L}_n can be simulated by a program in the Post-Turing language \mathcal{T} . In fact, the program \mathcal{Q} in the language \mathcal{T} which is constructed to simulate a given program \mathcal{P} in the language \mathcal{L}_n has the particular property that when \mathcal{Q} is executed, the tapehead never moves to the left of the square initially scanned. Hence, the program \mathcal{Q} would work exactly as well on a one-way infinite tape whose leftmost square is initially scanned. And, it is an easy matter, as in the proof of Theorem 1.1, to convert \mathcal{Q} into a Turing machine.

Although this is an entirely convincing argument, we would like to mention another approach which is interesting in its own right, namely: we directly face the question, how can the information contained in a two-way infinite tape be handled by a Turing machine with one tapehead on a one-way infinite tape? The intuitive idea is to think of a two-way infinite tape as being “hinged” so it can be folded as in Fig. 6.2. Thus our two-way infinite tape can be represented by a one-way infinite tape with two “tracks,” an “upper” and a “lower.” Moreover, by adding enough symbols to the alphabet, we

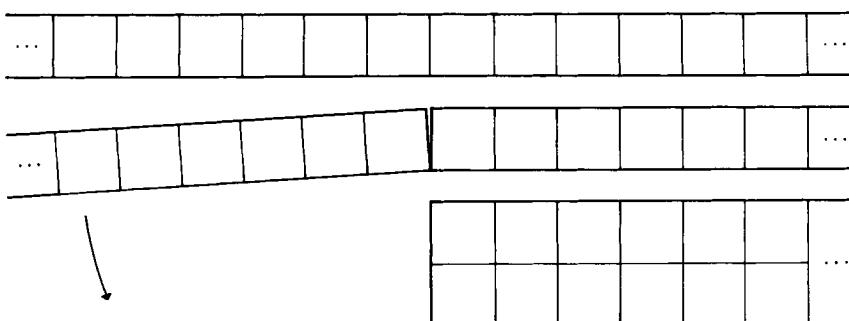


Fig. 6.2

can code each pair consisting of an upper and a lower symbol by a single symbol.

Thus, let us begin with a Turing machine \mathcal{M} with alphabet $A = \{s_1, \dots, s_n\}$ and states q_1, \dots, q_K . Let \mathcal{M} compute a unary¹ partial function g on A_0^* , where $A_0 \subseteq A$. Thus the input configuration when \mathcal{M} is computing $g(x)$ for $x \in A_0^*$ will be

$$\begin{array}{c} B \ x \\ \uparrow \\ q_1 \end{array}$$

We will construct a Turing machine $\bar{\mathcal{M}}$ which computes g on a one-way infinite tape. The initial configuration for $\bar{\mathcal{M}}$ will be

$$\begin{array}{c} \# \ B \ x \\ \uparrow \\ q_1 \end{array}$$

where $\#$ is a special symbol which will occupy the leftmost square on the tape for most of the computation. The alphabet of $\bar{\mathcal{M}}$ will be

$$A \cup \{\#\} \cup \{b_j^i \mid 0 \leq i, j \leq n\},$$

where we think of the symbol b_j^i as indicating that s_i is on the upper track and s_j is on the lower track. The states of $\bar{\mathcal{M}}$ are q_1, q_2, q_3, q_4, q_5 , and

$$\{\bar{q}_i, \tilde{q}_i \mid i = 1, 2, \dots, K\}$$

as well as certain additional states.

We can think of the quadruples constituting $\bar{\mathcal{M}}$ as made up of three sections: BEGINNING, MIDDLE, and END. BEGINNING serves to copy the input on the upper track putting blanks on the corresponding lower track of each square. BEGINNING consists of the quadruples

$$\begin{aligned} & q_1 \ B \ R \ q_2 \\ & q_2 \ s_i \ R \ q_2 \quad i = 1, 2, \dots, n, \\ & q_2 \ B \ L \ q_3 \\ & q_3 \ s_i \ b_0^i \ q_3 \quad i = 0, 1, 2, \dots, n, \\ & q_3 \ b_0^i \ L \ q_3 \quad i = 0, 1, 2, \dots, n, \\ & q_3 \ # \ R \ \bar{q}_1. \end{aligned}$$

Thus, starting with the configuration

$$\begin{array}{c} \# \ B \ s_2 \ s_1 \ s_3 \\ \uparrow \\ q_1 \end{array}$$

¹ The restriction to unary functions is, of course, not essential.

BEGINNING will halt in the configuration

$$\begin{array}{c} \# \ b_0^0 \ b_0^2 \ b_0^1 \ b_0^3 \ B. \\ \uparrow \\ \bar{q}_1 \end{array}$$

Note that b_0^0 is different from $s_0 = B$. MIDDLE will consist of quadruples corresponding to those of \mathcal{M} as well as additional quadruples as indicated in Table 6.1. The states \bar{q}_i, \tilde{q}_i correspond to actions on the upper track and lower track, respectively. Note in (b) and (c) that on the lower track left and right are reversed. The quadruples in (d) replace single blanks B by double blanks b_0^0 as needed. The quadruples (e) arrange for switchover from the upper to the lower track. It should be clear that MIDDLE simulates \mathcal{M} .

Table 6.1

Quadruple of \mathcal{M}	Quadruple of $\bar{\mathcal{M}}$
(a) $q_i s_j s_k q_l$	$\bar{q}_i b_m^j b_m^k \bar{q}_l \quad m = 0, 1, \dots, n$ $\tilde{q}_i b_j^m b_k^m \tilde{q}_l \quad m = 0, 1, \dots, n$
(b) $q_i s_j R q_l$	$\bar{q}_i b_m^j R \bar{q}_l \quad m = 0, 1, \dots, n$ $\tilde{q}_i b_j^m L \tilde{q}_l \quad m = 0, 1, \dots, n$
(c) $q_i s_j L q_l$	$\bar{q}_i b_m^j L \bar{q}_l \quad m = 0, 1, \dots, n$ $\tilde{q}_i b_j^m R \tilde{q}_l \quad m = 0, 1, \dots, n$
(d) —————	$\bar{q}_i B b_0^0 \bar{q}_i \quad i = 1, 2, \dots, K$ $\tilde{q}_i B b_0^0 \tilde{q}_i \quad i = 1, 2, \dots, K$
(e) —————	$\bar{q}_i \# R \bar{q}_i \quad i = 1, 2, \dots, K$ $\tilde{q}_i \# L \tilde{q}_i \quad i = 1, 2, \dots, K$

END has the task of translating the output into a word on the original alphabet A . This task is complicated by the fact that the output is split between the two tracks. To begin with, END contains the following quadruples:

$$\left. \begin{array}{l} \bar{q}_i b_m^j b_m^k q_4 \\ \tilde{q}_i b_j^m b_k^m q_4 \end{array} \right\} \begin{array}{l} \text{whenever } \mathcal{M} \text{ contains no quadruple} \\ \text{beginning } q_i s_j, \text{ for } m = 0, 1, \dots, n; \quad 0 \leq i, j \leq n, \end{array}$$

$$q_4 b_j^i L q_4,$$

$$q_4 \# B q_5.$$

For each initial configuration for which \mathcal{M} halts, the effect of BEGINNING, MIDDLE, and this part of END is to ultimately produce a configuration of the form

$$\begin{array}{c} B \ b_{j_1}^{i_1} \ b_{j_2}^{i_2} \ \dots \ b_{j_k}^{i_k}. \\ \uparrow \\ q_5 \end{array}$$

The remaining task of END is to convert the tape contents into

$$s_{j_k} \ s_{j_{k-1}} \ \dots \ s_{j_1} \ s_{i_1} \ s_{i_2} \ \dots \ s_{i_k}.$$

Instead of giving quadruples for accomplishing this, we exhibit a program in the Post-Turing language \mathcal{T} , so that we can make use of some of the macros available in that language. Of course, this program can easily be translated into a set of quadruples using the method of proof of Theorem 1.1. Because our macros for \mathcal{T} were designed for use with “blocks” of symbols containing no blanks, we will use $\#$ instead of $s_0 = B$ in carrying out our translation. One final pass will be needed to replace each $\#$ by B . The program is given in Fig. 6.3. It works as follows:

[D]	RIGHT TO NEXT BLANK
	MOVE BLOCK RIGHT
	RIGHT
[C]	RIGHT
	IF b_j^i GOTO A_j^i ($0 \leq i, j \leq n$)
	IF B GOTO F
	GOTO C
[A_j^i]	PRINT s_i ($0 < i \leq n, 0 \leq j \leq n$)
	GOTO B_j
[A_j^0]	PRINT $\#$ ($0 \leq j \leq n$)
	GOTO B_j
[B_j]	LEFT TO NEXT BLANK ($0 < j \leq n$)
	PRINT s_j
	GOTO D
[B_0]	LEFT TO NEXT BLANK
	PRINT $\#$
	GOTO D
[F]	LEFT
	IF s_j GOTO F ($0 < j \leq n$)
	IF $\#$ GOTO G
	IF B GOTO E
[G]	PRINT B
	GOTO F

Fig. 6.3

Each b_j^i is processed going from left to right. b_j^i is replaced by s_i (or by $\#$ if $i = 0$) and s_j (or $\#$ if $j = 0$) is printed on the left. The “MOVE BLOCK RIGHT” macro is used to make room on the tape for printing the successive symbols from the “lower” track. As an example let us apply the program of Fig. 6.3 to the configuration

$$\begin{array}{cccc} B & b_1^2 & b_1^0 & b_0^1 \\ \uparrow & & & \end{array}$$

In Fig. 6.4 we show various stages in the computation; in each case the tape configuration is followed by the label on the next instruction to be executed.

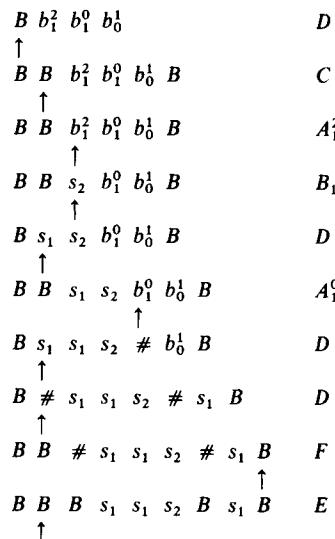


Fig. 6.4

The technique of thinking of the tape of a Turing machine as decomposed into a number of parallel tracks has numerous uses. (It will appear again in Chapter 10.) For the moment we note that it can be used to simulate the behavior of a multitape Turing machine by an ordinary Turing machine. For, in the first place a second track can be used to show the position of a tapehead on a one-tape machine as in the example shown in Fig. 6.5; the 1 under the s_3 shows the position of the head. In an entirely similar manner the contents of k tapes and the position of the tapehead on each can be represented as a single tape with $2k$ tracks. Using this representation, it is easy to see how to simulate any computations by a k -tape Turing machine using only one tape. The same result can also be obtained indirectly using the method of

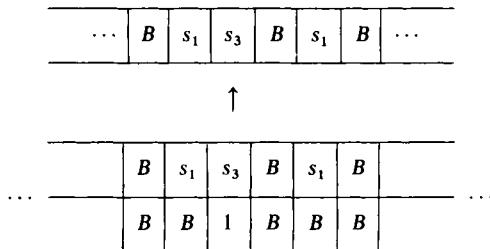


Fig. 6.5

proof of Theorem 6.1 in Chapter 5 to show that any function computed by a k -tape Turing machine is partially computable.

Exercises

1. Give a formal description of a Turing machine which uses three tapes: one with a “read only” head for input, one with a “write only” head for output, and one for “working.” Give an appropriate definition of computability by such machines and prove the equivalence with computability by ordinary Turing machines.
2. Do the same for a Turing machine with input tape, output tape, and k working tapes for any $k \geq 1$.
3. Let the Post-Turing language be augmented by the instructions UP, DOWN so that it can deal with computations on a two-dimensional “tape” infinite in all four directions. Supply an appropriate defintion of what it means to compute a function by a program in this language, and prove that any function computed by such a program is partially computable.

Processes and Grammars

1. Semi-Thue Processes

In this chapter we will see how the methods of computability theory can be used to deal with combinatorial problems involving substitution of one substring for another in a string.

Definition. Given a pair of words g, \bar{g} on some alphabet, the expression

$$g \rightarrow \bar{g}$$

is called a *semi-Thue production* or simply a *production*. The term *rewrite rule* is also used.

“Thue” is from Axel Thue, a Norwegian mathematician, and is pronounced too-ay.

If P is the semi-Thue production $g \rightarrow \bar{g}$, then we write

$$u \xrightarrow{P} v$$

to mean that there are (possibly null) words r, s such that

$$u = rgs \quad \text{and} \quad v = r\bar{g}s.$$

(In other words, v is obtained from u by a replacement of g by \bar{g} .)

Definition. A *semi-Thue process* is a finite set of semi-Thue productions.

If Π is a semi-Thue process, we write

$$u \xrightarrow{\Pi} v$$

to mean that

$$u \xrightarrow{P} v$$

for some production P which belongs to Π . Finally, we write

$$u \xrightarrow[\Pi]{}^* v$$

if there is a sequence

$$u = u_1 \xrightarrow[\Pi]{} u_2 \xrightarrow[\Pi]{} \cdots \xrightarrow[\Pi]{} u_n = v.$$

The sequence u_1, u_2, \dots, u_n is then called a *derivation of v from u* . In particular (taking $n = 1$)

$$u \xrightarrow[\Pi]{}^* u.$$

When no ambiguity results we often omit the explicit reference to Π , writing simply $u \Rightarrow v$ and $u \xrightarrow{*} v$.

Here is a simple example: We let $\Pi = \{ab \rightarrow aa, ba \rightarrow bb\}$. Then we have

$$aba \Rightarrow abb \Rightarrow aab \Rightarrow aaa.$$

Thus,

$$aba \xrightarrow{*} aaa,$$

and the sequence of words aba, abb, aab, aaa is a derivation of aaa from aba .

2. Simulation of Nondeterministic Turing Machines by Semi-Thue Processes

Let us begin with a nondeterministic Turing machine \mathcal{M} with alphabet $\{s_1, \dots, s_K\}$, and states q_1, q_2, \dots, q_n . We shall show how to simulate \mathcal{M} by a semi-Thue process $\Sigma(\mathcal{M})$ on the alphabet

$$s_0, s_1, \dots, s_K, q_0, q_1, q_2, \dots, q_n, q_{n+1}, h.$$

Each stage in a computation by \mathcal{M} is specified completely by the current configuration. We shall code each such stage by a word on the alphabet of $\Sigma(\mathcal{M})$. For example, the configuration

$$\begin{array}{ccccccccc} s_1 & s_1 & s_3 & s_2 & s_0 & s_1 & s_2 \\ & & & & \uparrow & & \\ & & & & q_4 & & \end{array}$$

will be represented by the single word

$$hs_1s_1s_3q_4s_2s_0s_1s_2h. \quad (2.1)$$

Note that h is used as a *beginning* and *end* marker, and the symbol q_4 indicates the state of \mathcal{M} and is placed immediately to the left of the scanned square. A

word like (2.1) will be called a *Post word*. Of course, the same configuration can be represented by infinitely many Post words because any number of additional blanks may be shown on the left or right. For example,

$$hs_0s_0s_1s_1s_3q_4s_2s_0s_1s_1s_0h$$

is a Post word representing the same configuration that (2.1) does.

In general, a word huq_ivh , where $0 \leq i \leq n + 1$, is called a *Post word* if u and v are words on the subalphabet $\{s_0, s_1, \dots, s_K\}$. We shall show how to associate suitable semi-Thue productions with each quadruple of \mathcal{M} ; the productions simulate the effect of that quadruple on Post words.

1. For each quadruple of \mathcal{M} of the form $q_i s_j s_k q_l$ we place in $\Sigma(\mathcal{M})$ the production

$$q_i s_j \rightarrow q_l s_k.$$

2. For each quadruple of \mathcal{M} of the form $q_i s_j R q_l$ we place in $\Sigma(\mathcal{M})$ the productions

$$q_i s_j s_k \rightarrow s_j q_l s_k, \quad k = 0, 1, \dots, K,$$

$$q_i s_j h \rightarrow s_j q_l s_0 h.$$

3. For each quadruple of \mathcal{M} of the form $q_i s_j L q_l$ we place in $\Sigma(\mathcal{M})$ the productions

$$s_k q_i s_j \rightarrow q_l s_k s_j, \quad k = 0, 1, \dots, K,$$

$$h q_i s_j \rightarrow h q_l s_0 s_j.$$

To see how these productions simulate the behavior of \mathcal{M} , suppose \mathcal{M} is in configuration

$$\begin{array}{cccc} s_2 & s_1 & s_0 & s_3. \\ & \uparrow & & \\ & q_4 & & \end{array}$$

This configuration is represented by the Post word

$$hs_2q_4s_1s_0s_3h.$$

Now suppose \mathcal{M} contains the quadruple

$$q_4 s_1 s_3 q_5.$$

Then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 \rightarrow q_5 s_3,$$

so that

$$hs_2q_4s_1s_0s_3h \xrightarrow{\Sigma(\mathcal{M})} hs_2q_5s_3s_0s_3h.$$

The Post word on the right then corresponds to the configuration immediately following application of the above quadruple. Now suppose that \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ R \ q_3.$$

(Of course, if \mathcal{M} is a *deterministic* Turing machine, it cannot contain both of these quadruples.) Then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 s_0 \rightarrow s_1 q_3 s_0,$$

so that

$$hs_2 q_4 s_1 s_0 s_3 h \xrightarrow{\Sigma(\mathcal{M})} hs_2 s_1 q_3 s_0 s_3 h.$$

Finally if \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ L \ q_2,$$

then $\Sigma(\mathcal{M})$ contains the production

$$s_2 q_4 s_1 \rightarrow q_2 s_2 s_1,$$

so that

$$hs_2 q_4 s_1 s_0 s_3 h \xrightarrow{\Sigma(\mathcal{M})} hq_2 s_2 s_1 s_0 s_3 h.$$

The productions involving h are to take care of cases where motion to the right or left would go past the part of the tape included in the Post word, so that an additional blank must be added. For example, if the configuration is

$$\begin{array}{ccc} s_2 & s_3 & s_1 \\ & \uparrow & \\ & q_4 & \end{array}$$

and \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ R \ q_3,$$

then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 h \rightarrow s_1 q_3 s_0 h$$

and we have

$$hs_2 s_3 q_4 s_1 h \xrightarrow{\Sigma(\mathcal{M})} hs_2 s_3 s_1 q_3 s_0 h,$$

so that the needed blank on the right has been inserted. The reader will readily verify that blanks on the left are similarly supplied when needed.

We now complete the specification of $\Sigma(\mathcal{M})$:

4. Whenever $q_i s_j (i = 1, \dots, n; j = 0, 1, \dots, K)$ are *not* the first two symbols of a quadruple of \mathcal{M} , we place in $\Sigma(\mathcal{M})$ the production

$$q_i s_j \rightarrow q_{n+1} s_j.$$

Thus, q_{n+1} serves as a “halt” state.

5. Finally, we place in $\Sigma(\mathcal{M})$ the productions

$$q_{n+1} s_i \rightarrow q_{n+1}, \quad i = 0, 1, \dots, K,$$

$$q_{n+1} h \rightarrow q_0 h$$

$$s_i q_0 \rightarrow q_0, \quad i = 0, 1, \dots, K.$$

We have

Theorem 2.1. Let \mathcal{M} be a *deterministic* Turing machine, and let w be a Post word on the alphabet of $\Sigma(\mathcal{M})$. Then

1. there is at most one word z such that $w \xrightarrow{\Sigma(\mathcal{M})} z$, and
2. if there is a word z satisfying (1), then z is a Post word.

Proof. We have $w = huq_i vh$.

If $1 \leq i \leq n$, then

- a. if $v = 0$ no production of $\Sigma(\mathcal{M})$ applies to w ;
- b. if v begins with the symbol s_j and there is a (necessarily unique) quadruple of \mathcal{M} which begins $q_i s_j$, then there is a uniquely applicable production of $\Sigma(\mathcal{M})$ and the result of applying it will be a Post word;
- c. if v begins with the symbol s_j and there is no quadruple of \mathcal{M} which begins $q_i s_j$, then the one applicable production of $\Sigma(\mathcal{M})$ is

$$q_i s_j \rightarrow q_{n+1} s_j,$$

which yields another Post word when applied to w .

If $i = n + 1$, then

- a. if $v = 0$, the only applicable production of $\Sigma(\mathcal{M})$ is

$$q_{n+1} h \rightarrow q_0 h,$$

which yields a Post word;

- b. if v begins with the symbol s_j , the only applicable production of $\Sigma(\mathcal{M})$ is

$$q_{n+1} s_j \rightarrow q_{n+1},$$

which again yields a Post word.

Finally, if $i = 0$, then

- a. if $u = 0$, no production of $\Sigma(\mathcal{M})$ can be applied;
- b. if u ends with s_j , the only applicable production of $\Sigma(\mathcal{M})$ is

$$s_j q_0 \rightarrow q_0,$$

which yields a Post word. ■

Our next result makes precise the sense in which $\Sigma(\mathcal{M})$ simulates \mathcal{M} .

Theorem 2.2. Let \mathcal{M} be a nondeterministic Turing machine. Then, for each string u on the alphabet of \mathcal{M} , \mathcal{M} accepts u if and only if

$$hq_1 s_0 u h \xrightarrow[\Sigma(\mathcal{M})]{} hq_0 h.$$

Proof. Let the alphabet of \mathcal{M} be s_1, \dots, s_k . First let us suppose that \mathcal{M} accepts u . Then, if \mathcal{M} begins in the configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

it will eventually reach a state q_i scanning a symbol s_k where no quadruple of \mathcal{M} begins $q_i s_k$. Then we will have (for appropriate words v, w on the alphabet of \mathcal{M})

$$\begin{aligned} hq_1 s_0 u h &\xrightarrow[\Sigma(\mathcal{M})]{} hvq_i s_k wh \xrightarrow[\Sigma(\mathcal{M})]{} hvq_{n+1} s_k wh \\ &\xrightarrow[\Sigma(\mathcal{M})]{} hvq_0 h \xrightarrow[\Sigma(\mathcal{M})]{} hq_0 h. \end{aligned}$$

Next suppose that \mathcal{M} does not accept u . Then, beginning with configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

\mathcal{M} will never halt. Let

$$w_1 = hq_1 s_0 u h,$$

and suppose that

$$w_1 \xrightarrow[\Sigma(\mathcal{M})]{} w_2 \xrightarrow[\Sigma(\mathcal{M})]{} w_3 \xrightarrow[\Sigma(\mathcal{M})]{} \cdots \xrightarrow[\Sigma(\mathcal{M})]{} w_m.$$

Then each w_j , $1 \leq j \leq m$, must contain a symbol q_i with $1 \leq i \leq n$. Hence there can be no derivation of a Post word containing q_0 from w_1 , and so, in particular, there is no derivation of $hq_0 h$ from w_1 . ■

Definition. The *inverse* of the production $g \rightarrow \bar{g}$ is the production $\bar{g} \rightarrow g$.

For example, the inverse of the production $ab \rightarrow aa$ is the production $aa \rightarrow ab$.

Let us write $\Omega(\mathcal{M})$ for the semi-Thue process which consists of the inverses of all the productions of $\Sigma(\mathcal{M})$. Then an immediate consequence of Theorem 2.2 is

Theorem 2.3. Let \mathcal{M} be a nondeterministic Turing machine. Then for each string u in the alphabet of \mathcal{M} , \mathcal{M} accepts u if and only if

$$hq_0 h \xrightarrow[\Omega(\mathcal{M})]{} hq_1 s_0 u h.$$

3. Unsolvable Word Problems

Definition. The *word problem* for a semi-Thue process Π is the problem of determining for any given pair u, v of words on the alphabet of Π whether $u \xrightarrow[\Pi]{*} v$.

We shall prove

Theorem 3.1. There is a Turing machine \mathcal{M} such that the word problem is unsolvable for both of the semi-Thue processes $\Sigma(\mathcal{M})$ and $\Omega(\mathcal{M})$.

Proof. By Theorem 3.1 in Chapter 6, there is a Turing machine \mathcal{M} (in fact, deterministic) which accepts a nonrecursive language. Suppose first that the word problem for $\Sigma(\mathcal{M})$ were solvable. Then there would be an algorithm for testing given words v, w to determine whether $v \xrightarrow[\Sigma(\mathcal{M})]{*} w$. By Theorem 2.2, we could use this algorithm to determine whether \mathcal{M} will accept a given word u by testing whether

$$hq_1 s_0 u h \xrightarrow[\Sigma(\mathcal{M})]{} hq_0 h.$$

We would thus have an algorithm for testing a given word u to see whether \mathcal{M} will accept it. But such an algorithm cannot exist since the language accepted by \mathcal{M} is not a recursive set.

Finally, an algorithm which solved the word problem for $\Omega(\mathcal{M})$ would also solve the word problem for $\Sigma(\mathcal{M})$, since

$$v \xrightarrow[\Sigma(\mathcal{M})]{*} w \quad \text{if and only if} \quad w \xrightarrow[\Omega(\mathcal{M})]{*} v. \quad \blacksquare$$

Definition. A semi-Thue process is called a *Thue process* if the inverse of each production in the process is also in the process.

The fact that Thue processes are in fact “two-way” processes is a curious coincidence.

We write $g \leftrightarrow \bar{g}$ to combine the production $g \rightarrow \bar{g}$ and its inverse $\bar{g} \rightarrow g$. For each Turing machine \mathcal{M} , we write

$$\Theta(\mathcal{M}) = \Sigma(\mathcal{M}) \cup \Omega(\mathcal{M}),$$

so that $\Theta(\mathcal{M})$ is a Thue process. We have

Theorem 3.2 (Post's Lemma). Let \mathcal{M} be a deterministic Turing machine. Let u be a word on the alphabet of \mathcal{M} such that

$$hq_1 s_0 u h \xrightarrow[\Theta(\mathcal{M})]{} hq_0 h.$$

Then

$$hq_1 s_0 u h \xrightarrow[\Sigma(\mathcal{M})]{} hq_0 h.$$

Proof. Let the sequence

$$hq_1 s_0 u h = w_1, w_2, \dots, w_l = hq_0 h$$

be a derivation in $\Theta(\mathcal{M})$. Since w_1 is a Post word, and each production of $\Theta(\mathcal{M})$ transforms Post words into Post words, we can conclude that the entire derivation consists of Post words. We need to show how to eliminate use of productions belonging to $\Omega(\mathcal{M})$ from this derivation. So let us assume that the last time in the derivation that a production of $\Omega(\mathcal{M})$ was used was in getting from w_i to w_{i+1} . That is, we assume

$$w_i \xrightarrow[\Omega(\mathcal{M})]{} w_{i+1}; \quad w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_{i+2} \xrightarrow[\Sigma(\mathcal{M})]{} \dots \xrightarrow[\Sigma(\mathcal{M})]{} w_l = hq_0 h.$$

Now, $\Omega(\mathcal{M})$ consists of inverses of productions of $\Sigma(\mathcal{M})$; hence we must have

$$w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_i.$$

Moreover, we must have $i+1 < l$ because no production of $\Sigma(\mathcal{M})$ can be applied to $w_l = hq_0 h$. Now, w_{i+1} is a Post word and

$$w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_i, \quad w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_{i+2}.$$

By Theorem 2.1, we conclude that $w_{i+2} = w_i$. Thus the transition from w_i to w_{i+1} and then back to $w_{i+2} = w_i$ is clearly an unnecessary detour. That is, the sequence

$$w_1, w_2, \dots, w_i, w_{i+3}, \dots, w_l$$

from which w_{i+1}, w_{i+2} have been omitted is a derivation in $\Theta(\mathcal{M})$.

We have shown that any derivation which uses a production belonging to $\Omega(\mathcal{M})$ can be shortened. Continuing this procedure, we eventually obtain a derivation using only productions of $\Sigma(\mathcal{M})$. ■

Theorem 3.3 (Post–Markov). If the deterministic Turing machine \mathcal{M} accepts a nonrecursive set, then the word problem for the Thue process $\Theta(\mathcal{M})$ is unsolvable.

Proof. Let u be a word on the alphabet of \mathcal{M} . Then we have, using Theorems 2.2 and 3.2,

$$\mathcal{M} \text{ accepts } u$$

if and only if

$$hq_1s_0uh \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h$$

if and only if

$$hq_1s_0uh \xrightarrow[\Theta(\mathcal{M})]{*} hq_0h.$$

Hence, an algorithm for solving the word problem for $\Theta(\mathcal{M})$ could be used to determine whether or not \mathcal{M} will accept u , which is impossible. ■

Now we consider semi-Thue processes on an alphabet of two symbols.

Theorem 3.4. There is a semi-Thue process on the alphabet $\{a, b\}$ whose word problem is unsolvable. Moreover, for each production $g \rightarrow h$ of this semi-Thue process, $g, h \neq 0$.

Proof. Let us begin with a semi-Thue process Π on the alphabet $A = \{a_1, \dots, a_n\}$ and with productions

$$g_i \rightarrow \bar{g}_i, \quad i = 1, 2, \dots, m,$$

whose word problem is unsolvable. We also assume that for each $i = 1, 2, \dots, m$, $g_i \neq 0$ and $\bar{g}_i \neq 0$. This is legitimate because this condition is satisfied by the productions of $\Sigma(\mathcal{M})$.

We write

$$a'_j = ba^{[j]}b, \quad j = 1, 2, \dots, n,$$

where there is a string of a 's of length j between the two b 's. Finally, for any word $w \neq 0$ in A^* ,

$$w = a_{j_1}a_{j_2}\cdots a_{j_k},$$

we write

$$w' = a'_{j_1}a'_{j_2}\cdots a'_{j_k}.$$

In addition we let $0' = 0$. Then, we consider the semi-Thue process Π' on the alphabet $\{a, b\}$ whose productions are

$$g'_i \rightarrow \bar{g}'_i, \quad i = 1, 2, \dots, m.$$

We have

Lemma 1. If $u \xrightarrow{\Pi} v$, then $u' \xrightarrow{\Pi'} v'$.

Proof. We have $u = rg_i s$, $v = r\bar{g}_i s$. Hence $u' = r'g'_i s'$, $v' = r'\bar{g}'_i s'$, so that $u' \xrightarrow{\Pi'} v'$. ■

Lemma 2. If $u' \xrightarrow{\Pi'} w$, then for some $v \in A^*$ we have $w = v'$ and $u \xrightarrow{\Pi} v$.

Proof. We have $u' = pg'_i q$, $w = p\bar{g}'_i q$. Now, since $g_i \neq 0$, g'_i begins and ends with the letter b . Hence each of p and q either begins and ends with b or is 0, so that $p = r'$, $q = s'$. Then, $u = rg_i s$. Let $v = r\bar{g}_i s$. Then $w = v'$ and $u \xrightarrow{\Pi} v$. ■

Lemma 3. $u \xrightarrow{*_{\Pi}} v$ if and only if $u' \xrightarrow{*_{\Pi'}} v'$.

Proof. If $u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_n = v$, then by Lemma 1

$$u' = u'_1 \xrightarrow{\Pi'} u'_2 \xrightarrow{\Pi'} \cdots \xrightarrow{\Pi'} u'_n = v'.$$

Conversely, if

$$u' = w_1 \xrightarrow{\Pi'} w_2 \xrightarrow{\Pi'} \cdots \xrightarrow{\Pi'} w_n = v',$$

then by Lemma 2, for each w_i there is a string $u_i \in A^*$ such that $w_i = u'_i$. Thus,

$$u' = u'_1 \xrightarrow{\Pi'} u'_2 \xrightarrow{\Pi'} \cdots \xrightarrow{\Pi'} u'_n = v'.$$

By Lemma 2 once again,

$$u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_n = v,$$

so that $u \xrightarrow{*_{\Pi}} v$. ■

Proof of Theorem 3.4 Concluded. By Lemma 3, if the word problem for Π' were solvable, the word problem for Π would also be solvable. Hence, the word problem for Π' is unsolvable. ■

In the above proof it is clear that if the semi-Thue process Π with which we begin is actually a Thue process, then Π' will be a Thue process on $\{a, b\}$. We conclude:

Theorem 3.5. There is a Thue process on the alphabet $\{a, b\}$ whose word problem is unsolvable. Moreover, for each production $g \rightarrow h$ of this Thue process, $g, h \neq 0$.

4. Post's Correspondence Problem

The Post correspondence problem first appeared in a paper by Emil Post in 1946. It was only much later that this problem was seen to have important applications in the theory of formal languages.

Our treatment of the Post correspondence problem is a simplification of a proof due to Floyd, itself much simpler than Post's original work.

The correspondence problem may conveniently be thought of as a solitaire game played with special sets of dominoes. Each domino has a word (on some given alphabet) appearing on each half. A typical domino is shown in Fig. 4.1. A *Post correspondence system* is simply a finite set of

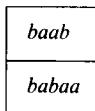


Fig. 4.1

dominoes of this kind. Figure 4.2 gives a simple example of a Post correspondence system using 3 dominoes and the alphabet $\{a, b\}$. Each move in the solitaire game defined by a particular Post correspondence system consists of simply placing one of the dominoes of the system to the right of the

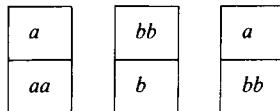


Fig. 4.2

dominoes laid down on previous moves. The key fact is that the *dominoes are not used up by being played, so that each one can be used any number of times*. The way to “win” the game is to reach a situation where the very same word appears on the top halves as on the bottom halves of the dominoes when we read across from left to right. Figure 4.3 shows how to win the game

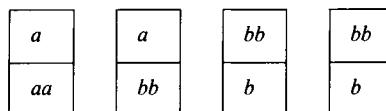


Fig. 4.3

defined by the dominoes of Fig. 4.2. (Note that one of the dominoes is used twice.) The word $aabbba$ which appears across both the top halves and the bottom halves is called a *solution* of the given Post correspondence system. Thus a Post correspondence system possesses a *solution* if and only if it is possible to win the game defined by that system.

We shall prove

Theorem 4.1. There is no algorithm which can test a given Post correspondence system to determine whether it has a solution.

Proof. Using Theorem 3.4, we begin with a semi-Thue process Π on the alphabet $\{a, b\}$ whose word problem is unsolvable. We modify Π in the following trivial way: we add to the productions of Π the two productions

$$a \rightarrow a, \quad b \rightarrow b.$$

Naturally this addition has no effect on whether

$$u \xrightarrow[\Pi]{*} v$$

for given words u, v . However, it does guarantee that whenever $u \xrightarrow[\Pi]{*} v$, there is a derivation

$$u = u_1 \xrightarrow[\Pi]{} u_2 \xrightarrow[\Pi]{} \cdots \xrightarrow[\Pi]{} u_m = v,$$

where m is an odd number. This is because with the added productions we have

$$u_i \xrightarrow[\Pi]{} u_i$$

for each i , so that any step in a derivation (e.g., the first) can be repeated if necessary to change the length of the derivation from an even to an odd number.

Let u and v be any given words on the alphabet $\{a, b\}$. We shall construct a Post correspondence system $P_{u,v}$ (which depends on Π as well as on the words u and v) such that $P_{u,v}$ has a solution if and only if $u \xrightarrow[\Pi]{*} v$. Once we have obtained this $P_{u,v}$ we are through. For, if there were an algorithm for testing given Post correspondence systems for possessing a solution, this algorithm could be applied in particular to $P_{u,v}$ and therefore to determine whether $u \xrightarrow[\Pi]{*} v$; since Π has an unsolvable word problem, this is impossible.

We proceed to show how to construct $P_{u,v}$. Let the productions of Π (including the two we have just added) be $g_i \rightarrow h_i$, $i = 1, 2, \dots, n$. The alphabet of $P_{u,v}$ consists of the eight symbols

$$a \ b \ \tilde{a} \ \tilde{b} \ [\] * \ \tilde{*}.$$

For any word w on $\{a, b\}$, we write \tilde{w} for the word on $\{\tilde{a}, \tilde{b}\}$ obtained by placing “~” on top of each symbol of w . $P_{u,v}$ is then to consist of the $2n + 4$

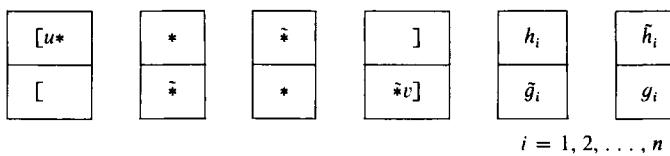
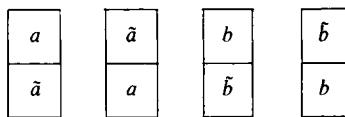
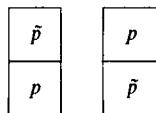


Fig. 4.4

dominoes shown in Fig. 4.4. Note that because Π contains the productions $a \rightarrow a$ and $b \rightarrow b$, $P_{u,v}$ contains the four dominoes



Therefore, it is clear that in our play it is legitimate to use dominoes of the form



where p is any word on $\{a, b\}$, since any such dominoes can be assembled by lining up single dominoes selected appropriately from the four above.

We proceed to show that $P_{u,v}$ has a solution if and only if $u \xrightarrow[\Pi]{} v$.

First suppose that $u \xrightarrow[\Pi]{} v$. Let

$$u = u_1 \xrightarrow[\Pi]{} u_2 \xrightarrow[\Pi]{} \cdots \xrightarrow[\Pi]{} u_m = v,$$

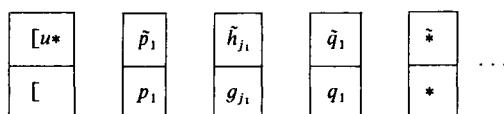
where m is an odd number. Thus, for each i , $1 \leq i < m$, we can write

$$u_i = p_i g_{j_i} q_i, \quad u_{i+1} = p_i h_{j_i} q_i,$$

where the transition from u_i to u_{i+1} is via the j_i th production of Π . Then we claim that the word

$$[u_1 * \tilde{u}_2 \tilde{*} u_3 * \cdots * \tilde{u}_{m-1} \tilde{*} u_m] \tag{4.1}$$

is a solution of $P_{u,v}$. To see this, let us begin to play by laying down the dominoes:



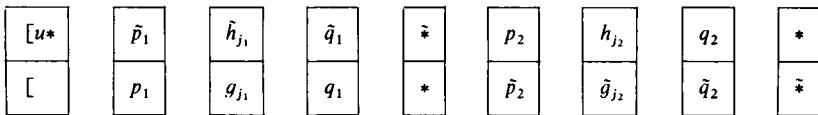
At this stage, the word on top is

$$[u_1 * \tilde{u}_2 * \tilde{*}$$

while the word on the bottom is

$$[u_1 * .$$

We can continue to play as follows:



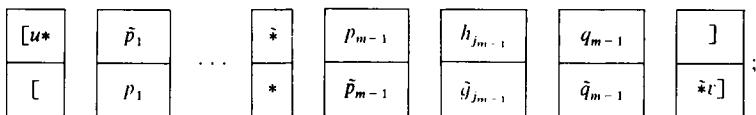
Now the word on top is

$$[u_1 * \tilde{u}_2 * \tilde{u}_3 * \tilde{*}$$

and the word on the bottom is

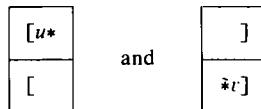
$$[u_1 * \tilde{u}_2 * *$$

Recalling that m is an odd number we see that we can win by continuing as follows:



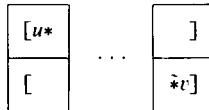
for, at this point the word both on top and on bottom is (4.1).

Conversely suppose that $P_{u,v}$ has a solution w . Examining Fig. 4.4, we see that the only possible way to win involves playing



first and last, respectively. This is because none of the other dominoes in $P_{u,v}$ have tops and bottoms which begin (or end) with the same symbol. Thus, w must begin with [and end with]. Let us write $w = [z]y$, where z contains no]. (Of course it is quite possible that $y = 0$.) Since the only domino containing] contains it on the far right on top and on bottom, we

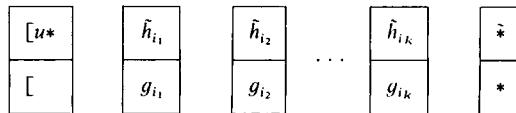
see that $[z]$ itself is already a solution to $P_{u,v}$. We work with this solution. So far we know that the game looks like this:



so that the solution $[z]$ looks like this:

$$[u * \dots \tilde{*} v].$$

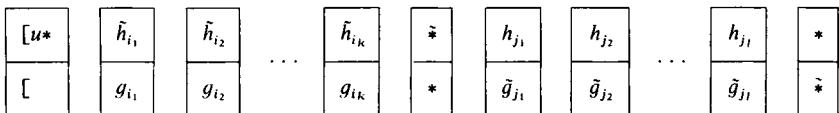
Continuing from the left we see that the play must go



where $g_{i_1}g_{i_2}\dots g_{i_k} = u$. (This is necessary in order for the bottom to “catch up” with the $u*$ which is already on top.) Writing $u = u_1$ and $u_2 = h_{i_1}h_{i_2}\dots h_{i_k}$ we see that $u_1 \xrightarrow{*} u_2$ and that the solution has the form

$$[u_1 * \tilde{u}_2 * \dots \tilde{*} v].$$

Now we see how the play must continue:



where of course $u_2 = g_{j_1}g_{j_2}\dots g_{j_l}$. Again, writing $u_3 = h_{j_1}h_{j_2}\dots h_{j_l}$ we have that $u_2 \xrightarrow{*} u_3$ and that the solution has the form

$$[u_1 * \tilde{u}_2 * u_3 * \dots \tilde{*} v].$$

Continuing, it is clear that the solution can be written

$$[u_1 * \tilde{u}_2 * u_3 * \dots * \tilde{u}_{m-1} * \tilde{*} u_m],$$

where

$$u = u_1 \xrightarrow{*} u_2 \xrightarrow{*} u_3 \xrightarrow{*} \dots \xrightarrow{*} u_{m-1} \xrightarrow{*} u_m = v,$$

so that $u \xrightarrow{*} v$. ■

5. Grammars

A *phrase-structure grammar* or simply a *grammar* is just a semi-Thue process in which the letters of the alphabet are separated into two disjoint sets called the *variables* and the *terminals*, with one of the variables singled out as the *start symbol*. It is customary (but, of course, not necessary) to use lower case letters for terminals, capital letters for variables, and in particular the letter S for the start symbol.

Let Γ be a grammar with start symbol S and let V, T be the sets of variables and terminals of Γ , respectively. Then we define

$$L(\Gamma) = \{u \in T^* \mid S \xrightarrow{\Gamma} u\},$$

and call $L(\Gamma)$ the *language generated by Γ* . Our purpose in this section is to characterize languages which can be generated by grammars.

We first prove

Theorem 5.1. Let U be a language accepted by a nondeterministic Turing machine. Then there is a grammar Γ such that $U = L(\Gamma)$.

Proof. Let $U \subseteq T^*$ and let \mathcal{M} be a nondeterministic Turing machine which accepts U . We will construct Γ by modifying the semi-Thue process $\Omega(\mathcal{M})$ from Section 2. Let \mathcal{M} have the states q_1, \dots, q_n . Then we recall that the alphabet of $\Omega(\mathcal{M})$ [which is the same as that of $\Sigma(\mathcal{M})$] consists of $s_0, q_0, q_1, q_2, \dots, q_n, q_{n+1}, h$ in addition to the letters of the alphabet of \mathcal{M} . We let the *terminals* of Γ be just the letters of T , and the variables of Γ be the symbols from the alphabet of $\Omega(\mathcal{M})$ not in T , together with the two additional symbols S and q . S is to be the start symbol of Γ . The productions of Γ are then the productions of $\Omega(\mathcal{M})$ together with the productions

$$\begin{aligned} S &\rightarrow hq_0h \\ hq_1s_0 &\rightarrow q \\ qs &\rightarrow sq \quad \text{for each } s \in T \\ qh &\rightarrow 0. \end{aligned}$$

Now, let \mathcal{M} accept $u \in T^*$. Then, using Theorem 2.3, we have

$$S \xrightarrow{\Gamma} hq_0h \xrightarrow[\Gamma]{} hq_1s_0uh \xrightarrow[\Gamma]{} quh \xrightarrow[\Gamma]{} uqh \Rightarrow u,$$

so that $u \in L(\Gamma)$.

Conversely, let $u \in L(\Gamma)$. Then $u \in T^*$ and $S \xrightarrow{\Gamma} u$. Examining the list of productions of Γ , we see that we must in fact have

$$S \xrightarrow{\Gamma} hq_0h \xrightarrow[\Gamma]{} vqhz \xrightarrow[\Gamma]{} vz = u.$$

Proceeding further, we see that the symbol q could only be introduced using the production

$$hq_1s_0 \rightarrow q.$$

Hence, our derivation must have the form

$$\frac{S \Rightarrow hq_0 h}{\Gamma} \stackrel{*}{\Rightarrow} xhq_1s_0 yhz \frac{\Rightarrow}{\Gamma} xqyhz \stackrel{*}{\Rightarrow} xyqhz \frac{\Rightarrow}{\Gamma} xyz = u,$$

where of course $xy = v$. Thus, there is a derivation of $xhq_1s_0 yhz$ from $hq_0 h$ in Γ . Moreover, this must actually be a derivation in $\Omega(\mathcal{M})$ since the added productions are clearly inapplicable. Moreover, the productions of $\Omega(\mathcal{M})$ always lead from Post words to Post words. Hence, $xhq_1s_0 yhz$ must be a Post word. That is, $x = z = 0$ and $u = xyz = y$. We conclude that

$$hq_0 h \xrightarrow[\Omega(\mathcal{M})]{*} hq_1s_0 uh.$$

Thus by Theorem 2.3, \mathcal{M} accepts u . ■

Now, let us begin with a grammar Γ and see what we can say about $L(\Gamma)$. Thus, let the alphabet of Γ be

$$\{s_1, \dots, s_n, V_1, \dots, V_k\},$$

where $T = \{s_1, \dots, s_n\}$ is the set of terminals, V_1, \dots, V_k are the variables, and $S = V_1$ is the start symbol. Let us order the alphabet of Γ as shown. Thus strings on this alphabet are notations for integers in the base $n + k$. We have

Lemma 1. The predicate $u \xrightarrow{\Gamma} v$ is primitive recursive.

Proof. Let the productions of Γ be $g_i \rightarrow h_i$, $i = 1, 2, \dots, l$. We write, for $i = 1, 2, \dots, l$,

$$\text{PROD}_i(u, v) \Leftrightarrow (\exists r, s)_{\leq u} [u = \text{CONCAT}(r, g_i, s) \& v = \text{CONCAT}(r, h_i, s)].$$

Since, by Chapter 5, Section 1, CONCAT is primitive recursive, each of the predicates PROD_i is primitive recursive. But

$$u \xrightarrow{\Gamma} v \Leftrightarrow \text{PROD}_1(u, v) \vee \text{PROD}_2(u, v) \vee \dots \vee \text{PROD}_l(u, v),$$

and the result follows. ■

We write $\text{DERIV}(u, y)$ to mean that for some m , $y = [u_1, \dots, u_m, 1]$, where the sequence u_1, \dots, u_m is a derivation of u from S in Γ . (The “1” has been added to avoid complications in case $u_m = u = 0$.) Then, since the

value of S in base $n + k$ is $n + 1$ [because $S = V_1$ is the $(n + 1)$ th symbol in our alphabet], we have

$$\begin{aligned} \text{DERIV}(u, y) \Leftrightarrow & (\exists m)_{\leq y} (m + 1 = Lt(y) \& (y)_1 = n + 1 \\ & \& (y)_m = u \& (y)_{m+1} = 1 \\ & \& (\forall j)_{<m} \{j = 0 \vee [(y)_j \stackrel{\Gamma}{\Rightarrow} (y)_{j+1}]\}). \end{aligned}$$

Using Lemma 1, we have proved

Lemma 2. $\text{DERIV}(u, y)$ is primitive recursive.

Also, by definition of $\text{DERIV}(u, y)$, we have for every word u on the alphabet of Γ

$$S \stackrel{\Gamma}{\Rightarrow} u \Leftrightarrow (\exists y) \text{DERIV}(u, y). \quad (5.1)$$

Finally, (5.1) shows that

$$S \stackrel{\Gamma}{\Rightarrow} u \Leftrightarrow \min_y \text{DERIV}(u, y) \downarrow.$$

Hence, by Lemma 2 and Theorem 7.2 in Chapter 3, we see that $\{u | S \stackrel{\Gamma}{\Rightarrow} u\}$ is r.e. But

$$L(\Gamma) = T^* \cap \{u | S \stackrel{\Gamma}{\Rightarrow} u\} \quad (5.2)$$

(T is the alphabet of terminals of Γ), so that $L(\Gamma)$ is the intersection of two r.e. sets and hence is r.e. Combining this result with Theorem 5.1 in Chapter 6 and Theorem 5.1 in this chapter, we have

Theorem 5.2. A language U is r.e. if and only if there is a grammar Γ such that $U = L(\Gamma)$.

We now are able to obtain easily the promised converse to Theorem 5.1 in Chapter 6. In fact putting Theorem 3.1 in Chapter 6 and Theorems 5.1 and 5.2 in this chapter all together, we have

Theorem 5.3. Let L be a given language. Then the following conditions are all equivalent:

1. L is r.e.;
2. L is accepted by a deterministic Turing machine;
3. L is accepted by a nondeterministic Turing machine;
4. there is a grammar Γ such that $L = L(\Gamma)$.

Theorem 5.3 involves some of the main concerns of theoretical computer science: on the one hand, the relation between grammars, the languages they generate, and the devices which accept them; on the other hand, the relation, for various devices, between determinism and nondeterminism.

We next obtain a general result about r.e. sets of numbers.

Theorem 5.4. Let U be an r.e. set of numbers. Then there is a primitive recursive predicate $R(x, t)$ such that

$$U = \{x | (\exists t)R(x, t)\}.$$

Proof. We think of U as an r.e. language on the alphabet $\{s_1\}$. (That is, we are using unary notation for numbers.) By Theorem 5.2, there is a grammar Γ with the single terminal s_1 such that $U = L(\Gamma)$. Letting the alphabet of Γ be s_1, V_1, \dots, V_k , we note that the string $s_1^{[x]}$, which represents x in base 1, represents $f(x)$ in base $(k + 1)$, where

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ \sum_{i=0}^{x-1} (k+1)^i & \text{otherwise,} \end{cases}$$

so that $f(x)$ is primitive recursive. Hence, by (5.1)

$$x \in U \Leftrightarrow (\exists y)\text{DERIV}(f(x), y),$$

and the result follows from Lemma 2. ■

We will conclude this section by obtaining a result that will be needed in Chapter 10, but can easily be proved at this point.

Definition. A grammar Γ is called *context-sensitive* if for each production $g \rightarrow h$ of Γ we have $|g| \leq |h|$.

Lemma 3. If Γ is context-sensitive, then

$$\{u | S \xrightarrow{\Pi} u\}$$

is recursive.

Proof. It will suffice to obtain a recursive bound for y in formula (5.1). Since

$$1 = |u_1| \leq |u_2| \leq \dots \leq |u_m| = |u|$$

for any derivation u_1, \dots, u_m of u from S in the context-sensitive grammar Γ , we must have

$$u_1, u_2, \dots, u_m \leq g(u),$$

where $g(u)$ is the smallest number which represents a string of length $|u| + 1$ in base $n + k$. Now, since $g(u)$ is simply the value in base $n + k$ of a string consisting of $|u| + 1$ repetitions of 1, we have

$$g(u) = \sum_{i=0}^{|u|} (n+k)^i,$$

which is primitive recursive because $|u|$ is primitive recursive. Next, note that we may assume that the derivation

$$S = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_m = u$$

contains no repetitions. This is because given a sequence of steps

$$z = u_i \Rightarrow u_{i+1} \Rightarrow \cdots \Rightarrow u_{i+l} = z,$$

we could simply eliminate the steps u_{i+1}, \dots, u_{i+l} . Hence the length m of the derivation is bounded by the total number of distinct strings of length $\leq |u|$ on our alphabet of $n + k$ symbols. But this number is just $g(u)$. Hence,

$$[u_1, \dots, u_m, 1] = \prod_{i=1}^m p_i^{u_i} \cdot p_{m+1} \leq h(u),$$

where we have written $h(u)$ for the primitive recursive function defined by

$$h(u) = \prod_{i=1}^{g(u)} p_i^{g(u)} \cdot p_{g(u)+1}.$$

Finally, we have

$$S \xrightarrow[\Pi]{*} u \Leftrightarrow (\exists y)_{\leq h(u)} \text{DERIV}(u, y),$$

which gives the result. ■

Theorem 5.5. If Γ is a context-sensitive grammar, then $L(\Gamma)$ is recursive.

Proof. We will use Lemma 3 and Eq. (5.2). Since T^* is a recursive set, the result follows at once. ■

6. Some Unsolvable Problems Concerning Grammars

How much information can we hope to obtain about $L(\Gamma)$ by a computation that uses the grammar Γ as input? Not much at all, as we shall see.

Let \mathcal{M} be a Turing machine and let u be some given word on the alphabet of \mathcal{M} . We shall construct a grammar Γ_u as follows:

The variables of Γ_u are the *entire* alphabet of $\Sigma(\mathcal{M})$ together with S (the start symbol) and V . There is just one terminal, namely, a . The productions of Γ_u are all of the productions of $\Sigma(\mathcal{M})$ together with

$$S \rightarrow hq_1s_0uh$$

$$hq_0h \rightarrow V$$

$$V \rightarrow aV$$

$$V \rightarrow a.$$

Then it follows at once from Theorems 2.1 and 2.2 that $S \xrightarrow{\Gamma_u} V$ if and only if \mathcal{M} accepts u . Thus we have

Lemma. If \mathcal{M} accepts u , then $L(\Gamma_u) = \{a^{[i]} \mid i \neq 0\}$. If \mathcal{M} does not accept u , then $L(\Gamma_u) = \emptyset$.

Now we can select \mathcal{M} so that the language it accepts is not recursive. Then there is no algorithm for determining for given u whether \mathcal{M} accepts u . But the lemma obviously implies the equivalences

$$\begin{aligned}\mathcal{M} \text{ accepts } u &\Leftrightarrow L(\Gamma_u) \neq \emptyset \\ &\Leftrightarrow L(\Gamma_u) \text{ is infinite} \\ &\Leftrightarrow a \in L(\Gamma_u).\end{aligned}$$

We have obtained

Theorem 6.1. There is no algorithm to determine of a given grammar Γ whether

1. $L(\Gamma) = \emptyset$,
2. $L(\Gamma)$ is infinite, or
3. $v_0 \in L(\Gamma)$ for a fixed word v_0 .

We can also prove

Theorem 6.2. There is no algorithm for determining of a given pair Γ, Δ of grammars whether

1. $L(\Delta) \subseteq L(\Gamma)$,
2. $L(\Delta) = L(\Gamma)$.

Proof. Let Δ be the grammar with the single variable S , the single terminal a , and the productions

$$S \rightarrow aS$$

$$S \rightarrow a.$$

Then $L(\Delta) = \{a^{[i]} \mid i \neq 0\}$. Thus we have by the above lemma

$$\mathcal{M} \text{ accepts } u \Leftrightarrow L(\Delta) = L(\Gamma_u) \Leftrightarrow L(\Delta) \subseteq L(\Gamma_u).$$

The result follows at once. ■

7. Recursion and Minimalization

Theorem 5.4 is an important result which has many applications. In the logical development of our subject, it really belonged in Chapter 4. We chose to wait until this point because we could give such an easy proof.

We begin with some theorems about r.e. sets.

Theorem 7.1. Let S be a nonempty r.e. set. Then there is a primitive recursive function $f(u)$ such that $S = \{f(n) | n \in N\} = \{f(0), f(1), f(2), \dots\}$. That is, S is the range of f .

Proof. By Theorem 5.4

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

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 7.2. Let $f(x)$ be a partially computable function and let $S = \{f(x) | f(x) \downarrow\}$. (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

$$S = \{x | g(x) \downarrow\},$$

it suffices to show that $g(x)$ is partially computable. Let \mathcal{P} be a program which computes f and let $\#(\mathcal{P}) = p$. Then the following program computes $g(x)$:

```
[A]   IF ~STP(1)(Z, p, T) GOTO B
      V ← f(Z)
      IF V = X GOTO E
[B]   Z ← Z + 1
      IF Z ≤ T GOTO A
      T ← T + 1
      Z ← 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 7.1 and 7.2, we have

Theorem 7.3. Suppose that $S \neq \emptyset$. 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 7.1, (1) implies (2). Obviously, (2) implies (3), and (3) implies (4). By Theorem 7.2, (4) implies (1). Hence all four statements are equivalent. ■

Theorem 7.3 provides the motivation for the term “recursively enumerable.” In fact, such a set (if it is nonempty) is enumerated by a recursive function.

Theorem 7.4. Let $g(x)$ be a partially computable function and let $S = \{\langle x, y \rangle \mid y = g(x)\}$. Then S is r.e.

Proof. Let $f(x) = \langle x, g(x) \rangle$. Then f is clearly partially computable and its range is S . Thus the result follows at once from Theorem 7.2. ■

Theorem 7.5. Let $f(x)$ be a partially computable function. Then there is a primitive recursive predicate $R(x, y)$ such that

$$f(x) = l\left(\min_y R(x, y)\right).$$

Proof. Let $S = \{\langle x, y \rangle \mid y = f(x)\}$. By Theorems 5.4 and 7.4 there is a primitive recursive predicate $Q(u, t)$ such that

$$S = \{u \mid (\exists t)Q(u, t)\}.$$

Thus,

$$y = f(x) \Leftrightarrow \langle x, y \rangle \in S \Leftrightarrow (\exists t)Q(\langle x, y \rangle, t).$$

We claim that

$$f(x) = l\left(\min_y Q(\langle x, l(y) \rangle, r(y))\right),$$

which will give the desired result.

To verify the claim, suppose first that

$$y_0 = \min_y Q(\langle x, l(y) \rangle, r(y))$$

is defined, and let

$$v = l(y_0).$$

Then, setting $t = r(y_0)$, we have,

$$Q(\langle x, v \rangle, t),$$

so that $v = f(x)$.

If, on the other hand,

$$\min_y Q(\langle x, l(y) \rangle, r(y)) \uparrow,$$

then $Q(\langle x, v \rangle, t)$ is false for all v, t and hence $f(x) = v$ is false for all v . Thus $f(x) \uparrow$. So, our claim is correct and the proof is complete. ■

Theorem 7.6 (Normal Form Theorem). Let $f(x_1, \dots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \dots, x_n, y)$ such that

$$f(x_1, \dots, x_n) = l\left(\min_y R(x_1, \dots, x_n, y)\right).$$

Proof. The idea of the proof is to use Gödel numbers to code $f(x_1, \dots, x_n)$ by a function of one variable, and then to use the previous theorem.

Thus, let

$$g(x) = f((x)_1, \dots, (x)_n).$$

Since g is clearly partially computable, by Theorem 7.5 there is a primitive recursive predicate $R(x, y)$ such that

$$g(x) = l\left(\min_y R(x, y)\right).$$

Then

$$\begin{aligned} f(x_1, \dots, x_n) &= g([x_1, \dots, x_n]) \\ &= l\left(\min_y R([x_1, \dots, x_n], y)\right). \end{aligned}$$

We recall from Chapter 3, Section 3, that we call the functions $s(x) = x + 1$, $n(x) = 0$, and $u_i^n(x_1, \dots, x_n) = x_i$, *initial functions*. The normal form theorem just proved leads to a new characterization of the class of partially computable functions.

Theorem 7.7. 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.

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

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

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 l . ■

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

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

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

Theorem 7.8. 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.

*8. Normal Processes

Given a pair of words g and \bar{g} we write

$$gz \rightarrow z\bar{g}$$

to indicate a kind of transformation on strings called a *normal production*. If P is the normal production $gz \rightarrow z\bar{g}$ we write

$$u \xrightarrow{P} v$$

if for some string z we have

$$u = gz, \quad v = z\bar{g}.$$

That is, v can be obtained from u by crossing off g from the left of u and adjoining \bar{g} to the right. A *normal process* is simply a finite set of normal productions. If v is a normal process, we write

$$u \xrightarrow{v}$$

to mean that

$$u \xrightarrow{P} v$$

for some production P in v . Finally, we write

$$u \xrightarrow{*} v$$

to mean that there is a sequence (called a derivation)

$$u = u_1 \xrightarrow{v} u_2 \xrightarrow{v} \cdots \xrightarrow{v} u_m = v.$$

The *word problem* for v is the problem of determining of two given words u, v whether $u \xrightarrow{*} v$.

Let Π be a semi-Thue process on the alphabet $\{a, b\}$ with an unsolvable word problem. We shall show how to simulate Π by a normal process v on the alphabet $\{a, b, \tilde{a}, \tilde{b}\}$. As above, if $u \in \{a, b\}^*$, we write \tilde{u} for the word on $\{\tilde{a}, \tilde{b}\}$ obtained by placing \sim above each letter in u . Let the productions of Π be

$$g_i \rightarrow h_i, \quad i = 1, 2, \dots, n.$$

Then the productions of v will be

$$\begin{aligned} g_i z &\rightarrow z \tilde{h}_i \quad . \quad i = 1, 2, \dots, n \\ az &\rightarrow z \tilde{a} \\ bz &\rightarrow z \tilde{b} \\ \tilde{a}z &\rightarrow za \\ \tilde{b}z &\rightarrow zb. \end{aligned}$$

A word on $\{a, b, \tilde{a}, \tilde{b}\}$ is called *proper* if it can be written in one of the forms $u\tilde{v}$ or $\tilde{u}v$, where u, v are words on $\{a, b\}$. We say that two words are *associates* if there is a derivation of one from the other *using only the last four productions* of v . A word on $\{a, b\}$ of length n has $2n$ associates, all of which are proper. For example, the associates of $baab$ are as follows:

$$baab \Rightarrow aabb \Rightarrow abb\tilde{a} \Rightarrow b\tilde{b}\tilde{a}\tilde{a} \Rightarrow \tilde{b}\tilde{a}\tilde{a}\tilde{b} \Rightarrow \tilde{a}\tilde{a}\tilde{b}b \Rightarrow \tilde{a}\tilde{b}ba \Rightarrow \tilde{b}baa \Rightarrow baab.$$

Generally for $u, v \in \{a, b\}^*$, the proper words $u\tilde{v}$ and $\tilde{u}v$ are associates of each other and also of the word vu . In fact, vu is the unique word on $\{a, b\}$ which is an associate of $u\tilde{v}$. Thus, a word is proper just in case it is an associate of a word on $\{a, b\}$.

Lemma 1. If $u \xrightarrow{\Pi} v$, then $u \xrightarrow{*} v$.

Proof. We have $u = pg_iq$, $v = ph_iq$ for some i . Then

$$u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} g_i q \tilde{p} \Rightarrow q \tilde{p} \tilde{h}_i \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} p h_i q.$$
■

Lemma 2. If $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$, then $u \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$.

Proof. Immediate from Lemma 1. ■

Lemma 3. Let u be proper and let $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$. Then there are words r, s on $\{a, b\}$ which are associates of u, v , respectively, such that $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$.

Proof. If v is an associate of u , then u and v are both associates of some word r on $\{a, b\}$, and the result follows because $r \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} r$.

If v is not an associate of u , the production used to obtain v from u must be one of the $g_i z \rightarrow z \tilde{h}_i$. Since u is proper, we have $u = g_i q \tilde{p}$, where p, q are words on $\{a, b\}$. Then $v = q \tilde{p} \tilde{h}_i$. Thus, setting

$$r = pg_iq, \quad s = ph_iq,$$

the result follows because $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$. ■

Lemma 4. Let u be proper and let $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$. Then there are words r, s on $\{a, b\}$ which are associates of u, v , respectively, such that $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$.

Proof. By induction on the length of the derivation in v of v from u . The result is obvious if the derivation has length 1. Suppose the result is known for derivations of length m , and let

$$u = u_1 \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} u_2 \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} \cdots \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} u_m \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} u_{m+1} = v.$$

By the induction hypothesis, there are words r, z on $\{a, b\}$ which are associates of u, u_m , respectively, such that $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} z$. By Lemma 3, u_{m+1} is an associate of a word s on $\{a, b\}$ such that $z \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$. Thus, $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$. ■

Lemma 5. Let u, v be words on $\{a, b\}$. Then $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$ if and only if $u \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$.

Proof. By Lemma 2 we know that $u \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$ implies $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$. Conversely, if $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$, by Lemma 4, $r \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} s$, where r, s are words on $\{a, b\}$ which are associates of u, v , respectively. But since u, v are already words on $\{a, b\}$, we have $r = u$, $s = v$, so that $u \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$. ■

Since Π was chosen to have an unsolvable word problem, it is now clear that v has an unsolvable word problem. For, by Lemma 5, if we had an algorithm for deciding whether $u \xrightarrow[\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v$, we could use it to decide whether

$$u \xrightarrow{\underset{\text{v}}{\underset{\text{v}}{\Pi}}} v.$$

We have proved

Theorem 8.1. There is a normal process on a four-letter alphabet with an unsolvable word problem.

*9. A Non-R.E. Set

We conclude this chapter with an interesting 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 9.1. TOT is not r.e.

Proof. Suppose that TOT were r.e. Since $\text{TOT} \neq \emptyset$, by Theorem 7.1 there is a computable function $g(x)$ such that $\text{TOT} = \{g(0), g(1), g(2), \dots\}$. Let

$$h(x) = \Phi(x, g(x)) + 1.$$

Since each value $g(x)$ is the number of a program which 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 \mathcal{P} , and let $p = \#(\mathcal{P})$. Then $p \in \text{TOT}$, so that $p = g(i)$ for some i . Then

$$h(i) = \Phi(i, g(i)) + 1 = \Phi(i, p) + 1 = h(i) + 1,$$

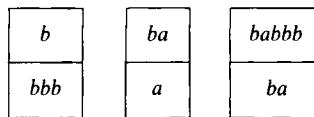
which is a contradiction. ■

Exercises

1. A semi-Thue system is defined to be a pair (u_0, Π) , where Π is a semi-Thue process and u_0 is a given word on the alphabet of Π . A word w is called a *theorem* of (u_0, Π) if $u_0 \xrightarrow{\Pi} w$. Show that there is a semi-Thue system for which no algorithm exists to determine whether a given string is a theorem of the system.

2. Let Π be a semi-Thue process containing only one production. Show that Π has a solvable word problem.

3. Find a solution to the Post correspondence problem defined by the dominoes



4. Find an algorithm for Post correspondence problems whose alphabet consists of just one symbol.

5. Show that there is a normal process with an unsolvable word problem whose alphabet contains only two letters.

6. (a) Let Γ have the variables S, B, C , the terminals a, b, c and the productions

$$\begin{array}{ll} S \rightarrow aSBC, & S \rightarrow aBC, \\ CB \rightarrow BC, & bB \rightarrow bb, \\ aB \rightarrow ab, & bC \rightarrow bc, \\ & cC \rightarrow cc. \end{array}$$

Prove that for each $n \neq 0$, $a^{[n]}b^{[n]}c^{[n]} \in L(\Gamma)$.

(b)* Prove that $L(\Gamma) = \{a^{[n]}b^{[n]}c^{[n]} | n \neq 0\}$.

Regular Languages

1. Finite Automata

Computability theory, discussed in Part 1, is the theory of computation obtained when limitations of space and time are deliberately ignored. In automata theory, which we study in this chapter, computation is studied in a context in which bounds on space and time are entirely relevant. The point of view of computability theory is exemplified in the behavior of a Turing machine (Chapter 6) in which a read–write head moves back and forth on an infinite tape, with no preset limit on the number of steps required to reach termination.¹ At the opposite pole, one can imagine a device which moves from left to right on a finite input tape, and it is just such devices, the so-called *finite automata*, that we will now study. Since a finite automaton will have only one opportunity to scan each square in its motion from left to right, nothing is to be gained by permitting the device to “print” new symbols on its tape.

Unlike modern computers, whose action is controlled in part by an internally stored list of instructions called a program, the computing devices we will consider in this chapter have no such programs and no internal memory for storing either programs or partial results. In addition, since, as we just indicated, a finite automaton is permitted only a single pass over the tape, there is no external memory available. Instead, there are internal *states* which control the automaton’s behavior and also function as memory in the sense of being able to retain some information about what has been read from the input tape up to a given point.

Thus, a finite automaton can be thought of as a very limited computing device which, after reading a string of symbols on the input tape, either

¹ The present chapter does not depend on familiarity with the material in Chapters 2–7.

Table 1.1

δ	a	b
q_1	q_2	q_4
q_2	q_2	q_3
q_3	q_4	q_3
q_4	q_4	q_4

accepts the input or rejects it, depending upon the state the machine is in when it has finished reading the tape.

The machine begins by reading the leftmost symbol on the tape, in a specified state called the *initial state* (the automaton is in this state whenever it is initially “turned on.”) If at a given time, the machine is in a state q_i reading a given symbol s_j on the input tape, the device moves one square to the right on the tape and enters a state q_k . The current state of the automaton plus the symbol on the tape being read completely determine the automaton’s next state.

Definition. A *finite automaton* \mathcal{M} on the alphabet² $A = \{s_1, \dots, s_n\}$ with states $Q = \{q_1, \dots, q_m\}$ is given by a function δ which maps each pair (q_i, s_j) , $1 \leq i \leq m$, $1 \leq j \leq n$, into a state q_k , together with a set $F \subseteq Q$. One of the states, usually q_1 , is singled out and called the *initial state*. The states belonging to the set F are called the *final* or *accepting* states. δ is called the *transition function*.

We can represent the function δ using a state versus symbol table. An example is given in Table 1.1, where the alphabet is $\{a, b\}$, $F = \{q_3\}$, and q_1 is the initial state. It is easy to check that for the tapes

a	a	b	b	b
b	a	b	a	
a	a	b	a	
a	b	b	b	

the automaton will terminate in states q_3 , q_4 , q_4 , and q_3 , respectively. We shall say that the automaton *accepts* the strings $aabb$ and $abbb$ (because

² For an introduction to alphabets and strings see Chapter 1, Section 3.

$q_3 \in F$), while it *rejects* the strings *baba* and *aaba* (because $q_4 \notin F$), i.e., that it *accepts* the first and fourth of the above tapes and *rejects* the second and third.

To proceed more formally, let \mathcal{M} be a finite automaton with transition function δ , initial state q_1 , and accepting states F . If q_i is any state of \mathcal{M} and $u \in A^*$, where A is the alphabet of \mathcal{M} , we shall write $\delta^*(q_i, u)$ for the state which \mathcal{M} will enter if it begins in state q_i at the left end of the string u and moves across u until the entire string has been processed. A formal definition by recursion is

$$\begin{aligned}\delta^*(q_i, 0) &= q_i, \\ \delta^*(q_i, us_j) &= \delta(\delta^*(q_i, u), s_j).\end{aligned}$$

Obviously, $\delta^*(q_i, s_j) = \delta(q_i, s_j)$. Then we say that \mathcal{M} *accepts* a word u provided that $\delta^*(q_1, u) \in F$. \mathcal{M} *rejects* u means that $\delta^*(q_1, u) \in Q - F$. Finally, the *language accepted by \mathcal{M}* , written $L(\mathcal{M})$, is the set of all $u \in A^*$ accepted by \mathcal{M} :

$$L(\mathcal{M}) = \{u \in A^* \mid \delta^*(q_1, u) \in F\}.$$

A language is called *regular* if there exists a finite automaton which accepts it.

It is important to realize that the notion of regular language does not depend on the particular alphabet. That is, if $L \subseteq A^*$ and $A \subseteq B$, then there is an automaton on the alphabet A which accepts L if and only if there is one on the alphabet B which accepts L . That is, an automaton with alphabet B can be contracted to one on the alphabet A by simply restricting the transition function δ to A ; clearly this will have no effect on which elements of A^* are accepted. Likewise, an automaton \mathcal{M} with alphabet A can be expanded to one with alphabet B by introducing a new “trap” state q and decreeing

$$\begin{aligned}\delta(q_i, b) &= q \text{ for all states } q_i \text{ of } \mathcal{M} \text{ and all } b \in B - A, \\ \delta(q, b) &= q \text{ for all } b \in B.\end{aligned}$$

Leaving the set of accepting states unchanged (so that q is not an accepting state), we see that the expanded automaton accepts the same language as \mathcal{M} .

Returning to the automaton given by Table 1.1 with $F = \{q_3\}$, it is easy to see that the language it accepts is

$$\{a^{[n]}b^{[m]} \mid n, m > 0\}. \quad (1.1)$$

Thus we have shown that (1.1) is a regular language.

We conclude this section by mentioning another way to represent the transition function δ . We can draw a graph in which each state is represented

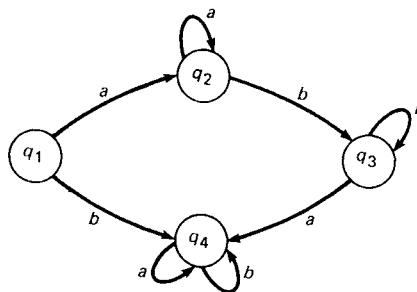


Fig. 1.1

by a *vertex*. Then, the fact that $\delta(q_i, s_j) = q_k$ is represented by drawing an arrow from vertex q_i to vertex q_k and labeling it s_j . The diagram thus obtained is called the *state transition diagram* for the given automaton. The state transition diagram for the transition function of Table 1.1 is shown in Fig. 1.1.

Exercises

1. In each of the following examples, an alphabet A and a language L are indicated with $L \subseteq A^*$. In each case show that L is regular by constructing a finite automaton \mathcal{M} which accepts L .

- (a) $A = \{1\}; L = \{1^{[6k]}1 \mid k \geq 0\}$.
- (b) $A = \{a, b\}; L$ consists of all words whose final four symbols form the string $bbab$.
- (c) $A = \{a, b\}; L$ consists of all words whose final five symbols include two a 's and three b 's
- (d) $A = \{0, 1\}; L$ consists of all strings which, when considered as binary numbers, have a value which is an integral multiple of 5.

$$(e) \quad A = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\};$$

\mathcal{M} is to be a binary addition checker in the sense that it accepts strings of binary triples

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} \cdots \begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix}$$

such that $c_1c_2 \cdots c_n$ is the sum of $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$ when each is treated as a binary number.

(f) $A = \{a, b, c\}$; A *palindrome* is a word such that $w = w^R$. That is, it reads the same backward and forward. L consists of all palindromes of length less than or equal to 6.

(g) $A = \{a, b\}$; L consists of all strings $s_1 s_2 \cdots s_n$ such that $s_{n-2} = b$. (Note that L contains no strings of length less than 3.)

2. Nondeterministic Finite Automata

Next we modify the definition of a finite automaton to permit transitions at each stage to either zero, one, or more than one states. Formally, we accomplish this by altering the definition of a finite automaton in the previous section by making the values of the transition function δ be *sets of states*, i.e., *sets of elements of Q* (rather than members of Q). The devices so obtained are called *nondeterministic finite automata (ndfa)*, and sometimes ordinary finite automata are then called *deterministic* finite automata (*dfa*). An ndfa on a given alphabet A with set of states Q is specified by giving such a transition function δ [which maps each pair (q_i, s_j) into a possibly empty subset of Q] and a fixed subset F of Q . For an ndfa, we define

$$\begin{aligned}\delta^*(q_i, 0) &= \{q_i\}, \\ \delta^*(q_i, us_j) &= \bigcup_{q \in \delta^*(q_i, u)} \delta(q, s_j).\end{aligned}$$

Thus, in calculating $\delta^*(q_i, u)$, one accumulates *all* states which the automaton can enter when it reaches the right end of u , beginning at the left end of u in state q_i . An ndfa \mathcal{M} with initial state q_1 accepts $u \in A^*$ if $\delta^*(q_1, u) \cap F \neq \emptyset$, i.e., if at least one of the states at which \mathcal{M} ultimately arrives belongs to F . Finally, $L(\mathcal{M})$, the *language accepted by \mathcal{M}* , is the set of all strings accepted by \mathcal{M} .

An example is given in Table 2.1 and Figure 2.1. Here $F = \{q_4\}$. It is not difficult to see that this ndfa accepts a string on the alphabet $\{a, b\}$ just in case at least one of the symbols has two successive occurrences in the string.

Table 2.1

δ	a	b
q_1	$\{q_1, q_2\}$	$\{q_1, q_3\}$
q_2	$\{q_4\}$	\emptyset
q_3	\emptyset	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$

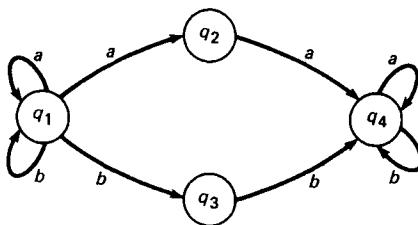


Fig. 2.1

In state q_1 , if the next character read is an a , then there are two possibilities. It might be that this a is the first of the desired pair of a 's. In that case we would want to remember that we had found one a and hence enter state q_2 to record that fact. On the other hand, it might be that the symbol following this a will be a b . Then this a is of no help in attaining the desired goal and hence we would remain in q_1 . Since we are not able to look ahead in the string, we cannot at this point determine which role the current a is playing and so the automaton “simultaneously” hypothesizes both possibilities. If the next character read is b , then since there is no transition from q_2 reading b , the choice has been resolved and the automaton will be in state q_1 . If instead, the character following the first a is another a , then since $q_2 \in \delta(q_1, a)$ and $q_4 \in \delta(q_2, a)$, and on any input the automaton once in state q_4 remains in q_4 , the input string will be accepted because q_4 is an accepting state. A similar analysis can be made if a b is read when the automaton is in state q_1 .

Strictly speaking, a dfa is *not* just a special kind of ndfa, although it is frequently thought of as such. This is because for a dfa, $\delta(q, s)$ is a state, whereas for an ndfa it is a set of states. But it is natural to identify the dfa \mathcal{M} with transition function δ , with the closely related ndfa $\tilde{\mathcal{M}}$ whose transition function $\tilde{\delta}$ is given by

$$\tilde{\delta}(q, s) = \{\delta(q, s)\},$$

and which has the same final states as \mathcal{M} . Obviously $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$.

The main theorem on nondeterministic finite automata is

Theorem 2.1. A language is accepted by an ndfa if and only if it is regular. Equivalently, a language is accepted by an ndfa if and only if it is accepted by a dfa.

Proof. As we have just seen, a language accepted by a dfa is also accepted by an ndfa. Conversely, let $L = L(\mathcal{M})$, where \mathcal{M} is an ndfa with transition function δ , set of states $Q = \{q_1, \dots, q_m\}$, and set of final states F . We will construct a dfa $\tilde{\mathcal{M}}$ such that $L(\tilde{\mathcal{M}}) = L(\mathcal{M}) = L$. The idea of the construction is that the individual states of $\tilde{\mathcal{M}}$ will be sets of states of \mathcal{M} .

Thus, we proceed to specify the dfa $\tilde{\mathcal{M}}$ on the same alphabet as \mathcal{M} . The states of $\tilde{\mathcal{M}}$ are just the 2^m sets of states (including \emptyset) of \mathcal{M} . We write these as $\tilde{Q} = \{Q_1, Q_2, \dots, Q_{2^m}\}$, where in particular $Q_1 = \{q_1\}$ is to be the initial state of $\tilde{\mathcal{M}}$. The set \mathcal{F} of final states of $\tilde{\mathcal{M}}$ is given by

$$\mathcal{F} = \{Q_i \mid Q_i \cap F \neq \emptyset\}.$$

The transition function $\tilde{\delta}$ of $\tilde{\mathcal{M}}$ is then defined by

$$\tilde{\delta}(Q_i, s) = \bigcup_{q \in Q_i} \delta(q, s).$$

Now, we have

Lemma 1. Let $R \subseteq \tilde{Q}$. Then

$$\tilde{\delta}\left(\bigcup_{Q_i \in R} Q_i, s\right) = \bigcup_{Q_i \in R} \tilde{\delta}(Q_i, s).$$

Proof. Let $\bigcup_{Q_i \in R} Q_i = Q$. Then by definition,

$$\begin{aligned} \tilde{\delta}(Q, s) &= \bigcup_{q \in Q} \delta(q, s) \\ &= \bigcup_{Q_i \in R} \bigcup_{q \in Q_i} \delta(q, s) \\ &= \bigcup_{Q_i \in R} \tilde{\delta}(Q_i, s). \end{aligned}$$
■

Lemma 2. For any string u ,

$$\tilde{\delta}^*(Q_i, u) = \bigcup_{q \in Q_i} \delta^*(q, u).$$

Proof. The proof is by induction on $|u|$. If $|u| = 0$, then $u = 0$ and

$$\tilde{\delta}^*(Q_i, 0) = Q_i = \bigcup_{q \in Q_i} \{q\} = \bigcup_{q \in Q_i} \delta^*(q, 0).$$

If $|u| = l + 1$ and the result is known for $|u| = l$, we write $u = vs$, where $|v| = l$, and observe that, using Lemma 1 and the induction hypothesis,

$$\begin{aligned} \tilde{\delta}^*(Q_i, u) &= \tilde{\delta}^*(Q_i, vs) \\ &= \tilde{\delta}(\tilde{\delta}^*(Q_i, v), s) \\ &= \tilde{\delta}\left(\bigcup_{q \in Q_i} \delta^*(q, v), s\right) \\ &= \bigcup_{q \in Q_i} \tilde{\delta}(\delta^*(q, v), s) \\ &= \bigcup_{q \in Q_i} \bigcup_{r \in \delta^*(q, v)} \delta(r, s) \\ &= \bigcup_{q \in Q_i} \delta^*(q, vs) \\ &= \bigcup_{q \in Q_i} \delta^*(q, u). \end{aligned}$$
■

Lemma 3. $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$.

Proof. $u \in L(\tilde{\mathcal{M}})$ if and only if $\tilde{\delta}^*(Q_1, u) \in \mathcal{F}$. But, by Lemma 2,

$$\tilde{\delta}^*(Q_1, u) = \tilde{\delta}^*(\{q_1\}, u) = \tilde{\delta}^*(q_1, u).$$

Hence,

$$\begin{aligned} u \in L(\tilde{\mathcal{M}}) &\quad \text{if and only if } \tilde{\delta}^*(q_1, u) \in \mathcal{F} \\ &\quad \text{if and only if } \tilde{\delta}^*(q_1, u) \cap F \neq \emptyset \\ &\quad \text{if and only if } u \in L(\mathcal{M}). \end{aligned}$$
■

Theorem 2.1 is an immediate consequence of Lemma 3. ■

Note that this proof is constructive. Not only have we shown that if a language is accepted by some ndfa, it is also accepted by some dfa, but we have also provided within the proof, an algorithm for carrying out the conversion. This is important because, although it is frequently easier to design an ndfa than a dfa to accept a particular language, actual machines which are built are deterministic.

Exercises

1. Let \mathcal{M} be a dfa with a single accepting state. Consider the ndfa \mathcal{M}' formed by reversing the roles of the initial and accepting states, and reversing the direction of the arrows of all transitions in the transition diagram. Describe $L(\mathcal{M}')$ in terms of $L(\mathcal{M})$.

2. Prove that, given any ndfa \mathcal{M}_1 , there exists an ndfa \mathcal{M}_2 with exactly one accepting state such that

$$L(\mathcal{M}_2) = L(\mathcal{M}_1) \text{ or } L(\mathcal{M}_1) - \{0\}.$$

3. Additional Examples

We first give two simple examples of finite automata and their associated regular languages.

For our first example we consider a unary even parity checker. That is, we want to design a finite automaton over the alphabet $\{1\}$ such that the machine terminates in an accepting state if and only if the input string contains an even number of 1's. Intuitively then, the machine must contain two states which "remember" whether an even or an odd number of 1's have been encountered so far. When the automaton begins, no 1's, and hence an even number of 1's, have been read; hence the initial state q_1 will represent the even parity state, and q_2 , the odd parity state. Furthermore, since we want

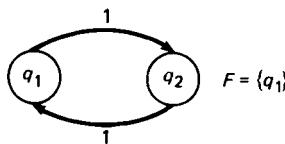


Fig. 3.1

to accept words containing an even number of 1's, q_1 will be an accepting state.

Thus the finite automaton to perform the required task is as shown in Fig. 3.1, and the language it accepts is

$$\{(11)^{[n]} \mid n \geq 0\}.$$

We next consider a slightly more complicated example. Suppose we wish to design a finite automaton which will function as a 25¢ candy vending machine. The alphabet consists of the three symbols n , d , and q (representing nickel, dime, and quarter, respectively—no pennies, please!). If more than 25¢ is deposited, no change is returned and no credit is given for the overage. Intuitively, the states keep track of the amount of money deposited so far. The automaton is exhibited in Fig. 3.2, with each state labeled to indicate its role. The state labeled 0 is the initial state. Note that the state labeled d is a “dead” state; i.e., once that state is entered it may never be left. Whenever sufficient money has been inserted so that the automaton has entered the 25¢ (accepting) state, any additional coins will send the machine into this dead state, which may be thought of as a coin return state. Presumably when in the accepting state, a button can be pressed to select your candy and the machine is reset to 0.

Unlike the previous example, the language accepted by this finite automaton is a finite set. It consists of the following combinations of nickels, dimes, and quarters: $\{nnnnn, nnnnd, nnnnq, nnnd, nnnq, nndn, nndd, nndq, nnq, ndnn, ndnd, ndnq, ndd, ndq, nq, dnnn, dnnd, dnqq, dnd, dnq, ddn, ddd, ddq, dq, q\}$.

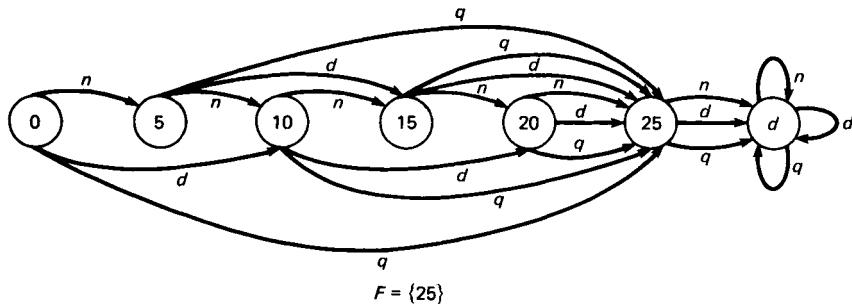


Fig. 3.2

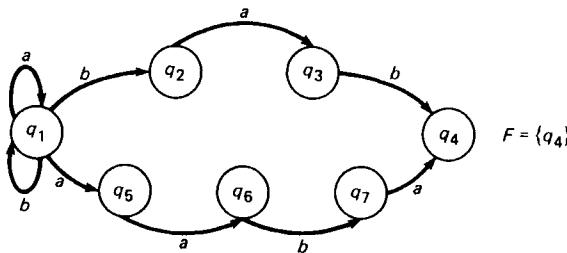


Fig. 3.3

Suppose we wish to design an automaton on the alphabet $\{a, b\}$ which accepts all and only strings which end in bab or $aaba$. A real-world analog of this problem might arise in a demographic study in which people of certain ethnic groups are to be identified by checking to see if their family name ends in certain strings of letters.

It is easy to design the desired ndfa: see Fig. 3.3.

As our final example, we discuss a slightly more complicated version of the first example considered in Section 1:

$$L = \{a^{[n_1]}b^{[m_1]}\dots a^{[n_k]}b^{[m_k]} \mid n_1, m_1, \dots, n_k, m_k > 0\}.$$

An ndfa \mathcal{M} such that $L(\mathcal{M}) = L$ is shown in Fig. 3.4.

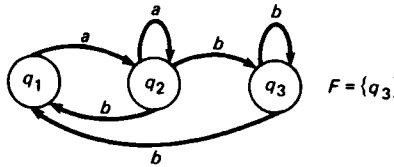


Fig. 3.4

These two examples of ndfas illustrate an important characteristic of such machines: not only is it permissible to have many alternative transitions for a given state–symbol pair, but frequently there are no transitions for a given pair. In a sense, this means that whereas for a dfa one has to describe what happens for *any* string whether or not that string is a word in the language, for an ndfa one need only describe the behavior of the automaton for words in the language.

4. Closure Properties

We will be able to prove that the class of regular languages is closed under a large number of operations. It will be helpful that, by the equivalence

theorems of the previous two sections, we can use deterministic or non-deterministic finite automata to suit our convenience.

Definition. A dfa is called *nonrestarting* if there is no pair q, s for which

$$\delta(q, s) = q_1,$$

where q_1 is the initial state.

Theorem 4.1. There is an algorithm which will transform a given dfa \mathcal{M} into a nonrestarting dfa $\tilde{\mathcal{M}}$ such that $L(\tilde{\mathcal{M}}) = L(\mathcal{M})$.

Proof. Let $Q = \{q_1, q_2, \dots, q_n\}$ be the set of states of \mathcal{M} , q_1 the initial state, F the set of accepting states, and δ the transition function. We construct $\tilde{\mathcal{M}}$ with the set of states $\tilde{Q} = Q \cup \{q_{n+1}\}$, initial state q_1 , and transition function $\tilde{\delta}$ defined by

$$\begin{aligned}\tilde{\delta}(q, s) &= \begin{cases} \delta(q, s) & \text{if } q \in Q \text{ and } \delta(q, s) \neq q_1 \\ q_{n+1} & \text{if } q \in Q \text{ and } \delta(q, s) = q_1, \end{cases} \\ \tilde{\delta}(q_{n+1}, s) &= \tilde{\delta}(q_1, s).\end{aligned}$$

Thus, there is no transition into state q_1 for $\tilde{\mathcal{M}}$. The set of accepting states \tilde{F} of $\tilde{\mathcal{M}}$ is defined by

$$\tilde{F} = \begin{cases} F & \text{if } q_1 \notin F \\ F \cup \{q_{n+1}\} & \text{if } q_1 \in F. \end{cases}$$

To see that $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$ as required, one need only observe that $\tilde{\mathcal{M}}$ follows the same transitions as \mathcal{M} except that whenever \mathcal{M} reenters q_1 , $\tilde{\mathcal{M}}$ enters q_{n+1} . ■

Theorem 4.2. If L and \tilde{L} are regular languages, then so is $L \cup \tilde{L}$.

Proof. Without loss of generality, by Theorem 4.1, let $\mathcal{M}, \tilde{\mathcal{M}}$ be nonrestarting dfas which accept L and \tilde{L} respectively, with Q, q_1, F, δ and $\tilde{Q}, \tilde{q}_1, \tilde{F}, \tilde{\delta}$ the set of states, initial state, set of accepting states, and transition function of \mathcal{M} and $\tilde{\mathcal{M}}$, respectively. We also assume that \mathcal{M} and $\tilde{\mathcal{M}}$ have no states in common, i.e., $Q \cap \tilde{Q} = \emptyset$. Furthermore, by the discussion in Section 1, we can assume that the alphabets of L and \tilde{L} are the same, say, A . We define the ndfa \mathcal{M} with states \check{Q} , initial state \check{q}_1 , set of accepting states \check{F} , and transition function $\check{\delta}$ as follows:

$$\check{Q} = Q \cup \tilde{Q} \cup \{\check{q}_1\} - \{q_1, \tilde{q}_1\}.$$

(That is, $\tilde{\mathcal{M}}$ contains a new initial state \tilde{q}_1 and all states of \mathcal{M} and $\tilde{\mathcal{M}}$ except their initial states.)

$$\tilde{F} = \begin{cases} F \cup \tilde{F} \cup \{\tilde{q}_1\} - \{q_1, \tilde{q}_1\} & \text{if } q_1 \in F \text{ or } \tilde{q}_1 \in \tilde{F} \\ F \cup \tilde{F} & \text{otherwise.} \end{cases}$$

The transition function of $\tilde{\mathcal{M}}$ is defined as follows for $s \in A$:

$$\begin{aligned}\tilde{\delta}(q, s) &= \begin{cases} \{\delta(q, s)\} & \text{if } q \in Q - \{q_1\} \\ \{\tilde{\delta}(q, s)\} & \text{if } q \in \tilde{Q} - \{\tilde{q}_1\} \end{cases} \\ \tilde{\delta}(\tilde{q}_1, s) &= \{\delta(q_1, s)\} \cup \{\tilde{\delta}(\tilde{q}_1, s)\}.\end{aligned}$$

Thus, since $Q \cap \tilde{Q} = \emptyset$ and \mathcal{M} and $\tilde{\mathcal{M}}$ are nonrestarting, once a first transition has been selected, the automaton $\tilde{\mathcal{M}}$ is locked into one of the two automata \mathcal{M} and $\tilde{\mathcal{M}}$. Hence $L(\tilde{\mathcal{M}}) = L \cup \tilde{L}$. ■

Theorem 4.3. Let $L \subseteq A^*$ be a regular language. Then $A^* - L$ is regular.

Proof. Let \mathcal{M} be a dfa which accepts L . Let $\tilde{\mathcal{M}}$ have alphabet A , set of states Q , and set of accepting states F . Let $\tilde{\mathcal{M}}$ be exactly like \mathcal{M} except that it accepts precisely when \mathcal{M} rejects. That is, the set of accepting states of $\tilde{\mathcal{M}}$ is $Q - F$. Then $\tilde{\mathcal{M}}$ clearly accepts $A^* - L$. ■

Theorem 4.4. If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$.

Proof. Let $L_1, L_2 \subseteq A^*$. Then we have the De Morgan identity:

$$L_1 \cap L_2 = A^* - ((A^* - L_1) \cup (A^* - L_2)).$$

Theorems 4.2 and 4.3 then give the result. ■

Theorem 4.5. \emptyset and $\{0\}$ are regular languages.

Proof. \emptyset is clearly the language accepted by any automaton whose set of accepting states is empty. Next, the automaton with states q_1, q_2 , alphabet $\{a\}$, accepting states $F = \{q_1\}$, and transition function $\delta(q_1, a) = \delta(q_2, a) = q_2$ clearly accepts $\{0\}$, as does any nonrestarting dfa on any alphabet provided $F = \{q_1\}$. ■

Theorem 4.6. Let $u \in A^*$. Then $\{u\}$ is a regular language.

Proof. For $u = 0$, we already know this from Theorem 4.5. Otherwise let $u = a_1 a_2 \cdots a_l a_{l+1}$, where $a_1, a_2, \dots, a_l, a_{l+1} \in A$. Let \mathcal{M} be the ndfa with states q_1, q_2, \dots, q_{l+2} , initial state q_1 , accepting state q_{l+2} , and transition function δ given by

$$\begin{aligned}\delta(q_i, a_i) &= \{q_{i+1}\}, & i &= 1, \dots, l+1, \\ \delta(q_i, a) &= \emptyset & \text{for } a \in A - \{a_i\}.\end{aligned}$$

Then $L(\mathcal{M}) = \{u\}$. ■

Corollary 4.7. Every finite subset of A^* is regular.

Proof. We have already seen that \emptyset is regular. If $L = \{u_1, \dots, u_n\}$, where $u_1, \dots, u_n \in A^*$, we note that

$$L = \{u_1\} \cup \{u_2\} \cup \dots \cup \{u_n\},$$

and apply Theorems 4.2 and 4.6. ■

5. Kleene's Theorem

In this section we will see how the class of regular languages can be characterized as the class of all languages obtained from finite languages using a few operations.

Definition. Let $L_1, L_2 \subseteq A^*$. Then, we write

$$L_1 \cdot L_2 = L_1 L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}.$$

Definition. Let $L \subseteq A^*$. Then we write

$$L^* = \{u_1 u_2 \cdots u_n \mid n \geq 0, u_1, \dots, u_n \in L\}.$$

With respect to this last definition, note that

1. $0 \in L^*$ automatically because $n = 0$ is allowed;
2. for A^* the present notation is consistent with what we have been using.

Theorem 5.1. If L, \tilde{L} are regular languages, then $L \cdot \tilde{L}$ is a regular language.

Proof. Assume \mathcal{M} and $\tilde{\mathcal{M}}$ are dfa's which accept L and \tilde{L} , respectively, with Q, q_1, F, δ and $\tilde{Q}, \tilde{q}_1, \tilde{F}, \tilde{\delta}$ the set of states, initial state, set of accepting states, and transition function, respectively. Assume that \mathcal{M} and $\tilde{\mathcal{M}}$ have no states in common, i.e., $Q \cap \tilde{Q} = \emptyset$. By our discussion in Section 1, we can assume without loss of generality that the alphabets of L and \tilde{L} are the same. Consider the ndfa $\hat{\mathcal{M}}$ formed by “gluing together” \mathcal{M} and $\tilde{\mathcal{M}}$ in the following way. The set \hat{Q} of states of $\hat{\mathcal{M}}$ is $Q \cup \tilde{Q}$, the initial state is q_1 , and the set of accepting states is \tilde{F} . We will define the transition function $\hat{\delta}$ of $\hat{\mathcal{M}}$ in such a way that the transitions of $\hat{\mathcal{M}}$ will contain all transitions of \mathcal{M} and $\tilde{\mathcal{M}}$. In addition $\hat{\delta}(q, s)$ will contain $\tilde{\delta}(\tilde{q}_1, s)$ for every $q \in F$. Thus, any time a symbol of the input string causes \mathcal{M} to enter an accepting state, $\hat{\mathcal{M}}$ can either continue by treating the

next symbol of the input as being from the word of L or as the first symbol of the word of \tilde{L} . Formally we define $\dot{\delta}$ as follows:

$$\dot{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{for } q \in Q - F \\ \{\delta(q, s)\} \cup \{\tilde{\delta}(\tilde{q}_1, s)\} & \text{for } q \in F \\ \{\tilde{\delta}(q, s)\} & \text{for } q \in \tilde{Q}. \end{cases}$$

Thus, $\dot{\mathcal{M}}$ begins by behaving exactly like \mathcal{M} . Just when \mathcal{M} would accept a word by entering an accepting state, however, $\dot{\mathcal{M}}$ will instead enter the initial state of $\tilde{\mathcal{M}}$ and proceed as $\tilde{\mathcal{M}}$ would. Thus

$$L(\dot{\mathcal{M}}) = L \cdot \tilde{L}. \quad \blacksquare$$

Theorem 5.2. If L is a regular language, then so is L^* .

Proof. Let \mathcal{M} be a nonrestarting dfa which accepts L with alphabet A , set of states Q , initial state q_1 , accepting states F , and transition function δ . We construct the ndfa $\dot{\mathcal{M}}$ with the same states and initial state as \mathcal{M} , and accepting state q_1 . The transition function $\dot{\delta}$ is defined as follows:

$$\dot{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{if } \delta(q, s) \notin F \\ \{\delta(q, s)\} \cup \{q_1\} & \text{if } \delta(q, s) \in F. \end{cases}$$

That is, whenever \mathcal{M} would enter an accepting state, $\dot{\mathcal{M}}$ will enter either the corresponding accepting state or the initial state. Clearly $L^* = L(\dot{\mathcal{M}})$, so that L^* is a regular language. \blacksquare

Theorem 5.3 (Kleene's Theorem). Every regular language can be obtained from finite languages by applying the three operations $\cup, \cdot, *$, a finite number of times.

The characterization of regular languages which Kleene's theorem gives resembles the definition of the primitive recursive functions and the characterization of the partially computable functions of Theorem 7.7 in Chapter 7. In each case one begins with some initial objects and applies certain operations a finite number of times.

Proof. Let L be a regular language, $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with states q_1, \dots, q_n . As usual, q_1 is the initial state, F is the set of accepting states, δ is the transition function, and $A = \{s_1, \dots, s_K\}$ is the alphabet. We define the sets $R_{i,j}^k$, $i, j > 0$, $k \geq 0$, as follows:

$$R_{i,j}^k = \{x \in A^* \mid \delta^*(q_i, x) = q_j \text{ and } \mathcal{M} \text{ passes through no state } q_l \text{ with } l > k \text{ as it moves across } x\}.$$

More formally, $R_{i,j}^k$ is the set of words $x = s_{i_1}s_{i_2}\cdots s_{i_r}s_{i_{r+1}}$ such that we can write

$$\begin{aligned}\delta(q_i, s_{i_1}) &= q_{j_1}, \\ \delta(q_{j_1}, s_{i_2}) &= q_{j_2}, \\ &\vdots \\ \delta(q_{j_{r-1}}, s_{i_r}) &= q_{j_r}, \\ \delta(q_{j_r}, s_{i_{r+1}}) &= q_j,\end{aligned}$$

where $j_1, j_2, \dots, j_r \leq k$. Now, we observe that

$$R_{i,j}^0 = \{a \in A \mid \delta(q_i, a) = q_j\},$$

since for a word of length 1, \mathcal{M} passes directly from state q_i into state q_j while in processing any word of length > 1 , \mathcal{M} will pass through some intermediate state q_l , $l \geq 1$. Thus $R_{i,j}^0$ is a finite set. Furthermore, we have

$$R_{i,j}^{k+1} = R_{i,j}^k \cup [R_{i,k+1}^k \cdot (R_{k+1,k+1}^k)^* \cdot R_{k+1,j}^k]. \quad (5.1)$$

This rather imposing formula really states something quite simple: The set $R_{i,j}^{k+1}$ contains all the elements of $R_{i,j}^k$ and in addition contains strings x , such that \mathcal{M} in scanning x passes through the state q_{k+1} (but through none with larger subscript) some finite number of times. Such a string can be decomposed into a left end, which \mathcal{M} enters in state q_i and leaves in state q_{k+1} (passing only through states with subscripts less than $k+1$ in the process), followed by some finite number of pieces each of which \mathcal{M} enters and leaves in state q_{k+1} (passing only through q_l with $l \leq k$), and a right end which \mathcal{M} enters in state q_{k+1} and leaves in state q_j (again passing only through states with subscript $\leq k$ in between). Now we have

Lemma. Each $R_{i,j}^k$ can be obtained from finite languages by a finite number of applications of the operations $\cup, \cdot, *$.

Proof. We prove by induction on k that for all i, j , the set $R_{i,j}^k$ has the desired property. For $k = 0$ this is obvious, since $R_{i,j}^0$ is finite.

Assuming the result known for k , (5.1) yields the result for $k + 1$. ■

Now to conclude the proof of Kleene's theorem, we note that

$$L(\mathcal{M}) = \bigcup_{q_j \in F} R_{1,j}^n;$$

thus, the result follows at once from the lemma. ■

Kleene's theorem makes it possible to give names to regular languages in a particularly simple way. Let us begin with an alphabet $A = \{s_1, s_2, \dots, s_k\}$. Then we define the corresponding alphabet:

$$\tilde{A} = \{s_1, s_2, \dots, s_k, \mathbf{0}, \emptyset, \cup, \cdot, *, (,)\}.$$

The class of *regular expressions* on the alphabet A is then defined to be the subset of \tilde{A}^* determined by (1)–(5) below:

1. $\emptyset, 0, s_1, \dots, s_k$ are regular expressions.
2. If α and β are regular expressions, then so is $(\alpha \cup \beta)$.
3. If α and β are regular expressions, then so is $(\alpha \cdot \beta)$.
4. If α is a regular expression, then so is α^* .
5. No expression is regular unless it can be generated using a finite number of applications of (1)–(4).

Here are a few examples of regular expressions on the alphabet $A = \{a, b, c\}$:

$$(a \cdot (b^* \cup c^*))$$

$$(0 \cup (a \cdot b)^*)$$

$$(c^* \cdot b^*).$$

For each regular expression γ , we define a corresponding regular language $\langle \gamma \rangle$ by recursion according to the following “semantic” rules:

$$\langle s_i \rangle = \{s_i\},$$

$$\langle 0 \rangle = \{0\},$$

$$\langle \emptyset \rangle = \emptyset,$$

$$\langle (\alpha \cup \beta) \rangle = \langle \alpha \rangle \cup \langle \beta \rangle,$$

$$\langle (\alpha \cdot \beta) \rangle = \langle \alpha \rangle \cdot \langle \beta \rangle,$$

$$\langle \alpha^* \rangle = \langle \alpha \rangle^*.$$

When $\langle \gamma \rangle = L$, we say that the regular expression γ represents L . Thus,

$$\langle (a \cdot (b^* \cup c^*)) \rangle = \{ab^{[n]} \mid n \geq 0\} \cup \{ac^{[m]} \mid m \geq 0\},$$

$$\langle (0 \cup (a \cdot b)^*) \rangle = \langle (a \cdot b)^* \rangle = \{(ab)^{[n]} \mid n \geq 0\},$$

$$\langle (c^* \cdot b^*) \rangle = \{c^{[m]}b^{[n]} \mid m, n \geq 0\}.$$

We have

Theorem 5.4. For every finite subset L of A^* , there is a regular expression γ on A such that $\langle \gamma \rangle = L$.

Proof. If $L = \emptyset$, then $L = \langle \emptyset \rangle$. If $L = \{0\}$, then $L = \langle 0 \rangle$. If $L = \{x\}$, where $x = s_{i_1} s_{i_2} \cdots s_{i_l}$, then

$$L = \langle (s_{i_1} \cdot (s_{i_2} \cdot (s_{i_3} \cdots s_{i_l}) \cdots)) \rangle.$$

This gives the result for languages L consisting of 0 or 1 element. Assuming the result known for languages of k elements, let L have $k + 1$ elements. Then we can write

$$L = L_1 \cup \{x\},$$

where $x \in A^*$ and L_1 contains k elements. By the induction hypothesis, there is a regular expression α such that $\langle \alpha \rangle = L_1$. By the one-element case already considered, there is a regular expression β such that $\langle \beta \rangle = \{x\}$. Then we have

$$\langle (\alpha \cup \beta) \rangle = \langle \alpha \rangle \cup \langle \beta \rangle = L_1 \cup \{x\} = L. \quad \blacksquare$$

Theorem 5.5 (Kleene's Theorem—Second Version). For every regular language $L \subseteq A^*$, there is a regular expression γ on A such that $\langle \gamma \rangle = L$.

Proof. By Theorem 5.4, this is true for finite languages. Let L be a regular language. By Kleene's theorem, L can be obtained from certain finite language by a finite number of applications of the operations $\cup, \cdot, *$. By beginning with regular expressions representing these finite languages, we can build up a regular expression representing L by simply indicating each use of the operations $\cup, \cdot, *$ by writing $\cup, \cdot, *$, respectively, and punctuating with (and). \blacksquare

Exercises

1. For regular expressions α, β , let us write $\alpha \equiv \beta$ to mean that $\langle \alpha \rangle = \langle \beta \rangle$. For α, β, γ given regular expressions, prove the following identities:

- (a) $(\alpha \cup \alpha) \equiv \alpha$
- (b) $((\alpha \cdot \beta) \cup (\alpha \cdot \gamma)) \equiv (\alpha \cdot (\beta \cup \gamma))$
- (c) $((\beta \cdot \alpha) \cup (\gamma \cdot \alpha)) \equiv ((\beta \cup \gamma) \cdot \alpha)$
- (d) $(\alpha^* \cdot \alpha^*) \equiv \alpha^*$
- (e) $(\alpha \cdot \alpha^*) \equiv (\alpha^* \cdot \alpha)$
- (f) $\alpha^{**} \equiv \alpha^*$
- (g) $(\mathbf{0} \cup (\alpha \cdot \alpha^*)) \equiv \alpha^*$
- (h) $((\alpha \cdot \beta)^* \cdot \alpha) \equiv (\alpha \cdot (\beta \cdot \alpha)^*)$
- (i) $(\alpha \cup \beta)^* \equiv (\alpha^* \cdot \beta^*)^* \equiv (\alpha^* \cup \beta^*)^*$.

2. Using the identities of Exercise 1 prove that

$$((abb \cup ba)^*(b \cup aa)) \equiv (abb)^*((\mathbf{0} \cup (b(ab)^*a))b \cup (ba)^*(aa)).$$

(Note that parentheses and the symbol “.” have been omitted to facilitate reading.)

3. Let α, β be given regular expressions such that $0 \notin \langle \alpha \rangle$. Consider the equation in the “unknown” regular expression ξ :

$$\xi \equiv (\beta \cup (\xi \cdot \alpha)).$$

Prove that this equation has the solution

$$\xi \equiv (\beta \cdot \alpha^*)$$

and that the solution is unique in the sense that if ξ' also satisfies the equation, then $\xi \equiv \xi'$.

4. Let $L = \{x \in \{a, b\}^* \mid x \neq 0 \text{ and } bb \text{ is not a substring of } x\}$.

- (a) Show that L is regular by constructing a dfa \mathcal{M} such that $L = L(\mathcal{M})$.
- (b) Find a regular expression γ such that $L = \langle \gamma \rangle$.

5. Let $L = \langle ((a \cdot a) \cup (a \cdot a \cdot a))^* \rangle$. Find a dfa \mathcal{M} which accepts L .

6. The Pumping Lemma and Its Applications

We will make use of the following basic combinatorial fact:

Pigeon-Hole Principle. If $(n + 1)$ objects are distributed among n sets, then at least one of the sets must contain at least 2 objects.

We will use this pigeon-hole principle to prove the following result:

Theorem 6.1 (Pumping Lemma). Let $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with n states. Let $x \in L$, where $|x| \geq n$. Then we can write $x = uvw$, where $v \neq 0$ and $uv^{[i]}w \in L$ for all $i = 0, 1, 2, 3, \dots$.

Proof. Since x consists of at least n symbols, \mathcal{M} must go through at least n state transitions as it scans x . Including the initial state, this requires at least $n + 1$ (not necessarily distinct) states. But since there are only n states in all, we conclude (here is the pigeon-hole principle!) that \mathcal{M} must be in at least one state more than once. Let q be a state in which \mathcal{M} finds itself at least twice. Then we can write $x = uvw$, where

$$\delta^*(q_1, u) = q,$$

$$\delta^*(q, v) = q,$$

$$\delta^*(q, w) \in F.$$

That is, \mathcal{M} arrives in state q for the first time after scanning the last (right-hand) symbol of u and then again after scanning the last symbol of v . Since this “loop” can be repeated any number of times, it is clear that

$$\delta^*(q_1, uv^{[i]}w) = \delta^*(q_1, uvw) \in F.$$

Hence $uv^{[i]}w \in L$. ■

Theorem 6.2. Let \mathcal{M} be a dfa with n states. Then, if $L(\mathcal{M}) \neq \emptyset$, there is a string $x \in L(\mathcal{M})$ such that $|x| < n$.

Proof. Let x be a string in $L(\mathcal{M})$ of the shortest possible length. Suppose $|x| \geq n$. By the pumping lemma, $x = uvw$, where $v \neq 0$ and $uw \in L(\mathcal{M})$. Since $|uw| < |x|$, this is a contradiction. Thus, $|x| < n$. ■

This theorem furnishes an algorithm for testing a given dfa \mathcal{M} to see whether the language it accepts is empty. We need only “run” \mathcal{M} on all strings of length less than the number of states of \mathcal{M} . If none are accepted, we will be able to conclude that $L(\mathcal{M}) = \emptyset$.

Next we turn to infinite regular languages. If $L = L(\mathcal{M})$ is infinite, then L must surely contain words having length greater than the number of states of \mathcal{M} . Hence from the pumping lemma, we can conclude

Theorem 6.3. If L is an infinite regular language, then there are words u, v, w , such that $v \neq 0$ and $uv^{[i]}w \in L$ for $i = 0, 1, 2, 3, \dots$.

This theorem is useful in showing that certain languages are *not* regular. However, for infinite regular languages we can say even more.

Theorem 6.4 Let \mathcal{M} be a dfa with n states. Then $L(\mathcal{M})$ is infinite if and only if $L(\mathcal{M})$ contains a string x such that $n \leq |x| < 2n$.

Proof. First let $x \in L(\mathcal{M})$ with $n \leq |x| < 2n$. By the pumping lemma, we can write $x = uvw$, where $v \neq 0$ and $uv^{[i]}w \in L(\mathcal{M})$ for all i . But then $L(\mathcal{M})$ is infinite.

Conversely, let $L(\mathcal{M})$ be infinite. Then $L(\mathcal{M})$ must contain strings of length $\geq 2n$. Let $x \in L(\mathcal{M})$, where x has the shortest possible length $\geq 2n$. We write $x = x_1x_2$, where $|x_1| = n$. Thus $|x_2| \geq n$. Then using the pigeon-hole principle as in the proof of the pumping lemma, we can write $x_1 = uvw$, where

$$\delta^*(q_1, u) = q,$$

$$\delta^*(q, v) = q \quad \text{with} \quad 1 \leq |v| \leq n,$$

$$\delta^*(q, wx_2) \in F.$$

Thus $uwx_2 \in L(\mathcal{M})$. But

$$|uwx_2| \geq |x_2| \geq n,$$

and $|uwx_2| < |x|$, and since x was a shortest word of $L(\mathcal{M})$ with length at least $2n$, we have

$$n \leq |uwx_2| < 2n. \quad \blacksquare$$

This theorem furnishes an algorithm for testing a given dfa \mathcal{M} to determine whether $L(\mathcal{M})$ is finite. We need only run \mathcal{M} on all strings x such that

$n \leq |x| < 2n$, where \mathcal{M} has n states. $L(\mathcal{M})$ is infinite just in case \mathcal{M} accepts at least one of these strings.

For another example of an algorithm, let $\mathcal{M}_1, \mathcal{M}_2$ be dfa's on the alphabet A and let us seek to determine whether $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$. Using the methods of proof of Theorems 4.2–4.4, we can obtain a dfa \mathcal{M} such that

$$L(\mathcal{M}) = L(\mathcal{M}_1) \cap [A^* - L(\mathcal{M}_2)].$$

Then $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$ if and only if $L(\mathcal{M}) = \emptyset$. Since Theorem 6.2 enables us to test algorithmically whether $L(\mathcal{M}) = \emptyset$, we have an algorithm by means of which we can determine whether $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$.

7. The Myhill–Nerode Theorem

We conclude this chapter by giving another characterization of the regular languages on an alphabet A . We begin with a pair of definitions:

Definition. Let $L \subseteq A^*$, where A is an alphabet. For strings $x, y \in A^*$, we write $x \equiv_L y$ to mean that for every $w \in A^*$ we have $xw \in L$ if and only if $yw \in L$.

It is obvious that

$$x \equiv_L x.$$

If $x \equiv_L y$, then $y \equiv_L x$. If $x \equiv_L y$ and $y \equiv_L z$, then $x \equiv_L z$. (Relations having these three properties are known as *equivalence relations*.)

It is also obvious that

$$\text{If } x \equiv_L y, \text{ then for all } w \in A^*, xw \equiv_L yw.$$

Definition. Let $L \subseteq A^*$, where A is an alphabet. Let $S \subseteq A^*$. Then S is called a *spanning set for L* if

1. S is finite, and
2. for every $x \in A^*$, there is a $y \in S$ such that $x \equiv_L y$.

Then we can prove

Theorem 7.1 (Myhill–Nerode). A language is regular if and only if it has a spanning set.

Proof. First let L be regular. Then $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with set of states Q , initial state q_1 , and transition function δ . Let us call a state $q \in Q$ *reachable* if there exists $y \in A^*$ such that

$$\delta^*(q_1, y) = q. \tag{7.1}$$

For each reachable state q , we select one particular string y that satisfies (7.1) and we write it as y_q . Thus,

$$\delta^*(q_1, y_q) = q$$

for every reachable state q . We set

$$S = \{y_q \mid q \text{ is reachable}\}.$$

S is clearly finite. To show that S is a spanning set for L , we let $x \in A^*$ and show how to find $y \in S$ such that $x \equiv_L y$. In fact, let $\delta^*(q_1, x) = q$, and set $y = y_q$. Thus, $y \in S$ and $\delta^*(q_1, y) = q$. Now for every $w \in A^*$,

$$\delta^*(q_1, xw) = \delta^*(q, w) = \delta^*(q_1, yw).$$

Hence, $\delta^*(q_1, xw) \in F$ if and only if $\delta^*(q_1, yw) \in F$; i.e., $xw \in L$ if and only if $yw \in L$. Thus, $x \equiv_L y$.

Conversely, let $L \subseteq A^*$ and let $S \subseteq A^*$ be a spanning set for L . We show how to construct a dfa \mathcal{M} such that $L(\mathcal{M}) = L$. We define the set of states of \mathcal{M} to be $Q = \{q_x \mid x \in S\}$, where we have associated a state q_x with each element $x \in S$. Since S is a spanning set for L , there is an $x_0 \in S$ such that $0 \equiv_L x_0$; we take q_{x_0} to be the initial state of \mathcal{M} . We let the final states of \mathcal{M} be

$$F = \{q_y \mid y \in L\}.$$

Finally, for $a \in A$, we set $\delta(q_x, a) = q_y$, where $y \in S$ and $xa \equiv_L y$. Then we claim that for all $w \in A^*$,

$$\delta^*(q_x, w) = q_y, \quad \text{where } xw \equiv_L y.$$

We prove this claim by induction on $|w|$. For $|w| = 0$, we have $w = 0$. Moreover, $\delta^*(q_x, 0) = q_x$ and $x0 = x \equiv_L x$. Suppose our claim is known for all words w such that $|w| = k$, and consider $w \in A^*$ with $|w| = k + 1$. Then $w = ua$, where $|u| = k$ and $a \in A$. We have

$$\delta^*(q_x, w) = \delta(\delta^*(q_x, u), a) = \delta(q_y, a) = q_z,$$

where, using the induction hypothesis, $xu \equiv_L y$ and, by definition of δ , $ya \equiv_L z$. Then $xw = xua \equiv_L ya \equiv_L z$, which proves the claim. Now, we have

$$L(\mathcal{M}) = \{w \in A^* \mid \delta^*(q_{x_0}, w) \in F\}.$$

Let $\delta^*(q_{x_0}, w) = q_y$. Then by the way x_0 was defined and our claim,

$$w \equiv_L x_0 w \equiv_L y.$$

Thus, $w \in L$ if and only if $y \in L$, which in turn is true if and only if $q_y \in F$, i.e., if and only if $w \in L(\mathcal{M})$. Hence $L = L(\mathcal{M})$. ■

The Myhill–Nerode theorem and the pumping lemma both furnish techniques for showing that given languages are *not* regular.

Exercises

1. Prove that there is no dfa which accepts exactly the set of all words which are palindromes over a given alphabet containing at least two symbols. (For a definition of palindrome see Exercise 1f, Section 1, this chapter.)

2. u is called an *initial segment* of a word w if there is a word v such that $w = uv$. Let L be a regular language. Prove that the language consisting of all initial segments of words of L is a regular language.

3. Let L be a regular language and L' the language consisting of all words w such that both w and $w \cdot w$ are words in L . Prove that L' is regular.

4. Prove the following statement if it is true, or give a counterexample: Every language which is a subset of a regular language is regular.

5. Prove that each of the following is *not* a regular language:

(a) The language on the alphabet $\{a, b\}$ consisting of all strings in which the number of occurrences of b is greater than the number of occurrences of a .

(b) The language L over the alphabet $\{\cdot, 0, 1, \dots, 9\}$, consisting of all strings which are initial segments of the infinite decimal expansion of π . [$L = \{3, 3., 3.1, 3.14, 3.141, 3.1415, \dots\}$.]

(c) The language L over the alphabet $\{a, b\}$ consisting of all strings which are initial segments of the infinite string

$$bababaaaabaaaab\dots$$

6. Let $L = \{a^{[n]}b^{[n]} \mid n > 0\}$. Show that L is not regular.

7. Let $L = \{a^{[n]}b^{[2n]} \mid n > 0\}$. Show that L is not regular.

8. Let $L = \{a^{[n]}b^{[m]} \mid 0 < n \leq m\}$. Show that L is not regular.

9. Let \mathcal{M} be a finite automaton with alphabet A , set of states $Q = \{q_1, \dots, q_n\}$, initial state q_1 , and transition function δ . Let a_1, a_2, a_3, \dots be an *infinite sequence* of symbols of A . We can think of these symbols as being “fed” to \mathcal{M} in the given order producing a sequence of states r_1, r_2, r_3, \dots , where r_1 is just the initial state q_1 and $r_{i+1} = \delta(r_i, a_i)$, $i = 1, 2, 3, \dots$. Suppose there are integers p, k such that

$$a_{i+p} = a_i \quad \text{for all } i \geq k.$$

Prove that there are integers l, s such that $s \leq np$ and

$$r_{i+s} = r_i \quad \text{for all } i \geq l.$$

(Hint: Use the pigeon-hole principle.)

Context-Free Languages

1. Context-Free Grammars and Their Derivation Trees

Let \mathcal{V}, T be a pair of disjoint alphabets. A *context-free production* on \mathcal{V}, T is an expression

$$X \rightarrow h$$

where $X \in \mathcal{V}$ and $h \in (\mathcal{V} \cup T)^*$. The elements of \mathcal{V} are called *variables*, and the elements of T are called *terminals*. If P stands for the production $X \rightarrow h$ and $u, v \in (\mathcal{V} \cup T)^*$, we write

$$u \xrightarrow{P} v$$

to mean that there are words $p, q \in (\mathcal{V} \cup T)^*$ such that $u = pXq$ and $v = phq$. In other words, v results from u by replacing the variable X by the word h . Productions $X \rightarrow 0$ are called *null productions*. A *context-free grammar* Γ with variables \mathcal{V} and terminals T consists of a finite set of context-free productions on \mathcal{V}, T together with a designated symbol $S \in \mathcal{V}$ called the *start symbol*. Collectively, the set $\mathcal{V} \cup T$ is called the *alphabet* of Γ . If none of the productions of Γ is a null production, Γ is called a *positive context-free grammar*.¹

If Γ is a context-free grammar with variables \mathcal{V} and terminals T , and if $u, v \in (\mathcal{V} \cup T)^*$, we write

$$u \xrightarrow{\Gamma} v$$

¹ Those who have read Chapter 7 should note that every positive context-free grammar is a context-sensitive grammar in the sense defined there. For the moment we are not assuming familiarity with Chapter 7. However, the threads will all be brought together in the next chapter.

to mean that $u \xrightarrow{P} v$ for some production P of Γ . We write

$$u \xrightarrow[\Gamma]{} v$$

to mean there is a sequence u_1, \dots, u_m where $u = u_1, u_m = v$, and

$$u_i \xrightarrow[\Gamma]{} u_{i+1} \quad \text{for } 1 \leq i < m.$$

The sequence u_1, \dots, u_m is called a *derivation of v from u in Γ* . The number m is called the *length* of the derivation.² The symbol Γ below the “ \Rightarrow ” may be omitted when no ambiguity results. Finally, we define

$$L(\Gamma) = \{u \in T^* \mid S \xrightarrow{\Gamma} u\}.$$

$L(\Gamma)$ is called the language *generated* by Γ . A language $L \subseteq T^*$ is called *context-free* if there is a context-free grammar Γ such that $L = L(\Gamma)$.

A simple example of a context-free grammar Γ is given by $V = \{S\}$, $T = \{a, b\}$, and the productions

$$S \rightarrow aSb, \quad S \rightarrow ab.$$

Here we clearly have

$$L(\Gamma) = \{a^{[n]}b^{[n]} \mid n > 0\};$$

thus, this language is context-free. By Chapter 8, Exercise 6 in Section 7, $L(\Gamma)$ is not regular. Later we shall see that every regular language is context-free. For the meanwhile we have proved

Theorem 1.1. The language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ is context-free but not regular.

We now wish to discuss the relation between context-free grammars in general and positive context-free grammars. It is obvious that if Γ is a positive context-free grammar, then $0 \notin L(\Gamma)$. We shall show that except for this limitation, everything that can be done using context-free grammars can be done with positive context-free grammars. This will require some messy technicalities, but working out the details now will simplify matters later.

Definition. We define the *kernel* of a given context-free grammar Γ , written $\ker(\Gamma)$, to be the set of variables V of Γ such that $V \xrightarrow[\Gamma]{} 0$.

As an example consider the context-free grammar Γ_0 with productions

$$S \rightarrow XYYX, \quad S \rightarrow aX, \quad X \rightarrow 0, \quad Y \rightarrow 0.$$

² Some authors use the number $m - 1$ as the length of the derivation.

Then $\ker(\Gamma_0) = \{X, Y, S\}$. This example suggests an algorithm for locating the elements of $\ker(\Gamma)$ for a given context-free grammar Γ . We let

$$\mathcal{V}_0 = \{V \mid V \rightarrow 0 \text{ is a production of } \Gamma\},$$

$$\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{V \mid V \rightarrow \alpha \text{ is a production of } \Gamma, \text{ where } \alpha \in \mathcal{V}_i^*\}.$$

Thus for Γ_0 , $\mathcal{V}_0 = \{X, Y\}$, $\mathcal{V}_1 = \{X, Y, S\}$, and $\mathcal{V}_i = \mathcal{V}_1$ for all $i > 1$. S is in \mathcal{V}_1 because $XYYX \in \mathcal{V}_0^*$. In the general case it is clear, because Γ has only finitely many variables, that a stage k will eventually be reached for which $\mathcal{V}_{k+1} = \mathcal{V}_k$ and that then $\mathcal{V}_i = \mathcal{V}_k$ for all $i > k$. We have

Lemma 1. If $\mathcal{V}_k = \mathcal{V}_{k+1}$, then $\ker(\Gamma) = \mathcal{V}_k$.

Proof. It is clear that $\mathcal{V}_i \subseteq \ker(\Gamma)$ for all i . Conversely, we show that if $V \in \ker(\Gamma)$, then $V \in \mathcal{V}_k$. We prove this by induction on the length of a derivation of 0 from V in Γ . If there is such a derivation of length 2, then $V \xrightarrow{\Gamma} 0$, so that $V \rightarrow 0$ is a production of Γ and $V \in \mathcal{V}_0$. Let us assume the result for all derivations of length $< r$ and let $V = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{r-1} \Rightarrow \alpha_r = 0$ be a derivation of length r in Γ . The words $\alpha_1, \alpha_2, \dots, \alpha_{r-1}$ must consist entirely of variables, since terminals cannot be eliminated by context-free productions. Let $\alpha_2 = V_1 V_2 \dots V_s$. Then we have $V_i \xrightarrow{\Gamma} 0$, $i = 1, 2, \dots, s$, by derivations of length $< r$. By the induction hypothesis, each $V_i \in \mathcal{V}_k$. Since Γ contains the production $V \rightarrow V_1 V_2 \dots V_s$, and $\alpha_2 \in \mathcal{V}_k^*$, we have $V \in \mathcal{V}_{k+1} = \mathcal{V}_k$. ■

Lemma 2. There is an algorithm which will transform a given context-free grammar Γ into a positive context-free grammar $\bar{\Gamma}$ such that $L(\Gamma) = L(\bar{\Gamma})$ or $L(\Gamma) = L(\bar{\Gamma}) \cup \{0\}$.

Proof. We begin by computing $\ker(\Gamma)$. Then we obtain $\bar{\Gamma}$ by first deleting from Γ all productions of the form $V \rightarrow 0$ and then adding all productions that can be obtained from the remaining productions of Γ by deleting from their right-hand sides one or more variables belonging to $\ker(\Gamma)$. (In our example, $\bar{\Gamma}_0$ would have the productions $S \rightarrow XYYX$, $S \rightarrow aX$, $S \rightarrow a$, $S \rightarrow YYX$, $S \rightarrow XYX$, $S \rightarrow XYY$, $S \rightarrow XY$, $S \rightarrow YY$, $S \rightarrow YX$, $S \rightarrow XX$, $S \rightarrow X$, $S \rightarrow Y$.) We shall show that $L(\Gamma) = L(\bar{\Gamma})$ or $L(\Gamma) = L(\bar{\Gamma}) \cup \{0\}$.

Let $V \rightarrow \beta_1 \beta_2 \dots \beta_s$ be a production of $\bar{\Gamma}$ which is not a production of Γ , where $\beta_1, \beta_2, \dots, \beta_s \in (\mathcal{V} \cup T)^*$, and where this production was obtained from a production of Γ of the form

$$V \rightarrow u_0 \beta_1 u_1 \beta_2 \dots \beta_s u_s,$$

with $u_0, u_1, u_2, \dots, u_s \in (\ker(\Gamma))^*$. [Of course, u_0, u_s might be 0. But since $0 \in (\ker(\Gamma))^*$, this creates no difficulty.] Now,

$$u_i \xrightarrow{\Gamma} 0, \quad i = 0, 1, 2, \dots, s,$$

so that

$$V \underset{\Gamma}{\Rightarrow} u_0 \beta_1 u_1 \beta_2 \cdots u_{s-1} \beta_s u_s \underset{\Gamma}{\Rightarrow} \beta_1 \beta_2 \cdots \beta_s.$$

Thus, the effect of this new production of $\bar{\Gamma}$ can be simulated in Γ . This proves that $L(\bar{\Gamma}) \subseteq L(\Gamma)$.

It remains to show that if $v \in L(\Gamma)$ and $v \neq 0$, then $v \in L(\bar{\Gamma})$. Let T be the set of terminals of Γ (and also of $\bar{\Gamma}$). We shall prove by induction the stronger assertion:

For any variable V , if $V \underset{\Gamma}{\Rightarrow} w \neq 0$ for $w \in T^$, then $V \underset{\bar{\Gamma}}{\Rightarrow} w$.*

If in fact $V \underset{\Gamma}{\Rightarrow} w$, then Γ contains the production $V \rightarrow w$ which is also a production of $\bar{\Gamma}$. So we may write

$$V \underset{\Gamma}{\Rightarrow} w_0 V_1 w_1 V_2 w_2 \cdots V_s w_s \underset{\Gamma}{\Rightarrow} w,$$

where V_1, \dots, V_s are variables and $w_0, w_1, w_2, \dots, w_s$ are (possibly null) words on the terminals. Then w can be written

$$w = w_0 v_1 w_1 v_2 w_2 \cdots v_s w_s,$$

where

$$V_i \underset{\Gamma}{\Rightarrow} v_i, \quad i = 1, 2, \dots, s.$$

Since each v_i must have a shorter derivation from V_i than w has from V , we may proceed inductively by assuming that for each v_i which is not 0, $V_i \underset{\bar{\Gamma}}{\Rightarrow} v_i$. On the other hand, if $v_i = 0$, then $V_i \in \text{ker}(\Gamma)$. We set

$$V_i^0 = \begin{cases} 0 & \text{if } v_i = 0 \\ V_i & \text{otherwise.} \end{cases}$$

Then $V \rightarrow w_0 V_1^0 w_1 V_2^0 w_2 \cdots V_s^0 w_s$ is one of the productions of $\bar{\Gamma}$. Hence we have

$$V \underset{\bar{\Gamma}}{\Rightarrow} w_0 V_1^0 w_1 V_2^0 w_2 \cdots V_s^0 w_s \underset{\bar{\Gamma}}{\Rightarrow} w_0 v_1 w_1 v_2 w_2 \cdots v_s w_s = w. \quad \blacksquare$$

We can now easily prove

Theorem 1.2. A language L is context-free if and only if there is a positive context-free grammar Γ such that

$$L = L(\Gamma) \text{ or } L = L(\Gamma) \cup \{0\}. \quad (1.1)$$

Moreover, there is an algorithm which will transform a context-free grammar Δ for which $L = L(\Delta)$ into a positive context-free grammar Γ which satisfies (1.1).

Proof. If L is context-free with $L = L(\Delta)$ for a context-free grammar Δ , then we can use the algorithm of Lemma 2 to construct a positive context-free grammar Γ such that $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Conversely, if Γ is a positive context-free grammar and $L = L(\Gamma)$, there is nothing to prove since a positive context-free grammar is already a context-free grammar. If $L = L(\Gamma) \cup \{0\}$, let S be the start symbol of Γ and let $\tilde{\Gamma}$ be the context-free grammar obtained from Γ by introducing \tilde{S} as a new start symbol and adding the productions

$$\tilde{S} \rightarrow S, \quad \tilde{S} \rightarrow 0.$$

Clearly, $L(\tilde{\Gamma}) = L(\Gamma) \cup \{0\}$. ■

Now, let Γ be a *positive* context-free grammar with alphabet $T \cup \mathcal{V}$, where T consists of the terminals and \mathcal{V} is the set of variables. We will make use of *trees* consisting of a finite number of points called *nodes* or *vertices*, each of which is labeled by a letter of the alphabet, i.e., an element of $T \cup \mathcal{V}$. Certain vertices will have other nodes as *immediate successors*, and the immediate successors of a given node are to be in some definite order. It is helpful (though of course not part of the formal development) to think of the immediate successors of a given node as being physically *below* the given node and arranged from left to right in their given order. Nodes are to be connected by line segments to their immediate successors. There is to be exactly one node which is not an immediate successor; this node is called the *root*. Each node other than the root is to be the immediate successor of precisely one node, its *predecessor*. Nodes which have no immediate successors are called *leaves*.

A tree is called a Γ -tree if it satisfies the following conditions:

1. the root is labeled by a variable;
2. each vertex which is not a leaf is labeled by a variable;
3. if a vertex is labeled X and its immediate successors are labeled $\alpha_1, \alpha_2, \dots, \alpha_k$ (reading from left to right), then $X \rightarrow \alpha_1 \alpha_2 \dots \alpha_k$ is a production of Γ .

Let \mathcal{T} be a Γ -tree, and let v be a vertex of \mathcal{T} which is labeled by the variable X . Then we shall speak of the *subtree* \mathcal{T}^v of \mathcal{T} determined by v . The vertices of \mathcal{T}^v are v , its immediate successors in \mathcal{T} , their immediate successors, and so on. The vertices of \mathcal{T}^v are labeled exactly as they are in \mathcal{T} . (In particular, the root of \mathcal{T}^v is v which is labeled X .) Clearly, \mathcal{T}^v is itself a Γ -tree.

If \mathcal{T} is a Γ -tree, we write $\langle \mathcal{T} \rangle$ for the word which consists of the *labels of the leaves of \mathcal{T}* reading from left to right (a vertex to the left of a given node is regarded as also being to the left of each of its immediate successors).

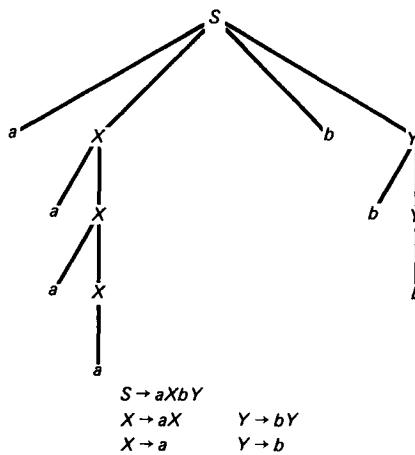


Fig. 1.1. A derivation tree for $a^{[4]}b^{[3]}$ in the indicated grammar.

If the root of \mathcal{T} is labeled by the start symbol S of Γ and if $w = \langle \mathcal{T} \rangle$, then \mathcal{T} is called a *derivation tree for w in Γ* . Thus the tree shown in Fig. 1.1 is a derivation tree for $a^{[4]}b^{[3]}$ in the grammar shown in the same figure.

Theorem 1.3. If Γ is a positive context-free grammar, and $S \xrightarrow{\Gamma} w$, then there is a derivation tree for w in Γ .

Proof. Our proof is by induction on the length of a derivation of w from S in Γ . If this length is 1, then $w = S$ and the required derivation tree consists of a single vertex labeled S (being both root and leaf).

Now let w have a derivation from S of length $r + 1$, where the result is known for derivations of length r . Then we have $S \xrightarrow{\Gamma} v \Rightarrow w$ with $v, w \in (\mathcal{V} \cup \mathcal{T})^*$, where the induction hypothesis applies to the derivation $S \xrightarrow{\Gamma} v$. Thus, we may assume that we have a derivation tree for v . Now since $v \Rightarrow w$, we must have $v = xXy$ and $w = x\alpha_1 \cdots \alpha_k y$, where Γ contains the production $X \rightarrow \alpha_1 \cdots \alpha_k$. Then the derivation tree for v can be extended to yield a derivation tree for w simply by giving k immediate successors to the node labeled X , labeled $\alpha_1, \dots, \alpha_k$ from left to right. ■

Before considering the converse of Theorem 1.3, it will be helpful to consider the following derivations of $a^{[4]}b^{[3]}$ from S with respect to the grammar indicated in Fig. 1.1:

1. $S \Rightarrow aXbY \Rightarrow a^{[2]}XbY \Rightarrow a^{[3]}XbY \Rightarrow a^{[4]}bY \Rightarrow a^{[4]}b^{[2]}Y \Rightarrow a^{[4]}b^{[3]}$
2. $S \Rightarrow aXbY \Rightarrow a^{[2]}XbY \Rightarrow a^{[2]}Xb^{[2]}Y \Rightarrow a^{[3]}Xb^{[2]}Y \Rightarrow a^{[3]}Xb^{[3]} \Rightarrow a^{[4]}b^{[3]}$
3. $S \Rightarrow aXbY \Rightarrow aXb^{[2]}Y \Rightarrow aXb^{[3]} \Rightarrow a^{[2]}Xb^{[3]} \Rightarrow a^{[3]}Xb^{[3]} \Rightarrow a^{[4]}b^{[3]}$.

Now, if the proof of Theorem 1.3 is applied to these three derivations, the very same derivation tree is obtained—namely, the one shown in Fig. 1.1. This shows that there does not exist a one-one correspondence between derivations and derivation trees, but that rather, several derivations may give rise to the same tree. Hence, there is no unique derivation which we can hope to be able to read off a given derivation tree.

Definition. We write $u \Rightarrow_l v$ (in Γ) if $u = xXy$ and $v = xzy$, where $X \rightarrow z$ is a production of Γ and $x \in T^*$. If, instead, $x \in (T \cup \mathcal{V})^*$ but $y \in T^*$, we write $u \Rightarrow_r v$.

Thus, when $u \Rightarrow_l v$, it is the *leftmost* variable in u for which a substitution is made, whereas when $u \Rightarrow_r v$, it is the *rightmost* variable in u . A derivation

$$u_1 \Rightarrow_l u_2 \Rightarrow_l u_3 \Rightarrow_l \cdots \Rightarrow_l u_n$$

is called a *leftmost* derivation, and then we write $u_1 \overset{*}{\Rightarrow}_l u_n$. Similarly, a derivation

$$u_1 \Rightarrow_r u_2 \Rightarrow_r u_3 \Rightarrow_r \cdots \Rightarrow_r u_n$$

is called a *rightmost* derivation, and we write $u_1 \overset{*}{\Rightarrow}_r u_n$. In the above examples of derivations of $a^{[4]}b^{[3]}$ from S in the grammar of Fig. 1.1, (1) is leftmost, (3) is rightmost, and (2) is neither.

Now we shall see how, given a derivation tree \mathcal{T} for a word $w \in T^*$, we can obtain a *leftmost derivation of w from S* and a *rightmost derivation of w from S*. Let the word which consists of the labels of the immediate successors of the root of \mathcal{T} (reading from left to right) be $v_0 X_1 v_1 X_2 \cdots X_r v_r$, where $v_0, v_1, \dots, v_r \in T^*$, $X_1, X_2, \dots, X_r \in \mathcal{V}$, and X_1, X_2, \dots, X_r label the vertices v_1, \dots, v_r , which are immediate successors of the root of \mathcal{T} . (Of course, some of the v_i may be 0.) Then $S \rightarrow v_0 X_1 v_1 X_2 \cdots X_r v_r$ is one of the productions of Γ . Now it is possible that the immediate successors of the root of \mathcal{T} are all leaves; this is precisely the case where $w = v_0$ and $r = 0$. If this is the case, then we have $S \Rightarrow_l w$ and $S \Rightarrow_r w$, so that we do have a leftmost as well as a rightmost derivation of w from S .

Otherwise, i.e., if $r > 0$, we consider the trees $\mathcal{T}_i = \mathcal{T}^{v_i}$, $i = 1, 2, \dots, r$. Here \mathcal{T}_i has its root v_i labeled X_i , and is made up of the part of \mathcal{T} consisting of v_i , its immediate successors, their immediate successors, etc. (See Fig. 1.2). Let Γ_i be the grammar whose productions and alphabet are the same as for Γ but which has start symbol X_i . Then \mathcal{T}_i is a derivation tree in Γ_i . Let \mathcal{T}_i be a derivation tree for w_i in Γ_i . Then, clearly,

$$w = v_0 w_1 v_1 w_2 v_2 \cdots w_r v_r.$$

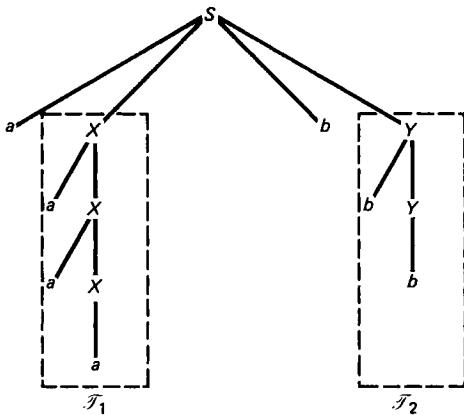


Fig. 1.2. Decomposition of the tree of Fig. 1.1 as in the proof of the existence of leftmost and rightmost derivations.

Moreover, since each \mathcal{T}_i contains fewer vertices than \mathcal{T} , we may assume inductively that for $i = 1, 2, \dots, r$

$$X_i \xrightarrow{*l} w_i \quad \text{and} \quad X_i \xrightarrow{*r} w_i.$$

Hence we have

$$\begin{aligned} S &\Rightarrow_l v_0 X_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*l} v_0 w_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*l} v_0 w_1 v_1 w_2 \cdots X_r v_r \\ &\vdots \\ &\xrightarrow{*l} v_0 w_1 v_1 w_2 \cdots w_r v_r = w \end{aligned}$$

and

$$\begin{aligned} S &\Rightarrow_r v_0 X_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*r} v_0 X_1 v_1 X_2 \cdots w_r v_r \\ &\vdots \\ &\xrightarrow{*r} v_0 X_1 v_1 w_2 \cdots w_r v_r \\ &\xrightarrow{*r} v_0 w_1 v_1 w_2 \cdots w_r v_r = w. \end{aligned}$$

So we have shown how to obtain a leftmost and a rightmost derivation of w from S in Γ .

Now, Theorem 1.3 tells us that if $w \in L(\Gamma)$, there is a derivation tree for w in Γ . And we have just seen that if there is a derivation tree for w in Γ , then there are both leftmost and rightmost derivations of w from S in Γ [so that, in particular, $w \in L(\Gamma)$]. Putting all of this information together we have

Theorem 1.4. Let Γ be a positive context-free grammar with start symbol S and terminals T . Let $w \in T^*$. Then the following conditions are equivalent:

1. $w \in L(\Gamma)$;
2. there is a derivation tree for w in Γ ;
3. there is a leftmost derivation of w from S in Γ ;
4. there is a rightmost derivation of w from S in Γ .

Definition. A positive context-free grammar is called *branching* if it has no productions of the form $X \rightarrow Y$, where X and Y are variables.

For a derivation tree in a branching grammar Γ , each vertex which is not a leaf cannot be the *only* immediate successor of its predecessor. Since we shall find it useful to work with branching grammars, we prove

Theorem 1.5. There is an algorithm that transforms a given positive context-free grammar Γ into a branching context-free grammar Δ such that $L(\Delta) = L(\Gamma)$.

Proof. Let \mathcal{V} be the set of variables of Γ . First suppose that Γ contains productions

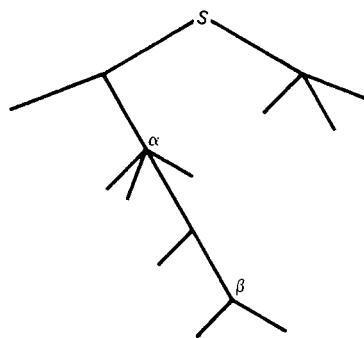
$$X_1 \rightarrow X_2, \quad X_2 \rightarrow X_3, \quad \dots, \quad X_k \rightarrow X_1, \tag{1.2}$$

where $k \geq 1$ and $X_1, X_2, \dots, X_k \in \mathcal{V}$. Then, we can eliminate the productions (1.2) and replace each variable X_i in the remaining productions of Γ by the new variable X . (If one of X_1, \dots, X_k is the start symbol, then X must now be the start symbol.) Obviously the language generated is not changed by this transformation.

Thus, we need only consider the case where no “cycles” like (1.2) occur in Γ . If Γ is not branching, it must contain a production $X \rightarrow Y$ such that Γ contains no productions of the form $Y \rightarrow Z$. We eliminate the production $X \rightarrow Y$, but add to Γ productions $X \rightarrow x$ for each word $x \in (\mathcal{V} \cup T)^*$ for which $Y \rightarrow x$ is a production of Γ . Again the language generated is unchanged, but the number of productions that Γ contains of the form $U \rightarrow V$ has been decreased. Iterating this process we arrive at a grammar Δ containing no productions of the form $U \rightarrow V$, which is therefore of the required form. ■

A *path* in a Γ -tree \mathcal{T} is a sequence $\alpha_1, \alpha_2, \dots, \alpha_k$ of vertices of \mathcal{T} such that α_{i+1} is an immediate successor of α_i for $i = 1, 2, \dots, k - 1$. All of the vertices on the path are called *descendants* of α_1 .

A particularly interesting situation arises when two different vertices α, β lie on the same path in the derivation tree \mathcal{T} and are labeled by the same variable X . In such a case one of the vertices is a descendant of the other, say, β is a descendant of α . \mathcal{T}^β is then not only a subtree of \mathcal{T} but also of \mathcal{T}^α . [In fact, $(\mathcal{T}^\alpha)^\beta = \mathcal{T}^\beta$.] We wish to consider two important operations on the derivation tree \mathcal{T} which can be performed in this case. The first operation, which we call *pruning*, is to remove the subtree \mathcal{T}^α from the vertex α and to *graft* the subtree \mathcal{T}^β in its place. The second operation, which we call *splicing*, is to remove the subtree \mathcal{T}^β from the vertex β and to *graft* an exact copy of \mathcal{T}^α in its place. (See Fig. 1.3.) Because α and β are labeled by the same variable, the trees obtained by pruning and splicing are themselves derivation trees.



Original tree \mathcal{T}
(α, β are labeled by the same variable)

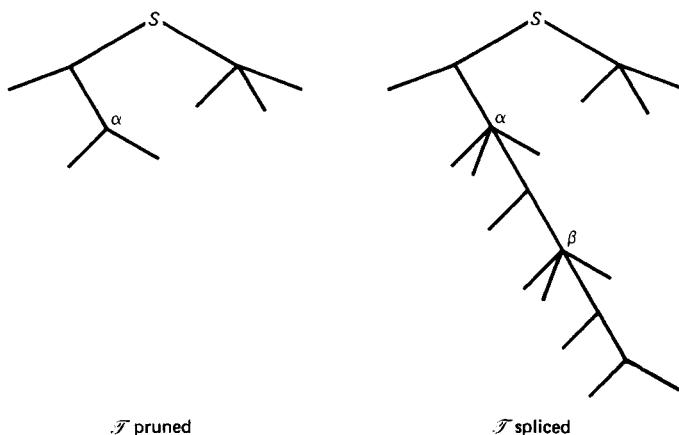


Fig. 1.3

Let \mathcal{T}_p and \mathcal{T}_s be trees obtained from a derivation tree \mathcal{T} in a *branching* grammar by pruning and splicing, respectively, where α and β are as above. We have $\langle \mathcal{T} \rangle = r_1 \langle \mathcal{T}^\alpha \rangle r_2$ for words r_1, r_2 and $\langle \mathcal{T}^\alpha \rangle = q_1 \langle \mathcal{T}^\beta \rangle q_2$ for words q_1, q_2 . Since α, β are distinct vertices, and since the grammar is branching, q_1 and q_2 cannot both be 0. (That is, $q_1 q_2 \neq 0$.) Also,

$$\langle \mathcal{T}_p \rangle = r_1 \langle \mathcal{T}^\beta \rangle r_2 \quad \text{and} \quad \langle \mathcal{T}_s \rangle = r_1 q_1^{[2]} \langle \mathcal{T}^\beta \rangle q_2^{[2]} r_2. \quad (1.3)$$

Since $q_1 q_2 \neq 0$, we have $|\langle \mathcal{T}^\beta \rangle| < |\langle \mathcal{T}^\alpha \rangle|$ and hence $|\langle \mathcal{T}_p \rangle| < |\langle \mathcal{T} \rangle|$. From this last inequality and Theorem 1.4, we can easily infer:

Theorem 1.6. Let Γ be a branching context-free grammar, let $u \in L(\Gamma)$, and let u have a derivation tree \mathcal{T} in Γ which has two different vertices on the same path labeled by the same variable. Then there is a word $v \in L(\Gamma)$ such that $|v| < |u|$.

Proof. Since $u = \langle \mathcal{T} \rangle$, we need only take $v = \langle \mathcal{T}_p \rangle$. ■

Exercises

1. Find a context-free grammar generating the set of arithmetic statements of FORTRAN (or BASIC or PL/I or Pascal).

2. Consider the grammar Γ with start symbol S and productions

$$\begin{array}{lll} S \rightarrow XXYY & X \rightarrow XX & Y \rightarrow YY \\ X \rightarrow a & & Y \rightarrow b. \end{array}$$

Show that Γ generates the same language as the grammar of Fig. 1.1.

2. Regular Grammars

We shall now see that regular languages are generated by context-free grammars of an especially simple form.

Definition. A context-free grammar is called *regular* if each of its productions has one of the two forms

$$U \rightarrow aV \quad \text{or} \quad U \rightarrow a,$$

where U, V are variables and a is a terminal.

Then we have

Theorem 2.1. If L is a regular language, then there is a regular grammar Γ such that either $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Proof. Let $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with states q_1, \dots, q_m , alphabet $\{s_1, \dots, s_n\}$, transition function δ , and set of accepting states F . We construct a grammar Γ with variables q_1, \dots, q_m , terminals s_1, \dots, s_n , and start symbol q_1 . The productions are

1. $q_i \rightarrow s_r q_j$ whenever $\delta(q_i, s_r) = q_j$, and
2. $q_i \rightarrow s_r$ whenever $\delta(q_i, s_r) \in F$.

Clearly the grammar Γ is regular. We shall show that $L(\Gamma)$ is just $L - \{0\}$.

First, suppose $u \in L$, $u \neq 0$; let $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}$. Thus, $\delta^*(q_1, u) \in F$, so that we have

$$\delta(q_1, s_{i_1}) = q_{j_1}, \quad \delta(q_{j_1}, s_{i_2}) = q_{j_2}, \quad \dots, \quad \delta(q_{j_l}, s_{i_{l+1}}) = q_{j_{l+1}} \in F. \quad (2.1)$$

Hence, the grammar Γ contains the productions

$$q_1 \rightarrow s_{i_1} q_{j_1}, \quad q_{j_1} \rightarrow s_{i_2} q_{j_2}, \quad \dots, \quad q_{j_{l-1}} \rightarrow s_{i_l} q_{j_l}, \quad q_{j_l} \rightarrow s_{i_{l+1}}. \quad (2.2)$$

Thus, we have in Γ

$$\begin{aligned} q_1 &\Rightarrow s_{i_1} q_{j_1} \\ &\Rightarrow s_{i_1} s_{i_2} q_{j_2} \\ &\vdots \\ &\Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} q_{j_l} \\ &\Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}} = u, \end{aligned} \quad (2.3)$$

so that $u \in L(\Gamma)$.

Conversely, suppose that $u \in L(\Gamma)$, $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}$. Then there is a derivation of u from q_1 in Γ , which must be of the form (2.3). Hence, the productions listed in (2.2) must belong to Γ , and finally, the transitions (2.1) must hold in \mathcal{M} . Thus, $u \in L(\mathcal{M})$. ■

Theorem 2.2. Let Γ be a regular grammar. Then $L(\Gamma)$ is a regular language.

Proof. Let Γ have the *variables* V_1, V_2, \dots, V_K , where $S = V_1$ is the start symbol, and the *terminals* s_1, \dots, s_n . Since Γ is assumed to be regular, its productions are of the form $V_i \rightarrow s_r V_j$ and $V_i \rightarrow s_r$. We shall construct an ndfa \mathcal{M} which accepts precisely $L(\Gamma)$.

The states of \mathcal{M} will be V_1, V_2, \dots, V_K and an additional state W . V_1 will be the initial state and W will be the only accepting state, i.e., $F = \{W\}$.

Let

$$\delta_1(V_i, s_r) = \{V_j \mid V_i \rightarrow s_r V_j \text{ is a production of } \Gamma\},$$

$$\delta_2(V_i, s_r) = \begin{cases} \{W\} & \text{if } V_i \rightarrow s_r \text{ is a production of } \Gamma \\ \emptyset & \text{otherwise.} \end{cases}$$

Then we take as the transition function δ of \mathcal{M}

$$\delta(V_i, s_r) = \delta_1(V_i, s_r) \cup \delta_2(V_i, s_r).$$

This completes the specification of \mathcal{M} .

Now let $u = s_{i_1}s_{i_2}\cdots s_{i_l}s_{i_{l+1}} \in L(\Gamma)$. Thus, we must have

$$V_1 \Rightarrow s_{i_1}V_{j_1} \Rightarrow s_{i_1}s_{i_2}V_{j_2} \xrightarrow{*} s_{i_1}s_{i_2}\cdots s_{i_l}V_{j_l} \Rightarrow s_{i_1}s_{i_2}\cdots s_{i_l}s_{i_{l+1}}, \quad (2.4)$$

where Γ contains the productions

$$\begin{aligned} V_1 &\rightarrow s_{i_1}V_{j_1}, \\ V_{j_1} &\rightarrow s_{i_2}V_{j_2}, \\ &\vdots \\ V_{j_{l-1}} &\rightarrow s_{i_l}V_{j_l}, \\ V_{j_l} &\rightarrow s_{i_{l+1}}. \end{aligned} \quad (2.5)$$

Thus,

$$\begin{aligned} V_{j_1} &\in \delta(V_1, s_{i_1}), \\ V_{j_2} &\in \delta(V_{j_1}, s_{i_2}), \\ &\vdots \\ V_{j_{l-1}} &\in \delta(V_{j_{l-1}}, s_{i_l}), \\ W &\in \delta(V_{j_l}, s_{i_{l+1}}). \end{aligned} \quad (2.6)$$

Thus, $W \in \delta^*(V_1, u)$ and $u \in L(\mathcal{M})$.

Conversely, if $u = s_{i_1}s_{i_2}\cdots s_{i_l}s_{i_{l+1}}$ is accepted by \mathcal{M} , then there must be a sequence of transitions of the form (2.6). Hence, the productions of (2.5) must all belong to Γ , so that there is a derivation of the form (2.4) of u from V_1 . ■

In order to combine Theorems 2.1 and 2.2 in a single equivalence, it is only necessary to show that if L is a regular language, then so is $L \cup \{0\}$. But this follows at once from Theorems 4.2 and 4.5 in Chapter 8.

Combining Theorems 2.1 and 2.2 with this discussion, we have

Theorem 2.3. A language L is regular if and only if there is a regular grammar Γ such that either $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Since regular grammars are context-free grammars, we have

Corollary 2.4. Every regular language is context-free.

The converse of Corollary 2.4 is not true, however, as we have already observed in Theorem 1.1.

There are more extensive classes of context-free grammars which can be shown to generate only regular languages. A particularly important example for us is the class of *right-linear* grammars:

Definition. A context-free grammar is called *right-linear* if each of its productions has one of the two forms

$$U \rightarrow xV \quad \text{or} \quad U \rightarrow x, \quad (2.7)$$

where U, V are variables and $x \neq 0$ is a word consisting entirely of terminals.

Thus a regular grammar is just a right-linear grammar in which $|x| = 1$ for each string x in (2.7). We have

Theorem 2.5. Let Γ be a right-linear grammar. Then $L(\Gamma)$ is regular.

Proof. Given a right-linear grammar Γ , we construct a regular grammar $\bar{\Gamma}$ as follows:

We replace each production of Γ of the form

$$U \rightarrow a_1 a_2 \cdots a_n V, \quad n > 1,$$

by the productions

$$\begin{aligned} U &\rightarrow a_1 Z_1, \\ Z_1 &\rightarrow a_2 Z_2, \\ &\vdots \\ Z_{n-2} &\rightarrow a_{n-1} Z_{n-1}, \\ Z_{n-1} &\rightarrow a_n V, \end{aligned}$$

where Z_1, \dots, Z_{n-1} are new variables. Also, we replace each production

$$U \rightarrow a_1 a_2 \cdots a_n, \quad n > 1,$$

by a list of productions similar to the list above except that instead of the last production we have

$$Z_{n-1} \rightarrow a_n.$$

It is obvious that $\bar{\Gamma}$ is regular and that $L(\bar{\Gamma}) = L(\Gamma)$. ■

Exercises

1. (a) Write a context-free grammar to generate all and only regular expressions over the alphabet $\{a, b\}$.
 (b) Can a regular grammar generate this language? Support your answer.
2. A grammar Γ is *self-embedding* if there is a variable X such that

$$X \xrightarrow[\Gamma]{} vXw, \quad \text{where } v, w \in (\mathcal{V} \cup T)^* - \{0\}.$$

Let L be a context-free language. Prove that L is regular if and only if there is a non-self-embedding context-free grammar Γ such that $L(\Gamma) = L$.

3. Chomsky Normal Form

Although context-free grammars are extremely simple, there are even simpler special classes of context-free grammars which suffice to give all context-free languages. Such classes are called *normal forms*.

Definition. A context-free grammar Γ with variables \mathcal{V} and terminals T is in *Chomsky normal form* if each of its productions has one of the two forms

$$X \rightarrow YZ \quad \text{or} \quad X \rightarrow a,$$

where $X, Y, Z \in \mathcal{V}$ and $a \in T$.

Then we can prove

Theorem 3.1. There is an algorithm that transforms a given positive context-free grammar Γ into a Chomsky normal form grammar Δ such that $L(\Gamma) = L(\Delta)$.

Proof. Using Theorem 1.5, we begin with a branching context-free grammar Γ with variables \mathcal{V} and terminals T . We continue by “disguising” the terminals as variables. That is, for each $a \in T$ we introduce a new variable X_a . Then we modify Γ by replacing each production $X \rightarrow x$ for which x is not a single terminal by $X \rightarrow x'$, where x' is obtained from x by replacing each terminal a by the corresponding new variable X_a . In addition all of the productions $X_a \rightarrow a$ are added. Clearly the grammar thus obtained generates the same language as Γ and has all of its productions in one of the two forms

$$X \rightarrow X_1 X_2 \dots X_k, \quad k \geq 2, \tag{3.1}$$

$$X \rightarrow a, \tag{3.2}$$

where X, X_1, \dots, X_k are variables and a is a terminal. To obtain a Chomsky normal form grammar we need to eliminate all of the productions of type (3.1) for which $k > 2$. We can do this by introducing the new variables Z_1, Z_2, \dots, Z_{k-2} and replacing (3.1) by the productions

$$\begin{aligned} X &\rightarrow X_1 Z_1 \\ Z_1 &\rightarrow X_2 Z_2 \\ &\vdots \\ Z_{k-3} &\rightarrow X_{k-2} Z_{k-2} \\ Z_{k-2} &\rightarrow X_{k-1} X_k. \end{aligned}$$

Thus we obtain a grammar in Chomsky normal form which generates $L(\Gamma)$. ■

As an example, let us convert the grammar of Fig. 1.1 to Chomsky normal form.

Step 1. Eliminate productions of the form $X_1 \rightarrow X_2$: there are no such productions so we skip this step.

Step 2. Disguise the terminals as variables: the grammar now consists of the productions

$$\begin{array}{lll} S \rightarrow X_a XX_b Y & X \rightarrow X_a X & Y \rightarrow X_b Y \\ X \rightarrow a & Y \rightarrow b & \\ X_a \rightarrow a & X_b \rightarrow b. & \end{array}$$

Step 3. Obtain Chomsky normal form by replacing the production $S \rightarrow X_a XX_b Y$ by the productions

$$\begin{aligned} S &\rightarrow X_a Z_1, \\ Z_1 &\rightarrow XZ_2, \\ Z_2 &\rightarrow X_b Y. \end{aligned}$$

The final Chomsky normal form grammar thus obtained consists of the productions

$$\begin{array}{llll} S \rightarrow X_a Z_1 & Z_1 \rightarrow XZ_2 & Z_2 \rightarrow X_b Y & \\ X \rightarrow X_a X & X \rightarrow a & Y \rightarrow X_b Y & X_b \rightarrow b \\ X_a \rightarrow a & & Y \rightarrow b & \end{array}$$

Exercises

1. (a) Find context-grammars Γ_1, Γ_2 such that

$$L(\Gamma_1) = \{a^{[i]}b^{[j]} \mid i \geq j > 0\}$$

$$L(\Gamma_2) = \{a^{[2i]}b^{[i]} \mid i > 0\}.$$

- (b) Find Chomsky normal form grammars which generate the same languages.

2. Let $T = \{\downarrow, p, q\}$ be the set of terminals for the grammar Γ :

$$S \rightarrow p, \quad S \rightarrow q, \quad S \rightarrow \downarrow SS.$$

Find a Chomsky normal form grammar which generates $L(\Gamma)$.

- 3.* A context-free grammar is said to be in *Greibach normal form* if every production of the grammar is of the form

$$X \rightarrow aY_1 Y_2 \cdots Y_k, \quad k \geq 0,$$

where $a \in T$ and $X, Y_1, Y_2, \dots, Y_k \in \mathcal{V}$. Show that there is an algorithm that transforms any positive context-free grammar into one in Greibach normal form that generates the same language.

- 4.* Show that there is an algorithm that transforms any positive context-free grammar into a grammar that generates the same language for which every production is of the form

$$A \rightarrow a,$$

$$A \rightarrow aB,$$

or

$$A \rightarrow aBC,$$

$$A, B, C \in \mathcal{V}, a \in T.$$

4. Bar-Hillel's Pumping Lemma

An important application of Chomsky normal form is in the proof of the following key theorem, which is an analog for context-free languages of the pumping lemma for regular languages.

Theorem 4.1 (Bar-Hillel's Pumping Lemma). Let Γ be a Chomsky normal form grammar with exactly n variables, and let $L = L(\Gamma)$. Then, for every $x \in L$ for which $|x| > 2^n$, we have $x = r_1 q_1 r q_2 r_2$, where

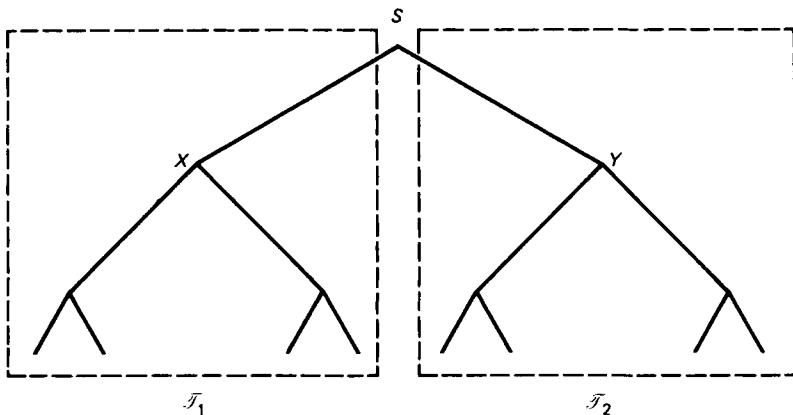


Fig. 4.1

1. $|q_1rq_2| \leq 2^n$;
2. $q_1q_2 \neq 0$;
3. for all $i \geq 0$, $r_1q_1^{[i]}rq_2^{[i]}r_2 \in L$.

Lemma. Let $S \xrightarrow[\Gamma]{} u$, where Γ is a Chomsky normal form grammar. Suppose that \mathcal{T} is a derivation tree for u in Γ and that no path in \mathcal{T} contains more than k nodes. Then $|u| \leq 2^{k-2}$.

Proof. First, suppose that \mathcal{T} has just one leaf labeled by a terminal a . Then $u = a$, and \mathcal{T} has just two nodes which are labeled S and a , respectively. Thus, no path in \mathcal{T} contains more than 2 nodes and $|u| = 1 \leq 2^{2-2}$.

Otherwise, since Γ is in Chomsky normal form, the root of \mathcal{T} must have exactly two immediate successors α, β in \mathcal{T} labeled by variables, say, X and Y , respectively. (In this case, Γ contains the production $S \rightarrow XY$) Now we will consider the two trees $\mathcal{T}_1 = \mathcal{T}^\alpha$ and $\mathcal{T}_2 = \mathcal{T}^\beta$ whose roots are labeled X and Y , respectively. (See Fig. 4.1.)

In each of \mathcal{T}_1 and \mathcal{T}_2 the longest path must contain $\leq k - 1$ nodes. Proceeding inductively, we may assume that each of $\mathcal{T}_1, \mathcal{T}_2$ have $\leq 2^{k-3}$ leaves. Hence,

$$|u| \leq 2^{k-3} + 2^{k-3} = 2^{k-2}. \quad \blacksquare$$

Proof of Theorem 4.1. Let $x \in L$, where $|x| > 2^n$, and let \mathcal{T} be a derivation tree for x in Γ . Let $\alpha_1, \alpha_2, \dots, \alpha_m$ be a path in \mathcal{T} where m is as large as possible. Then $m \geq n + 2$. (For, if $m \leq n + 1$, by the lemma, $|x| \leq 2^{n-1}$.) α_m is a leaf (otherwise we could get a longer path) and so is labeled by a terminal. $\alpha_1, \alpha_2, \dots, \alpha_{m-1}$ are all labeled by variables. Let us write

$$\gamma_i = \alpha_{m+i-n-2}, \quad i = 1, 2, \dots, n+2$$

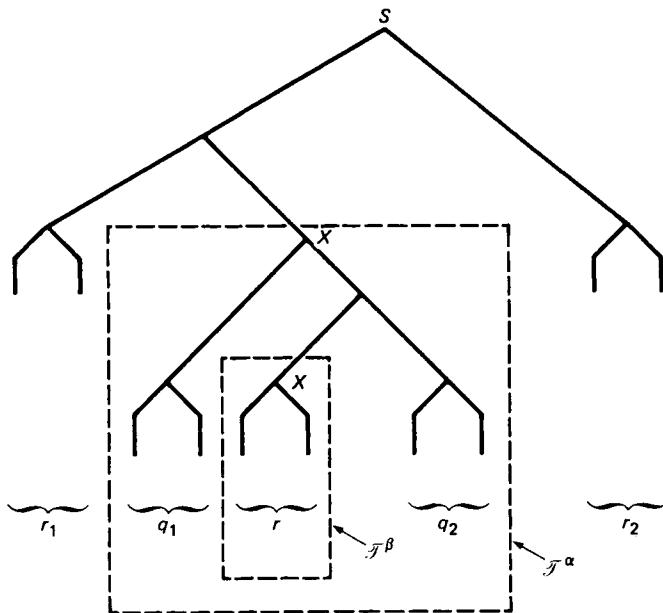


Fig. 4.2

so that the sequence of vertices $\gamma_1, \gamma_2, \dots, \gamma_{n+2}$ is simply the path consisting of the vertices

$$\alpha_{m-n-1}, \alpha_{m-n}, \alpha_{m-n+1}, \dots, \alpha_{m-1}, \alpha_m,$$

where $\gamma_{n+2} = \alpha_m$ is labeled by a terminal, and $\gamma_1, \dots, \gamma_{n+1}$ are labeled by variables. Since there are only n variables in the alphabet of Γ , the pigeon-hole principle guarantees that there is a variable X which labels two different vertices: $\alpha = \gamma_i$ and $\beta = \gamma_j$, $i < j$. (See Fig. 4.2.) Hence, the discussion of *pruning* and *splicing* at the end of Section 1 can be applied. We let the words $q_1 q_2, r_1, r_2$ be defined as in that discussion and set $r = \langle T^\beta \rangle$. Then [recalling (1.3)] we have

$$\begin{aligned}\langle T_p \rangle &= r_1 r r_2, \\ \langle T_s \rangle &= r_1 q_1^{[2]} r q_2^{[2]} r_2, \\ \langle (T_s)_s \rangle &= r_1 q_1^{[3]} r q_2^{[3]} r_2.\end{aligned}$$

Since pruning and splicing a derivation tree in Γ yields a new derivation tree in Γ , we see that all of these words belong to $L(\Gamma)$. If, in addition, we iterate the splicing operation, we see that all of the words $r_1 q_1^{[i]} r q_2^{[i]} r_2$, $i \geq 0$, belong to $L(\Gamma)$.

Finally, we note that the path $\gamma_1, \dots, \gamma_{n+2}$ in \mathcal{T}^α consists of $\leq n + 2$ nodes and that no path in \mathcal{T}^α can be longer. (This is true simply because if there were a path in \mathcal{T}^α consisting of more than $n + 3 - i$ vertices, it could be extended backward through $\alpha = \gamma_i$ to yield a path in \mathcal{T} consisting of more than m vertices.) Hence by the lemma

$$|q_1rq_2| = |q_1\langle \mathcal{F}^\beta \rangle q_2| = |\langle \mathcal{T}^\alpha \rangle| \leq 2^n. \quad \blacksquare$$

As an example of the uses of Bar-Hillel's pumping lemma, we show that the language $L = \{a^{[n]}b^{[n]}c^{[n]} \mid n > 0\}$ is *not* context-free:

Suppose that L is context-free with $L = L(\Gamma)$, where Γ is a Chomsky normal form grammar with n variables. Choose k so large that $|a^{[k]}b^{[k]}c^{[k]}| > 2^n$ (i.e., choose $k > 2^n/3$). Then we would have $a^{[k]}b^{[k]}c^{[k]} = r_1q_1rq_2r_2$, where, setting

$$x_i = r_1q_1^{[i]}rq_2^{[i]}r_2,$$

we have $x_i \in L$ for $i = 0, 1, 2, 3, \dots$. In particular,

$$x_2 = r_1q_1q_1rq_2q_2r_2 \in L.$$

Since the elements of L consist of a block of a 's, followed by a block of b 's, followed by a block of c 's, we see that q_1 and q_2 must each contain only one of these letters. Thus, one of the three letters occurs neither in q_1 nor in q_2 . But since as $i = 2, 3, 4, 5, \dots$, x_i contains more and more copies of q_1 and q_2 and since $q_1q_2 \neq 0$, it is impossible for x_i to have the same number of occurrences of a , b , and c . This contradiction shows that L is not context-free.

We have proved

Theorem 4.2. The language $L = \{a^{[n]}b^{[n]}c^{[n]} \mid n > 0\}$ is not context-free.

Exercises

1. Show that $\{a^{[i]} \mid i \text{ is a prime number}\}$ is not context-free.
2. Show that $\{a^{[i^2]} \mid i > 0\}$ is not context-free.
3. Show that a context-free language on a one-letter alphabet is regular.

5. Closure Properties

We now consider for context-free languages, some of the closure properties previously discussed for regular languages.

Theorem 5.1. If L_1, L_2 are context-free languages, then so is $L_1 \cup L_2$.

Proof. Let $L_1 = L(\Gamma_1), L_2 = L(\Gamma_2)$, where Γ_1, Γ_2 are context-free grammars with disjoint sets of variables \mathcal{V}_1 and \mathcal{V}_2 , and start symbols S_1, S_2 , respectively.

Let Γ be the context-free grammar with variables $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{S\}$ and start symbol S . The productions of Γ are those of Γ_1 and Γ_2 , together with the two additional productions $S \rightarrow S_1, S \rightarrow S_2$. Then obviously $L(\Gamma) = L(\Gamma_1) \cup L(\Gamma_2)$, so that $L_1 \cup L_2 = L(\Gamma)$. ■

Surprisingly enough, the class of context-free languages is not closed under intersection. In fact, let Γ_1 be the context-free grammar whose productions are

$$S \rightarrow Sc, \quad S \rightarrow Xc, \quad X \rightarrow aXb, \quad X \rightarrow ab.$$

Then clearly,

$$L_1 = L(\Gamma_1) = \{a^{[n]}b^{[n]}c^{[m]} \mid n, m > 0\}.$$

Now, let Γ_2 be the grammar whose productions are

$$S \rightarrow aS, \quad S \rightarrow aX, \quad X \rightarrow bXc, \quad X \rightarrow bc.$$

Then

$$L_2 = L(\Gamma_2) = \{a^{[m]}b^{[n]}c^{[n]} \mid n, m > 0\}.$$

Thus, L_1 and L_2 are context-free languages. But

$$L_1 \cap L_2 = \{a^{[n]}b^{[n]}c^{[n]} \mid n > 0\},$$

which, by Theorem 4.2, is not context-free. We have proved

Theorem 5.2. There are context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not context-free.

Corollary 5.3. There is a context-free language $L \subseteq A^*$ such that $A^* - L$ is not context-free.

Proof. Suppose otherwise, i.e., for every context-free language $L \subseteq A^*$, $A^* - L$ is also context-free. Then the De Morgan identity

$$L_1 \cap L_2 = A^* - ((A^* - L_1) \cup (A^* - L_2))$$

together with Theorem 5.1 would contradict Theorem 5.2. ■

Although, as we have just seen, the intersection of context-free languages need not be context-free, the situation is different if one of the two languages is regular:

Theorem 5.4. If R is a regular language and L is a context-free language, then $R \cap L$ is context-free.

Proof. Let A be an alphabet such that $L, R \subseteq A^*$. Let $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$, where Γ is a positive context-free grammar with variables \mathcal{V} , terminals A and start symbol S . Finally, let M be a dfa which accepts R

with states Q , initial state $q_1 \in Q$, accepting states $F \subseteq Q$, and transition function δ . Now, for each symbol $\sigma \in A \cup \mathcal{V}$ and each ordered pair $p, q \in Q$, we introduce a new symbol σ^{pq} . We shall construct a positive context-free grammar $\tilde{\Gamma}$ whose terminals are just the elements of A (i.e., the terminals of Γ) and whose set of variables consists of a start symbol \tilde{S} together with all of the new symbols σ^{pq} for $\sigma \in A \cup \mathcal{V}$ and $p, q \in Q$. (Note that for $a \in A$, a is a terminal, but a^{pq} is a variable for each $p, q \in Q$.) The productions of $\tilde{\Gamma}$ are as follows:

- (1) $\tilde{S} \rightarrow S^{q_1 q}$ for all $q \in F$.
- (2) $X^{pq} \rightarrow \sigma_1^{pr_1} \sigma_2^{r_1 r_2} \cdots \sigma_n^{r_{n-1} q}$ for all productions $X \rightarrow \sigma_1 \sigma_2 \cdots \sigma_n$ of Γ and all $p, r_1, r_2, \dots, r_{n-1}, q \in Q$.
- (3) $a^{pq} \rightarrow a$ for all $a \in A$ and all $p, q \in Q$ such that $\delta(p, a) = q$.

We shall now prove that $L(\tilde{\Gamma}) = L(\Gamma) \cap R$. Since $\tilde{\Gamma}$ is clearly a positive context-free grammar, this will prove the theorem.

First, let $u = a_1 a_2 \cdots a_n \in L(\Gamma) \cap R$. Since $u \in L(\Gamma)$, we have

$$\tilde{S} \xrightarrow[\tilde{\Gamma}]{} a_1 a_2 \cdots a_n.$$

Using productions (1) and (2) of $\tilde{\Gamma}$, we have

$$\tilde{S} \xrightarrow[\tilde{\Gamma}]{} S^{q_1, q_{n+1}} \xrightarrow[\tilde{\Gamma}]{} a_1^{q_1 q_2} a_2^{q_2 q_3} \cdots a_n^{q_n q_{n+1}}, \quad (5.1)$$

where q_2, q_3, \dots, q_n are arbitrary states of \mathcal{M} and q_{n+1} is any state in F . (q_1 is of course the initial state.) But since $u \in L(\mathcal{M})$, we can choose the states q_2, q_3, \dots, q_{n+1} so that

$$\delta(q_i, a_i) = q_{i+1}, \quad i = 1, 2, \dots, n, \quad (5.2)$$

and $q_{n+1} \in F$. In this case, not only does (5.1) hold, but also the productions

$$a_i^{q_i q_{i+1}} \rightarrow a_i, \quad i = 1, 2, \dots, n, \quad (5.3)$$

all belong to $\tilde{\Gamma}$. Hence, finally,

$$\tilde{S} \xrightarrow[\tilde{\Gamma}]{} a_1 a_2 \cdots a_n.$$

Conversely, let $\tilde{S} \xrightarrow[\tilde{\Gamma}]{} u \in A^*$. We shall need the following:

Lemma. Let $\sigma^{pq} \xrightarrow[\tilde{\Gamma}]{} u \in A^*$. Then, $\delta^*(p, u) = q$. Moreover, if σ is a variable, then $\sigma \xrightarrow[\tilde{\Gamma}]{} u$.

Since $\tilde{S} \xrightarrow[\tilde{\Gamma}]{} S^{q_1 q} \xrightarrow[\tilde{\Gamma}]{} u$ where $q \in F$, we can use the Lemma to conclude that $\delta^*(q_1, u) = q$, and $S \xrightarrow[\tilde{\Gamma}]{} u$. Hence, $u \in R \cap L(\Gamma)$. Theorem 5.4 then follows immediately. ■

It remains to prove the Lemma.

Proof of Lemma. The proof is by induction on the length of a derivation of u from σ^{pq} in $\tilde{\Gamma}$. If that length is 2, we must have $\sigma \in A$, $u = \sigma$. Then, $\delta^*(p, u) = \delta(p, u) = q$. Otherwise we can write

$$\sigma^{pq} \xrightarrow[\tilde{\Gamma}]{} \sigma_1^{r_0 r_1} \sigma_2^{r_1 r_2} \cdots \sigma_n^{r_{n-1} r_n} \xrightarrow[\tilde{\Gamma}]{} u$$

where we have written $r_0 = p$ and $r_n = q$. Thus, we have

$$\sigma_i^{r_{i-1} r_i} \xrightarrow[\tilde{\Gamma}]{} u_i, \quad i = 1, 2, \dots, n, \quad (5.4)$$

where $u = u_1 u_2 \cdots u_n$. Clearly, the induction hypothesis can be applied to the derivations in (5.4) so that $\delta^*(r_{i-1}, u_i) = r_i$, $i = 1, 2, \dots, n$. Hence $\delta^*(p, u) = r_n = q$. Also, if σ_i is a variable, the induction hypothesis will give $\sigma_i \xrightarrow[\tilde{\Gamma}]{} u_i$, while otherwise $\sigma_i \in A$ and $\sigma_i = u_i$. Finally,

$$\sigma \rightarrow \sigma_1 \sigma_2 \cdots \sigma_n$$

must be a production of Γ . Hence, we have:

$$\begin{aligned} \sigma &\xrightarrow[\tilde{\Gamma}]{} \sigma_1 \sigma_2 \cdots \sigma_n \\ &\xrightarrow[\tilde{\Gamma}]{} u_1 u_2 \cdots u_n = u. \end{aligned}$$
■

Let A, P be alphabets such that $P \subseteq A$. For each letter $a \in A$, let us write

$$a^0 = \begin{cases} 0 & \text{if } a \in P \\ a & \text{if } a \in A - P. \end{cases}$$

If $x = a_1 a_2 \cdots a_n \in A^*$, we write

$$\text{Er}_P(x) = a_1^0 a_2^0 \cdots a_n^0.$$

In other words, $\text{Er}_P(x)$ is the word which results from x when all the symbols in it that are part of the alphabet P are “erased.” If $L \subseteq A^*$, we also write

$$\text{Er}_P(L) = \{\text{Er}_P(x) \mid x \in L\}.$$

Finally, if Γ is any context-free grammar with terminals T and if $P \subseteq T$, we write $\text{Er}_P(\Gamma)$ for the context-free grammar with terminals $T - P$, the same variables and start symbol as Γ , and productions

$$X \rightarrow \text{Er}_P(v)$$

for each production $X \rightarrow v$ of Γ . [Note that even if Γ is a positive context-free grammar, $\text{Er}_P(\Gamma)$ may not be positive; that is, it is possible that $\text{Er}_P(v) = 0$ even if $v \neq 0$.] We have

Theorem 5.5. If Γ is a context-free grammar and $\tilde{\Gamma} = \text{Er}_P(\Gamma)$, then $L(\tilde{\Gamma}) = \text{Er}_P(L(\Gamma))$.³

³ Readers familiar with the terminology may enjoy noting that this theorem states that the “operators” L and Er_P commute.

Proof. Let S be the start symbol of Γ and $\tilde{\Gamma}$. Suppose that $w \in L(\Gamma)$. We have

$$S = w_1 \xrightarrow{\Gamma} w_2 \cdots \xrightarrow{\Gamma} w_m = w.$$

Let $v_i = \text{Er}_P(w_i)$, $i = 1, 2, \dots, m$. Then clearly,

$$S = v_1 \xrightarrow{\tilde{\Gamma}} v_2 \cdots \xrightarrow{\tilde{\Gamma}} v_m = \text{Er}_P(w),$$

so that $\text{Er}_P(w) \in L(\tilde{\Gamma})$. This proves that $L(\tilde{\Gamma}) \supseteq \text{Er}_P(L(\Gamma))$.

To complete the proof it will suffice to show that whenever $X \xrightarrow{\tilde{\Gamma}} v \in (T - P)^*$, there is a word $w \in T^*$ such that $X \xrightarrow{\tilde{\Gamma}} w$ and $v = \text{Er}_P(w)$. We do this by induction on the length of a derivation of v from X in $\tilde{\Gamma}$. If $X \xrightarrow{\tilde{\Gamma}} v$, then $X \rightarrow v$ is a production of $\tilde{\Gamma}$, so that $X \rightarrow w$ is a production of Γ for some w with $\text{Er}_P(w) = v$. Proceeding inductively, let there be a derivation of v from X in $\tilde{\Gamma}$ of length $k > 2$, where the result is known for all derivations of length $< k$. Then, we can write

$$X \xrightarrow{\tilde{\Gamma}} u_0 V_1 u_1 V_2 u_2 \cdots V_s u_s \xrightarrow{\tilde{\Gamma}} v,$$

where $u_0, u_1, \dots, u_s \in (T - P)^*$ and V_1, V_2, \dots, V_s are variables. Thus, there are words $\bar{u}_0, \bar{u}_1, \dots, \bar{u}_s \in T^*$ such that $u_i = \text{Er}_P(\bar{u}_i)$, $i = 0, 1, \dots, s$, and

$$X \rightarrow \bar{u}_0 V_1 \bar{u}_1 V_2 \bar{u}_2 \cdots V_s \bar{u}_s$$

is a production of Γ . Also we can write

$$v = u_0 v_1 u_1 v_2 u_2 \cdots v_s u_s,$$

where

$$V_i \xrightarrow{\tilde{\Gamma}} v_i, \quad i = 1, \dots, s. \quad (5.5)$$

Since (5.5) clearly involves derivations of length $< k$, the induction hypothesis applies, and we can conclude that there are words $\bar{v}_i \in T^*$, $i = 1, 2, \dots, s$, such that $v_i = \text{Er}_P(\bar{v}_i)$ and $V_i \xrightarrow{\tilde{\Gamma}} \bar{v}_i$, $i = 1, 2, \dots, s$. Hence, we have

$$X \xrightarrow{\tilde{\Gamma}} \bar{u}_0 V_1 \bar{u}_1 V_2 \bar{u}_2 \cdots V_s \bar{u}_s \xrightarrow{\tilde{\Gamma}} \bar{u}_0 \bar{v}_1 \bar{u}_1 \bar{v}_2 \bar{u}_2 \cdots \bar{v}_s \bar{u}_s.$$

But

$$\text{Er}_P(\bar{u}_0 \bar{v}_1 \bar{u}_1 \bar{v}_2 \cdots \bar{v}_s \bar{u}_s) = u_0 v_1 u_1 v_2 u_2 \cdots v_s u_s = v,$$

which completes the proof. ■

Corollary 5.6. If $L \subseteq A^*$ is a context-free language and $P \subseteq A$, then $\text{Er}_P(L)$ is also a context-free language.

Proof. Let $L = L(\Gamma)$, where Γ is a context-free grammar, and let $\tilde{\Gamma} = \text{Er}_P(\Gamma)$. Then, by Theorem 5.5, $\text{Er}_P(L) = L(\tilde{\Gamma})$, so that $\text{Er}_P(L)$ is context-free. ■

*6. Solvable and Unsolvable Problems⁴

Let Γ be a context-free grammar with terminals T and start symbol S , let $u \in T^*$, and let us consider the problem of determining whether $u \in L(\Gamma)$. First let $u = 0$. Then we can use the algorithms provided in Section 1 to compute $\ker(\Gamma)$. Since $0 \in L(\Gamma)$ if and only if $S \in \ker(\Gamma)$, we can answer the question in this case. For $u \neq 0$, we use Theorems 1.2 and 3.1 to obtain a Chomsky normal form grammar Δ such that $u \in L(\Gamma)$ if and only if $u \in L(\Delta)$. To test whether $u \in L(\Delta)$, we use the following:

Lemma. Let Δ be a Chomsky normal form grammar with terminals T . Let V be a variable of Δ and let

$$V \xrightarrow[\Delta]{} u \in T^*.$$

Then there is a derivation of u from V in Δ of length $2|u|$.

Proof. The proof is by induction on $|u|$. If $|u| = 1$, then u is a terminal and Δ must contain a production $V \rightarrow u$, so that we have a derivation of u from V of length 2.

Now, let $V \xrightarrow[\Delta]{} u$, where $|u| > 1$, and let us assume the result known for all strings of length $< |u|$. Recalling the definition of a Chomsky normal form grammar, we see that

$$V \Rightarrow XY \xrightarrow[\Delta]{} u.$$

Thus, we must have $X \xrightarrow[\Delta]{} v$, $Y \xrightarrow[\Delta]{} w$, $u = vw$, where $|v|, |w| < |u|$. By the induction hypothesis we have derivations

$$X = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{2|v|} = v,$$

$$Y = \beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow \beta_{2|w|} = w.$$

Hence, we can write the derivation

$$V \Rightarrow XY = \alpha_1 Y \Rightarrow \alpha_2 Y \Rightarrow \cdots \Rightarrow \alpha_{2|v|} Y = v\beta_1 \Rightarrow v\beta_2 \Rightarrow \cdots \Rightarrow v\beta_{2|w|},$$

where $v\beta_{2|w|} = vw = u$. But this derivation is of length $2|v| + 2|w| = 2|u|$, which completes the proof. ■

Now to test $u \in L(\Delta)$, we simply write out all derivations from S of length $2|u|$. We have $u \in L(\Delta)$ if and only if at least one of these derivations terminates in the string u .

We have proved

⁴ The * does not refer to the material through Theorem 6.4.

Theorem 6.1.⁵ There is an algorithm that will test a given context-free grammar Γ and a given word u to determine whether $u \in L(\Gamma)$.

Next we wish to consider the question of whether a given context-free grammar generates the empty language \emptyset . Let Γ be a given context-free grammar. We first check as above to decide whether $0 \in L(\Gamma)$. If $0 \in L(\Gamma)$, we know that $L(\Gamma) \neq \emptyset$. Otherwise we use Theorems 1.2 and 1.5 to obtain a *branching* context-free grammar $\bar{\Gamma}$ such that $L(\Gamma) = L(\bar{\Gamma})$. Let $\bar{\Gamma}$ have n variables and set of terminals T . Suppose that $L(\bar{\Gamma}) \neq \emptyset$. Let $u \in L(\bar{\Gamma})$, where u has the shortest possible length of any word in $L(\bar{\Gamma})$. Then *in any derivation tree for u in $\bar{\Gamma}$, each path contains fewer than $n + 2$ nodes*. This is because if there were a path containing at least $n + 2$ nodes, at least $n + 1$ of them would be labeled by variables, and by the pigeon-hole principle, Theorem 1.6 would apply and yield a word $v \in L(\bar{\Gamma})$ with $|v| < |u|$. Thus, we conclude that

$L(\bar{\Gamma}) \neq \emptyset$ if and only if there is a derivation tree \mathcal{T} in Γ of a word $u \in T^*$ such that each path in \mathcal{T} contains fewer than $n + 2$ nodes.

It is a straightforward matter (at least in principle) to write out explicitly all derivation trees in Γ in which no path has length $\geq n + 2$. To test whether $L(\bar{\Gamma}) \neq \emptyset$, it suffices to note whether there is such a tree \mathcal{T} for which $\langle \mathcal{T} \rangle \in T^*$. Thus we have

Theorem 6.2. There is an algorithm to test a given context-free grammar Γ to determine whether $L(\Gamma) = \emptyset$.

Next we seek an algorithm to test whether $L(\Gamma)$ is finite or infinite for a given context-free grammar Γ . Such an algorithm can easily be obtained from the following:

Theorem 6.3. Let Γ be a Chomsky normal form grammar with exactly n variables. Then $L(\Gamma)$ is infinite if and only if there is a word $x \in L(\Gamma)$ such that

$$2^n < |x| \leq 2^{n+1}.$$

Proof. If there is a word $x \in L(\Gamma)$ with $|x| > 2^n$, then by Bar-Hillel's pumping lemma, $L(\Gamma)$ is infinite.

Conversely, let $L(\Gamma)$ be infinite. Let u be a word of shortest possible length such that $u \in L(\Gamma)$ and $|u| > 2^{n+1}$. By Bar-Hillel's pumping lemma,

⁵ This result follows at once from Theorem 5.5 in Chapter 7; but the algorithm given here is of some independent interest.

we have $u = r_1 q_1 r q_2 r_2$ where $q_1 q_2 \neq 0$, $|q_1 r q_2| \leq 2^n$ and $x = r_1 r r_2 \in L(\Gamma)$. Now,

$$|x| \geq |r_1 r_2| = |u| - |q_1 r q_2| > 2^n.$$

Since $|x| < |u|$, the manner in which we chose u guarantees that $|x| \leq 2^{n+1}$. ■

Theorem 6.4. There is an algorithm to test a given context-free grammar Γ to determine whether $L(\Gamma)$ is finite or infinite.

Proof. Given context-free grammar Γ with terminals T , we use the algorithms of Theorems 1.2 and 3.1 to construct a Chomsky normal form grammar Δ with $L(\Gamma) = L(\Delta)$ or $L(\Delta) \cup \{0\}$. Let Δ have n variables and let $l = 2^n$. Then we simply use Theorem 6.1 to test each word $u \in T^*$ for which $l < |u| \leq 2l$ to see whether $u \in L(\Gamma)$. $L(\Gamma)$ is infinite if and only if at least one of these words u does belong to $L(\Gamma)$. ■

Remarkably enough, there are also some very simple unsolvable problems related to context-free grammars.⁶ The easiest way to obtain these results is to associate a pair of context-free grammars with each Post correspondence system.

Thus, suppose we are given the finite set of dominoes:



$i = 1, 2, \dots, n$, where $u_i, v_i \in A^*$ for some given alphabet A . We introduce n new symbols c_1, c_2, \dots, c_n and define two context-free grammars Γ_1, Γ_2 , both of which have as their terminals $A \cup \{c_1, c_2, \dots, c_n\}$. Γ_1 has the single variable S_1 , its start symbol, and Γ_2 has S_2 as its only variable and start symbol. The productions of Γ_1 are

$$S_1 \rightarrow u_i S_1 c_i, \quad S_1 \rightarrow u_i c_i, \quad i = 1, 2, \dots, n,$$

and those of Γ_2 are

$$S_2 \rightarrow v_i S_2 c_i, \quad S_2 \rightarrow v_i c_i, \quad i = 1, 2, \dots, n.$$

Now, the given Post correspondence system has a solution if and only if we can have

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m} \quad .$$

⁶ The remainder of this section depends on Chapter 7. Readers who have not covered this material should move on to Section 7.

for some sequence i_1, i_2, \dots, i_m . Moreover,

$$L(\Gamma_1) = \{u_{i_1}u_{i_2}\cdots u_{i_m}c_{i_m}\cdots c_{i_2}c_{i_1}\}$$

and

$$L(\Gamma_2) = \{v_{i_1}v_{i_2}\cdots v_{i_m}c_{i_m}\cdots c_{i_2}c_{i_1}\}.$$

Thus, we have

Theorem 6.5. $L(\Gamma_1) \cap L(\Gamma_2) \neq \emptyset$ if and only if the given Post correspondence problem has a solution.

Using Theorem 4.1 in Chapter 7, we conclude

Theorem 6.6. There is no algorithm to test a given pair of context-free grammars Γ_1, Γ_2 to determine whether $L(\Gamma_1) \cap L(\Gamma_2) = \emptyset$.

Another important unsolvability result about context-free grammars concerns ambiguity.

Definition. A context-free grammar Γ is called *ambiguous* if there is a word $u \in L(\Gamma)$ which has two different leftmost derivations in Γ . If Γ is not ambiguous, it is said to be *unambiguous*.

Theorem 6.7. There is no algorithm to test a given context-free grammar to determine whether it is ambiguous.

Proof. Once again we begin with a Post correspondence system, and form the two context-free grammars Γ_1, Γ_2 used in proving Theorem 6.5. Γ_1 and Γ_2 are obviously both unambiguous. Now let Γ have start symbol S and all of the productions of Γ_1 and Γ_2 , together with $S \rightarrow S_1$ and $S \rightarrow S_2$. Then, since the first step of a derivation from S in Γ involves an irreversible commitment to either Γ_1 or Γ_2 , Γ will be ambiguous just in case $L(\Gamma_1) \cap L(\Gamma_2) \neq \emptyset$. By Theorem 6.5 this will be the case if and only if the given Post correspondence system has a solution. The result now follows again from Theorem 4.1 in Chapter 7. ■

Exercises

1. Write a context-free grammar Γ such that

$$L(\Gamma) = \{a^i b^j c^k \mid i = j \vee j = k\}.$$

This language is an example of an *inherently ambiguous language*, i.e., a language such that every grammar which generates it is ambiguous. Explain why this language is inherently ambiguous.

2. Give an unambiguous context-free grammar which generates the same language as the ambiguous grammar

$$\begin{aligned} S &\rightarrow aB \\ S &\rightarrow Ab \\ A &\rightarrow aAB \\ B &\rightarrow ABb \\ A &\rightarrow a \\ B &\rightarrow b. \end{aligned}$$

7. Bracket Languages

Let A be some finite set. Although we think of A as an alphabet, we will also wish to permit $A = \emptyset$. Let B be the alphabet we get from A by adjoining the $2n$ new symbols $\{\}_i, \}_{i}, i = 1, 2, \dots, n$, where n is some given positive integer. We will write $\text{PAR}_n(A)$ for the language consisting of all strings in B^* which are correctly “paired,” thinking of each pair $\{\}_{i}, \}_{i}$ as matching left and right brackets. More precisely, $\text{PAR}_n(A) = L(\Gamma_0)$, where Γ_0 is the context-free grammar with the single variable S , terminals B , and the productions

- (1) $S \rightarrow a$ for all $a \in A$,
- (2) $S \rightarrow \{S\}_i, i = 1, 2, \dots, n$,
- (3) $S \rightarrow SS, S \rightarrow 0$.

The languages $\text{PAR}_n(A)$ are called *bracket languages*.

Let us consider the example $A = \{a, b, c\}$, $n = 2$. For ease of reading we will use the symbol “(” for “ $\{_1$ ”, “)” for “ $\}_{1}$ ”, “[” for “ $\{_{2}$ ”, and “]” for “ $\}_{2}$ ”. Then $cb[(ab)c](a[b]c) \in \text{PAR}_2(A)$, as the reader should easily verify. Also, $()[] \in \text{PAR}_2(A)$, since we have

$$S \Rightarrow SS \Rightarrow (S)[S] \Rightarrow ()[].$$

Bracket languages have the following properties:

Theorem 7.1. $\text{PAR}_n(A)$ is a context-free language such that

- (a) $A^* \subseteq \text{PAR}_n(A)$;
- (b) if $x, y \in \text{PAR}_n(A)$, so is xy ;
- (c) if $x \in \text{PAR}_n(A)$, so is $\{x\}_i$, for $i = 1, 2, \dots, n$;
- (d) if $x \in \text{PAR}_n(A)$ and $x \notin A^*$, then we can write $x = u \{v\}_i w$, for some $i = 1, 2, \dots, n$, where $u \in A^*$ and $v, w \in \text{PAR}_n(A)$.

Proof. Since $\text{PAR}_n(A) = L(\Gamma_0)$ where Γ_0 is a context-free grammar, $\text{PAR}_n(A)$ must be context-free. (a) follows at once on considering the productions (1) and (3). For (b), let $S \Rightarrow x, S \Rightarrow y$. Then using the productions (3), we have

$$S \Rightarrow SS \Rightarrow xy.$$

For (c), let $S \Rightarrow x$. Then using the productions (2), we have

$$S \Rightarrow \{S_i\} \Rightarrow \{x_i\}.$$

To prove (d), note first that we can assume $|x| > 1$ because otherwise $x \in A^*$. Then, a derivation of x from S must begin by using a production containing S on the right. We proceed by induction assuming the result for all strings of length $< |x|$. There are two cases:

Case 1. $S \Rightarrow \{S_i\} \Rightarrow \{v_i\} = x$, where $S \Rightarrow v$; the result then follows (without using the induction hypothesis) with $u = w = 0$.

Case 2. $S \Rightarrow SS \Rightarrow rs = x$ where $S \Rightarrow r, S \Rightarrow s$, and $r \neq 0, s \neq 0$. Clearly, $|r|, |s| < |x|$. If $r \in A^*$, then $|s| > 1$ and we can use the induction hypothesis to write $s = u\{v\}w$, where $u \in A^*$ and $v, w \in \text{PAR}_n(A)$, and the desired result follows since $ru \in A^*$. Otherwise, we can use the induction hypothesis to write $r = u\{v\}w$ where $u \in A^*$ and $v, w \in \text{PAR}_n(A)$, so that the result follows since $ws \in \text{PAR}_n(A)$ by (b). ■

Historically, the special case $A = \emptyset$ has played an important role in studying context-free languages. The language $\text{PAR}_n(\emptyset)$ is called the Dyck language of order n and is usually written D_n .

Now let us begin with a Chomsky normal form grammar Γ , with terminals T and productions

$$X_i \rightarrow Y_i Z_i, \quad i = 1, 2, \dots, n, \quad (7.1)$$

in addition to certain productions of the form $V \rightarrow a$ with $a \in T$. We will construct a new grammar Γ_s which we call the *separator* of Γ . The terminals of Γ_s are the symbols of T together with $2n$ new symbols $\{, \}, i = 1, 2, \dots, n$. Thus a pair of “brackets” has been added for each of the productions (7.1).

The productions of Γ_s are

$$X_i \rightarrow \{Y_i\}Z_i, \quad i = 1, 2, \dots, n,$$

as well as all of the productions of Γ of the form $V \rightarrow a$ with $a \in T$.

As an example, let Γ have the productions

$$S \rightarrow XY, \quad S \rightarrow YX, \quad Y \rightarrow ZZ,$$

$$X \rightarrow a, \quad Z \rightarrow a.$$

Then Γ is ambiguous as we can see from the leftmost derivations:

$$S \Rightarrow XY \Rightarrow aY \Rightarrow aaZZ \Rightarrow aaZ \Rightarrow aaa,$$

$$S \Rightarrow YX \Rightarrow ZZX \Rightarrow aZX \Rightarrow aaX \Rightarrow aaa.$$

The productions of Γ_s can be written

$$\begin{aligned} S \rightarrow (X)Y, \quad S \rightarrow [Y]X, \quad Y \rightarrow \{Z\}Z, \\ X \rightarrow a, \quad Z \rightarrow a, \end{aligned}$$

using $()$, $[]$, and $\{ \}$ in place of the numbered brackets. The two derivations just given then become

$$\begin{aligned} S \Rightarrow (X)Y \Rightarrow (a)Y \Rightarrow (a)\{Z\}Z \Rightarrow (a)\{a\}Z \Rightarrow (a)\{a\}a, \\ S \Rightarrow [Y]X \Rightarrow [\{Z\}Z]X \Rightarrow [\{a\}Z]X \Rightarrow [\{a\}a]X \Rightarrow [\{a\}a]a. \end{aligned}$$

Γ_s thus *separates* the two derivations in Γ . The bracketing in the words $(a)\{a\}a$, $[\{a\}a]a$ enables their respective derivation trees to be recovered.

If we write P for the set of brackets $\{ , \}$, $i = 1, 2, \dots, n$, then clearly $\Gamma = \text{Er}_P(\Gamma_s)$. Hence by Theorem 5.5,

Theorem 7.2. $\text{Er}_P(L(\Gamma_s)) = L(\Gamma)$.

We also will prove

Lemma 1. $L(\Gamma_s) \subseteq \text{PAR}_n(T)$.

Proof. We show that if $X \xrightarrow{\Gamma_s} w \in (T \cup P)^*$ for any variable X , then $w \in \text{PAR}_n(T)$. The proof is by induction on the length of a derivation of w from X in Γ_s . If this length is 2, then w is a single terminal and the result is clear. Otherwise we can write

$$X = X_i \xrightarrow{\Gamma_s} \{Y_i\}Z_i \xrightarrow{\Gamma_s} \{u\}v = w,$$

where $Y_i \xrightarrow{\Gamma_s} u$ and $Z_i \xrightarrow{\Gamma_s} v$. By the induction hypothesis, $u, v \in \text{PAR}_n(T)$. By (b) and (c) of Theorem 7.1, so is w . ■

Now let Δ be the grammar whose variables, start symbol, and terminals are those of Γ_s and whose productions are as follows:

- (1) all productions $V \rightarrow a$ from Γ (or equivalently Γ_s) with $a \in T$,
- (2) all productions $X_i \rightarrow \{Y_i\}$, $i = 1, 2, \dots, n$,
- (3) all productions $V \rightarrow a_i^j Z_i$, $i = 1, 2, \dots, n$, for which $V \rightarrow a$ is a production of Γ with $a \in T$.

We have

Lemma 2. $L(\Delta)$ is regular.

Proof. Since Δ is obviously right-linear, the result follows at once from Theorem 2.5. ■

Lemma 3. $L(\Gamma_s) \subseteq L(\Delta)$.

Proof. We show that if $X \xrightarrow{\Gamma_s} u \in (T \cup P)^*$ then $X \xrightarrow{\Delta} u$. If u has a derivation of length 2, then $u \in T$, and $X \rightarrow u$ is a production of Γ_s and of Γ and therefore also of Δ . Thus $X \xrightarrow{\Delta} u$.

Proceeding by induction, let

$$X = X_i \xrightarrow{\Gamma_s} \{Y_i\} Z_i \xrightarrow{\Gamma_s} \{v_i\} w = u,$$

where the induction hypothesis applies to $Y_i \xrightarrow{\Gamma_s} v$ and to $Z_i \xrightarrow{\Gamma_s} w$. Thus, $Y_i \xrightarrow{\Delta} v$ and $Z_i \xrightarrow{\Delta} w$. Let $v = za$, $a \in T$. Then, examining the productions of the grammar Δ , we see that we must have

$$Y_i \xrightarrow{\Delta} zV \xrightarrow{\Delta} za = v,$$

where $V \rightarrow a$ is a production of Γ . But then we have

$$X_i \xrightarrow{\Delta} \{Y_i \xrightarrow{\Delta} \{zV \xrightarrow{\Delta} \{za\} Z_i \xrightarrow{\Delta} \{v_i\} w = u.\}$$
■

Lemma 4. $L(\Delta) \cap \text{PAR}_n(T) \subseteq L(\Gamma_s)$.

Proof. Let $X \xrightarrow{\Delta} u$, where $u \in \text{PAR}_n(T)$. We shall prove that $X \xrightarrow{\Gamma_s} u$. The proof is by induction on the total number of occurrences of the symbols $\{, \}$ in u . If this number is 0, then, examining the productions of Δ , we see that $u \in T$ and the production $X \rightarrow u$ is in Δ and hence in Γ_s . Thus $X \xrightarrow{\Gamma_s} u$.

Now let $X \xrightarrow{\Delta} u$, where u contains occurrences of the bracket symbols $\{, \}$ and where the result is known for words v containing fewer occurrences of these symbols than u . Examining the productions of Δ , we see that our derivation of u from X must begin with one of the productions (2). [If the derivation began with a production of the form (1), then u would be a terminal. If the derivation began with a production of the form (3), then $u = a\}w$ for some word w , which is impossible by Theorem 7.1d.] Therefore $u = \{z$, for some word z and some $i = 1, 2, \dots, n$. By Theorem 7.1d, $u = \{v_i\}w$, where $v, w \in \text{PAR}_n(T)$. In our derivation of u in Δ , the symbol $\{$ can only arise from the use of one of the productions of the form (3), say, $V \rightarrow a_i^j Z_i$, where $a \in T$ and $V \rightarrow a$ is a production of Γ . Then v must end in a , so that we can write $v = \bar{v}a$, where

$$X = X_i \Rightarrow \{Y_i \xrightarrow{*} \bar{v}V \Rightarrow \{\bar{v}a_i^j Z_i \xrightarrow{*} \{v_i\}w$$

and $Z_i \xrightarrow{\Delta} w$. Moreover, since $V \rightarrow a$ is a production of Γ , it is also one of the productions of Δ of the form (1). Therefore, we have in Δ

$$Y_i \xrightarrow{*} \bar{v}V \Rightarrow \bar{v}a = v.$$

Since v and w must each contain fewer occurrences of $\{, \}$ than u , we have by the induction hypothesis

$$Y_i \xrightarrow{\Gamma_s} v, \quad Z_i \xrightarrow{\Gamma_s} w.$$

Hence,

$$X_i \xrightarrow[\Gamma_s]{} \{Y_{ii}\} Z_i \xrightarrow[\Gamma_s]{} \{v_i\} w = u.$$
■

We are now ready to state

Theorem 7.3. Let Γ be a grammar in Chomsky normal form with terminals T . Then there is a regular language R such that

$$L(\Gamma_s) = R \cap \text{PAR}_n(T).$$

Proof. Let Δ be defined as above and let $R = L(\Delta)$. The result then follows at once from Lemmas 1–4.

Theorem 7.4 (Chomsky–Schützenberger Representation Theorem). A language $L \subseteq T^*$ is context-free if and only if there is a regular language R and a number n such that

$$L = \text{Er}_P(R \cap \text{PAR}_n(T)), \quad (7.2)$$

where $P = \{\langle , \rangle | i = 1, 2, \dots, n\}$.

Proof. It is clear by Theorems 7.2 and 7.3 that for every grammar Γ in Chomsky normal form, $L = L(\Gamma)$ satisfies (7.2). For an arbitrary context-free language L , by Theorems 1.2 and 3.1, there is a Chomsky normal form grammar Γ such that

$$L = L(\Gamma) \text{ or } L = L(\Gamma) \cup \{0\}.$$

If

$$L(\Gamma) = \text{Er}_P(R \cap \text{PAR}_n(T)),$$

then

$$L(\Gamma) \cup \{0\} = \text{Er}_P((R \cup \{0\}) \cap \text{PAR}_n(T))$$

since, by Theorem 7.1a, $0 \in \text{PAR}_n(T)$. But, by Theorems 4.2 and 4.5 in Chapter 8, $R \cup \{0\}$ is a regular language.

It remains only to show that any language L that satisfies (7.2) must be context-free. But since, by Theorem 7.1, $\text{PAR}_n(T)$ is context-free, this result follows at once from Theorem 5.4 and Corollary 5.6. ■

The Chomsky–Schützenberger theorem is usually expressed in terms of the Dyck languages $D_n = \text{PAR}_n(\emptyset)$. Since our form of the theorem is equivalent to the more usual form, we will give only a very brief sketch of the proof of the usual form. It is necessary to go back to the construction of Γ_s . Each element a of T is now thought of as a “left bracket” and is supplied with a “twin” a' to act as its corresponding right bracket. A new grammar Γ_t is then defined to have the same productions $X_i \rightarrow \{Y_{ii}\}$, $i = 1, 2, \dots, n$, as Γ_s but to have productions

$$V \rightarrow aa'$$

for each production $V \rightarrow a$ of Γ . Then clearly, $L(\Gamma_s)$ can be obtained from $L(\Gamma_s)$ by simply replacing all occurrences of letters $a \in T$ in words of $L(\Gamma_s)$ by aa' . By replacing a by aa' in productions of the forms (1) and (3) of Δ , we obtain a right linear grammar Δ' such that

$$L(\Gamma_s) = L(\Delta') \cap \text{PAR}_n(T'),$$

where $T' = \{a, a' \mid a \in T\}$. But in fact $L(\Gamma_s) \subseteq D_m$, where $m = n + k$ and there are k letters in T . Thus,

$$L(\Gamma_s) = L(\Delta') \cap D_m.$$

Finally letting $Q = \{\langle i, \rangle \mid i = 1, 2, \dots, n\} \cup \{a' \mid a \in T\}$, we have

$$L(\Gamma_s) = \text{Er}_Q(L(\Gamma_s)) = \text{Er}_Q(L(\Delta') \cap D_m).$$

Thus, we get

Theorem 7.5. A language L is context-free if and only if there is a regular language R , an alphabet Q , and an integer m such that

$$L = \text{Er}_Q(R \cap D_m).$$

8. Pushdown Automata

We are now ready to discuss the question of what kind of automaton is needed for accepting context-free languages? We take our cue from Theorem 7.2, and begin by trying to construct an appropriate automaton for recognizing $L(\Gamma_s)$, where Γ is a given Chomsky normal form grammar. We know that $L(\Gamma_s) = R \cap \text{PAR}_n(T)$, where R is a regular language. Thus R is accepted by a finite automaton. The problem we need to solve is this: what additional facilities does this finite automaton require in order to check that some given word belongs to $\text{PAR}_n(T)$? Those familiar with “stacks” and their uses will see at once that what is needed is a “pushdown stack” as an auxiliary storage device. Such a device behaves in a last-in-first-out manner. At each step in a computation with a pushdown stack one or both of a pair of operations can be performed:

1. The symbol at the “top” of the stack may be read and discarded. (This operation is called *popping the stack*.)
2. A new symbol may be “pushed” onto the stack.

A stack can be used to identify a string as belonging to $\text{PAR}_n(T)$ as follows: For each pair $\langle i, \rangle$, $i = 1, 2, \dots, n$, a special symbol J_i is introduced. Now, as our automaton moves from left to right over a string, it pushes J_i onto

the stack whenever it sees $\{$ and it pops the stack eliminating a J_i whenever it sees $\}$. Such an automaton will successfully scan the entire string and terminate with an empty stack just in case the string belongs to $\text{PAR}_n(T)$.

To move toward making these ideas precise, let T be a given alphabet and let $P = \{\langle , \rangle \mid i = 1, 2, \dots, n\}$. Let $\Omega = \{J_1, J_2, \dots, J_n\}$, where we have introduced a single symbol J_i for each pair \langle , \rangle , $1 \leq i \leq n$. Let $u \in (T \cup P)^*$, say, $u = c_1 c_2 \dots c_k$, where $c_1, c_2, \dots, c_k \in T \cup P$. We define a sequence $\gamma_j(u)$ of elements of Ω^* as follows:

$$\begin{aligned}\gamma_1(u) &= 0 \\ \gamma_{j+1}(u) &= \begin{cases} \gamma_j(u) & \text{if } c_j \in T \\ J_i \gamma_j(u) & \text{if } c_j = \langle \\ \alpha & \text{if } c_j = \rangle \text{ and } \gamma_j(u) = J_i \alpha \end{cases}\end{aligned}$$

for $j = 1, 2, \dots, k$. Note that if $c_j = \langle$, $\gamma_{j+1}(u)$ will be *undefined unless* γ_j begins with the symbol J_i for the very same value of i . Of course, if a particular $\gamma_r(u)$ is undefined, all $\gamma_j(u)$ with $j > r$ will also be undefined.

Definition. We say that the word $u \in (T \cup P)^*$ is *balanced* if $\gamma_j(u)$ is defined for $1 \leq j \leq |u| + 1$ and $\gamma_{|u|+1}(u) = 0$.

The heuristic considerations above suggest

Theorem 8.1. Let T be an alphabet and let

$$P = \{\langle , \rangle \mid i = 1, 2, \dots, n\}, \quad T \cap P = \emptyset.$$

Let $u \in (T \cup P)^*$, let $\Omega = \{J_1, J_2, \dots, J_n\}$. Then $u \in \text{PAR}_n(T)$ if and only if u is balanced.

The proof is via a series of easy lemmas.

Lemma 1. If $u \in T^*$, then u is balanced.

Proof. Clearly $\gamma_j(u) = 0$ for $1 \leq j \leq |u| + 1$ in this case. ■

Lemma 2. If u and v are balanced, so is uv .

Proof. Clearly $\gamma_j(uv) = \gamma_j(u)$ for $1 \leq j \leq |u| + 1$. Since $\gamma_{|u|+1}(u) = 0 = \gamma_{|u|+1}(uv) = \gamma_1(v)$, we have $\gamma_{|u|+1}(uv) = \gamma_1(v) = \gamma_j(v)$ for $1 \leq j \leq |v| + 1$. Hence, $\gamma_{|uv|+1}(uv) = \gamma_{|u|+|v|+1}(uv) = \gamma_{|v|+1}(v) = 0$. ■

Lemma 3. Let $v = \langle u_i \rangle$. Then u is balanced if and only if v is balanced.

Proof. We have $\gamma_1(v) = 0$, $\gamma_2(v) = J_i$, $\gamma_{j+1}(v) = \gamma_j(u) J_i$, $j = 1, 2, \dots, |v| - 1$. In particular, $\gamma_{|v|}(v) = \gamma_{|u|+2}(v) = \gamma_{|u|+1}(u) J_i$. Thus, if u is balanced, then $\gamma_{|u|+1}(u) = 0$, so that $\gamma_{|v|}(v) = J_i$ and $\gamma_{|v|+1}(v) = 0$. Conversely, if v is balanced, $\gamma_{|v|+1}(v) = 0$, so that $\gamma_{|v|}(v)$ must be J_i and $\gamma_{|u|+1}(u) = 0$. ■

Lemma 4. If u is balanced and uv is balanced, then v is balanced.

Proof. $\gamma_j(uv) = \gamma_j(u)$ for $1 \leq j \leq |u| + 1$. Since $\gamma_{|u|+1}(u) = 0$, we have $\gamma_{|u|+j}(uv) = \gamma_j(v)$ for $1 \leq j \leq |v| + 1$. Finally,

$$0 = \gamma_{|uv|+1}(uv) = \gamma_{|u|+|v|+1}(uv) = \gamma_{|v|+1}(v). \quad \blacksquare$$

Lemma 5. If $u \in \text{PAR}_n(T)$, then u is balanced.

Proof. The proof is by induction on the total number of occurrences of the symbols $\{\}, \}$ in u . If this number is 0, then $u \in T^*$, so by Lemma 1, u is balanced.

Proceeding by induction, let u have $k > 0$ occurrences of the symbols $\{\}, \}$, where the result is known for all strings with fewer than k occurrences of these symbols. Then, by Theorem 7.1d, we can write $u = v_i\{w_i\}z$, where $v, w, z \in \text{PAR}_n(T)$. By the induction hypothesis, v, w, z are all balanced, and by Lemmas 2 and 3, u is therefore balanced. \blacksquare

Lemma 6. If u is balanced, then $u \in \text{PAR}_n(T)$.

Proof. If $u \in T^*$, the result follows from Theorem 7.1a. Otherwise, we can write $u = xy$, where $x \in T^*$ and the initial symbol of y is in P . By the definition of $\gamma_j(u)$, we will have $\gamma_j(u) = 0$ for $1 \leq j \leq |x| + 1$. Therefore, the initial symbol of y cannot be one of the $\{\}$. Thus we can write $u = x\{z$, and $\gamma_{|x|+2}(u) = J_i$. Since u is balanced, $\gamma_{|u|+1}(u) = 0$, and we can let k be the least integer $> |x| + 1$ for which $\gamma_k(u) = 0$. Then $\gamma_{k-1}(u) = J_i$ and the $(k-1)$ th symbol of u must be $\}$. Thus $u = x\{v_i\}w$, where $k = |x| + |v| + 3$. Thus $0 = \gamma_{|x|+|v|+3}(u) = \gamma_{|x|+|v|+3}(x\{v_i\})$. Hence $x\{v_i\}$ is balanced. By Lemma 4, w is balanced. Since $x \in T^*$, x is balanced, and by Lemma 4 again, $\{v_i\}$ is balanced. By Lemma 3, v is balanced. Since $x \in T^*$, $x \in \text{PAR}_n(T)$. Since $|v|, |w| < |u|$, we can assume by mathematical induction that it is already known that $v, w \in \text{PAR}_n(T)$. By (b) and (c) of Theorem 7.1, we conclude that $u \in \text{PAR}_n(T)$. \blacksquare

Theorem 8.1 is an immediate consequence of Lemmas 5 and 6.

We now give a precise definition of *pushdown automata*. We begin with a finite set of *states* $Q = \{q_1, \dots, q_m\}$, q_1 being the *initial state*, a subset $F \subseteq Q$ of *accepting states*, a *tape alphabet* A , and a *pushdown alphabet* Ω . (We usually use lowercase letters for elements of A and capital letters for elements of Ω .) We assume that the symbol $\mathbf{0}$ does not belong to either A or Ω and write $\bar{A} = A \cup \{\mathbf{0}\}$, $\bar{\Omega} = \Omega \cup \{\mathbf{0}\}$. A *transition* is a quintuple of the form

$$q_i a U : V q_j$$

where $a \in \bar{A}$ and $U, V \in \bar{\Omega}$. Intuitively, if $a \in A$ and $U, V \in \Omega$, this is to read; "In state q_i scanning a , with U on top of the stack, move one square to the right,

‘pop’ the stack removing U , ‘push’ V onto the stack, and enter state q_j .” If $a = \mathbf{0}$, motion to the right does not take place and the stack action can occur regardless of what symbol is actually being scanned. Similarly, $U = \mathbf{0}$ indicates that nothing is to be popped and $V = \mathbf{0}$ that nothing is to be pushed. A *pushdown automaton* is specified by a finite set of transitions. The *distinct* transitions $q_i a U : V q_j, q_i b W : X q_k$ are called *incompatible* if one of the following is the case:

1. $a = b$ and $U = W$;
2. $a = b$ and U or W is $\mathbf{0}$;
3. $U = W$ and a or b is $\mathbf{0}$;
4. a or b is $\mathbf{0}$ and U or W is $\mathbf{0}$.

A pushdown automaton is *deterministic* if it has no pair of incompatible transitions.

Let $u \in A^*$ and let \mathcal{M} be a pushdown automaton. Then a *u -configuration for \mathcal{M}* is a triple $\Delta = (k, q_i, \alpha)$, where $1 \leq k \leq |u| + 1$, q_i is a state of \mathcal{M} , and $\alpha \in \Omega^*$. [Intuitively, the u -configuration (k, q_i, α) stands for the situation in which u is written on \mathcal{M} 's tape, \mathcal{M} is scanning the k th symbol of u —or, if $k = |u| + 1$, has completed scanning u —and α is the string of symbols on the pushdown stack.] We speak of q_i as the *state at configuration Δ* and of α as the *stack contents at configuration Δ* . If $\alpha = \mathbf{0}$, we say the *stack is empty at Δ* . For a pair of u -configurations, we write

$$u: (k, q_i, \alpha) \vdash_{\mathcal{M}} (l, q_j, \beta)$$

if \mathcal{M} contains a transition $q_i a U : V q_j$, where $\alpha = U\gamma$, $\beta = V\gamma$ for some $\gamma \in \Omega^*$, and either

1. $l = k$ and $a = \mathbf{0}$, or
2. $l = k + 1$ and the k th symbol of u is a .

Note that the equation $\alpha = U\gamma$ is to be read simply $\alpha = \gamma$ in case $U = \mathbf{0}$; likewise for $\beta = V\gamma$.

A sequence $\Delta_1, \Delta_2, \dots, \Delta_m$ of u -configurations is called a *u -computation by \mathcal{M}* if

1. $\Delta_1 = (1, q, \mathbf{0})$ for some $q \in Q$,
2. $\Delta_m = (|u| + 1, p, \gamma)$ for some $p \in Q$ and $\gamma \in \Omega^*$, and
3. $u: \Delta_i \vdash_{\mathcal{M}} \Delta_{i+1}$ for $1 \leq i < m$.

This u -computation is called *accepting* if the state at Δ_1 is the initial state q_1 , the state p at Δ_m is in F , and the stack at Δ_m is empty. We say that \mathcal{M} *accepts* the string $u \in A^*$ if there is an accepting u -computation by \mathcal{M} . We write $L(\mathcal{M})$ for the set of strings accepted by \mathcal{M} , and we call $L(\mathcal{M})$ the *language accepted by \mathcal{M}* .

Acceptance can alternatively be defined either by requiring only that the state at Δ_m is in F or only that $\gamma = 0$. It is not difficult to prove that the class of languages accepted by pushdown automata is not changed by either of these alternatives. However, we shall not concern ourselves with this matter.

A few examples should provide readers with some intuition for working with pushdown automata.

EXAMPLE \mathcal{M}_1 . Tape alphabet = $\{a, b\}$, pushdown alphabet = $\{A\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$. The transitions are

$$\begin{aligned} q_1a0 &: Aq_1 \\ q_1bA &: 0q_2 \\ q_2bA &: 0q_2. \end{aligned}$$

The reader should verify that $L(\mathcal{M}_1) = \{a^{[n]}b^{[n]} | n > 0\}$.

EXAMPLE \mathcal{M}_2 . Tape alphabet = $\{a, b, c\}$, pushdown alphabet = $\{A, B\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$. The transitions are

$$\begin{aligned} q_1a0 &: Aq_1 \\ q_1b0 &: Bq_1 \\ q_1c0 &: 0q_2 \\ q_2aA &: 0q_2 \\ q_2bB &: 0q_2. \end{aligned}$$

Here, $L(\mathcal{M}_2) = \{ucu^R | u \in \{a, b\}^*\}$.

EXAMPLE \mathcal{M}_3 . Tape alphabet = $\{a, b\}$, pushdown alphabet = $\{A, B\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$,

$$\begin{aligned} q_1a0 &: Aq_1 \\ q_1b0 &: Bq_1 \\ q_1aA &: 0q_2 \\ q_1bB &: 0q_2 \\ q_2aA &: 0q_2 \\ q_2bB &: 0q_2. \end{aligned}$$

In this case, $L(\mathcal{M}_3) = \{uu^R | u \in \{a, b\}^*, u \neq 0\}$. Note that while $\mathcal{M}_1, \mathcal{M}_2$ are deterministic, \mathcal{M}_3 is a nondeterministic pushdown automaton. Does there

exist a deterministic pushdown automaton which accepts $L(\mathcal{M}_3)$? Why not?

$L(\mathcal{M}_1)$, $L(\mathcal{M}_2)$, and $L(\mathcal{M}_3)$ are all context-free languages. We begin our investigation of the relationship between context-free languages and pushdown automata with the following theorem:

Theorem 8.2. Let Γ be a Chomsky normal form grammar with separator Γ_s . Then there is a deterministic pushdown automaton \mathcal{M} such that $L(\mathcal{M}) = L(\Gamma_s)$.

Proof. Let T be the set of terminals of Γ . By Theorem 7.3, for suitable n ,

$$L(\Gamma_s) = R \cap \text{PAR}_n(T),$$

where R is a regular language. Let $P = \{\langle , \rangle | i = 1, 2, \dots, n\}$. Let \mathcal{M}_0 be a dfa with alphabet $T \cup P$ that accepts R . Let $Q = \{q_1, \dots, q_m\}$ be the states of \mathcal{M}_0 , q_1 the initial state, $F \subseteq Q$ the accepting states, and δ the transition function. We construct a pushdown automaton \mathcal{M} with tape alphabet $T \cup P$ and the same states, initial state, and accepting states as \mathcal{M}_0 . \mathcal{M} is to have the pushdown alphabet $\Omega = \{J_1, \dots, J_n\}$. The transitions of \mathcal{M} are as follows for all $q \in Q$:

- (a) for each $a \in T$, $qa0 : 0p$, where $p = \delta(q, a)$,
- (b) for $i = 1, 2, \dots, n$, $q_i0 : J_i p_i$, where $p_i = \delta(q_i, \langle \rangle)$,
- (c) for $i = 1, 2, \dots, n$, $q_i J_i : 0\bar{p}_i$, where $\bar{p}_i = \delta(q_i, \rangle)$.

Since the second entry in these transitions is never 0 , we see that for any $u \in (T \cup P)^*$, a u -computation must be of length $|u| + 1$. It is also clear that no two of the transitions in (a)–(c) are incompatible; thus, \mathcal{M} is deterministic.

Now, let $u \in L(\Gamma_s)$, $u = c_1 c_2 \cdots c_K$, where $c_1, c_2, \dots, c_K \in (T \cup P)$. Since $u \in R$, the dfa \mathcal{M}_0 accepts u . Thus, there is a sequence $p_1, p_2, \dots, p_{K+1} \in Q$ such that $p_1 = q_1$, $p_{K+1} \in F$, and $\delta(p_i, c_i) = p_{i+1}$, $i = 1, 2, \dots, K$. Since $u \in \text{PAR}_n(T)$, by Theorem 8.1, u is balanced, so that $\gamma_j(u)$ is defined for $j = 1, 2, \dots, K + 1$ and $\gamma_{K+1}(u) = 0$. We let

$$\Delta_j = (j, p_j, \gamma_j(u)), \quad j = 1, 2, \dots, K + 1.$$

To see that the sequence $\Delta_1, \Delta_2, \dots, \Delta_{K+1}$ is an accepting u -computation by \mathcal{M} , it remains only to check that

$$u: \Delta_j \vdash_{\mathcal{M}} \Delta_{j+1}, \quad j = 1, 2, \dots, K.$$

But this is clear from the definition of $\gamma_j(u)$.

Conversely, let \mathcal{M} accept $u = c_1 c_2 \cdots c_K$. Thus, let $\Delta_1, \Delta_2, \dots, \Delta_{K+1}$ be an accepting u -computation by \mathcal{M} . Let

$$\Delta_j = (j, p_j, \gamma_j), \quad j = 1, 2, \dots, K + 1.$$

Since

$$u: \Delta_j \vdash_{\mathcal{M}} \Delta_{j+1}, \quad j = 1, 2, \dots, K,$$

and $\gamma_1 = 0$, we see that γ_j satisfies the defining recursion for $\gamma_j(u)$ and hence, $\gamma_j = \gamma_j(u)$ for $j = 1, 2, \dots, K + 1$. Since $\gamma_{K+1} = 0$, u is balanced and hence $u \in \text{PAR}_n(T)$. Finally, we have $p_1 = q_1$, $p_{K+1} \in F$, and $\delta(p_j, c_j) = p_{j+1}$. Therefore the dfa \mathcal{M}_0 accepts u , and $u \in R$. ■

We call a pushdown automaton *atomic* (whether or not it is deterministic) if all of its transitions are of one of the forms

- i. $p a 0 : 0q$,
- ii. $p 0 U : 0q$,
- iii. $p 0 0 : Vq$.

Thus, at each step in a computation an atomic pushdown automaton can read the tape and move right, or pop a symbol off the stack, or push a symbol on the stack. But, unlike pushdown automata in general, it cannot perform more than one of these actions in a single step.

Let \mathcal{M} be a given *atomic* pushdown automaton with tape alphabet T and pushdown alphabet $\Omega = \{J_1, J_2, \dots, J_n\}$. We set

$$P = \{\langle i, i \rangle \mid i = 1, 2, \dots, n\}$$

and show how to use the “brackets” to define a kind of “record” of a computation by \mathcal{M} . Let $\Delta_1, \Delta_2, \dots, \Delta_m$ be a (not necessarily accepting) v -computation by \mathcal{M} , where $v = c_1 c_2 \cdots c_K$ and $c_k \in T$, $k = 1, 2, \dots, K$, and where $\Delta_i = (l_i, p_i, \gamma_i)$, $i = 1, 2, \dots, m$. We set

$$\begin{aligned} w_1 &= 0 \\ w_{i+1} &= \begin{cases} w_i c_i & \text{if } \gamma_{i+1} = \gamma_i \\ w_i \} & \text{if } \gamma_{i+1} = J_j \gamma_i, \\ w_i \} & \text{if } \gamma_i = J_j \gamma_{i+1} \end{cases} \quad 1 \leq i < m. \end{aligned}$$

[Note that $\gamma_{i+1} = \gamma_i$ is equivalent to $l_{i+1} = l_i + 1$ and is the case when a transition of form (i) is used in getting from Δ_i to Δ_{i+1} ; the remaining two cases occur when transitions of the form (iii) or (ii), respectively, are used.] Now let $w = w_m$, so that $\text{Er}_P(w) = v$ and $m = |w| + 1$. This word w is called the *record* of the given v -computation $\Delta_1, \dots, \Delta_m$ by \mathcal{M} . From w we can read off not only the word v but also the sequence of “pushes” and “pops” as they occur.

Now we want to modify the pushdown automaton \mathcal{M} of Theorem 8.2 so that it will accept $L(\Gamma)$ instead of $L(\Gamma_s)$. In doing so we will have to give up determinism. The intuitive idea is to use nondeterminism by permitting our modified pushdown automaton to “guess” the location of the “brackets” \langle, \rangle . Thus, continuing to use the notation of the proof of Theorem 8.2, we

define a pushdown automaton $\bar{\mathcal{M}}$ with the same states, initial state, accepting states, and pushdown alphabet as \mathcal{M} . However, the tape alphabet of $\bar{\mathcal{M}}$ will be T (rather than $T \cup P$). The transitions of $\bar{\mathcal{M}}$ are

- (a) for each $a \in T$, $qa\emptyset : 0p$, where $p = \delta(q, a)$ [i.e., the same as the transitions (a) of \mathcal{M}];
- (b) for $i = 1, 2, \dots, n$, $q00 : J_i p_i$, where $p_i = \delta(q, \langle \rangle)$;
- (c) for $i = 1, 2, \dots, n$, $q0J_i : 0\bar{p}_i$ where $\bar{p}_i = \delta(q, \langle \rangle)$.

Depending on the transition function δ , $\bar{\mathcal{M}}$ can certainly be nondeterministic. We shall prove that $L(\bar{\mathcal{M}}) = L(\Gamma)$. Note that $\bar{\mathcal{M}}$ is atomic (although \mathcal{M} is not).

First, let $v \in L(\Gamma)$. Then, since $Er_p(L(\Gamma_s)) = L(\Gamma)$, there is a word $w \in L(\Gamma_s)$ such that $Er_p(w) = v$. By Theorem 8.2, $w \in L(\mathcal{M})$. Let $\Delta_1, \Delta_2, \dots, \Delta_m$ be an accepting w -computation by \mathcal{M} (where in fact $m = |w| + 1$). Let

$$\Delta_i = (i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Let $n_i = 1$ if $w: \Delta_i \vdash_{\mathcal{M}} \Delta_{i+1}$ via a transition belonging to group (a); otherwise $n_i = 0$, $1 \leq i < m$. Let

$$l_1 = 1,$$

$$l_{i+1} = l_i + n_i, \quad 1 \leq i < m.$$

Finally let

$$\bar{\Delta}_i = (l_i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Then, as is easily checked,

$$u: \bar{\Delta}_i \vdash_{\bar{\mathcal{M}}} \bar{\Delta}_{i+1}, \quad 1 \leq i < m.$$

Since $\bar{\Delta}_m = (|u| + 1, q, 0)$ with $q \in F$, we have $u \in L(\bar{\mathcal{M}})$.

Conversely, let $v \in L(\bar{\mathcal{M}})$. Let $\bar{\Delta}_1, \bar{\Delta}_2, \dots, \bar{\Delta}_m$ be an accepting v -computation by $\bar{\mathcal{M}}$, where we may write

$$\bar{\Delta}_i = (l_i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Using the fact that $\bar{\mathcal{M}}$ is atomic, we can let w be the *record* of this computation in the sense defined above so that $Er_p(w) = v$ and $m = |w| + 1$. We write

$$\Delta_i = (i, p_i, \gamma_i), \quad i = 1, 2, \dots, m,$$

and easily observe that

$$w: \Delta_i \vdash_{\mathcal{M}} \Delta_{i+1}, \quad i = 1, 2, \dots, m.$$

[In effect, whenever $\bar{\mathcal{M}}$ pushes J_i onto its stack, $\langle \rangle$ is inserted into w ; and whenever $\bar{\mathcal{M}}$ pops J_i , $\langle \rangle$ is inserted into w . This makes the transitions (b), (c) of \mathcal{M} behave on w just the way the corresponding transitions of $\bar{\mathcal{M}}$ behave]

on v .] Since $p_m \in F$ and $\gamma_m = 0, \Delta_1, \Delta_2, \dots, \Delta_m$ is an accepting w -computation by \mathcal{M} . Thus, by Theorem 8.2, $w \in L(\Gamma_s)$. Hence $v \in L(\Gamma)$.

We have shown that $L(\Gamma) = L(\bar{\mathcal{M}})$. Hence we have proved

Theorem 8.3. Let Γ be a Chomsky normal form grammar. Then there is a pushdown automaton $\bar{\mathcal{M}}$ such that $L(\bar{\mathcal{M}}) = L(\Gamma)$.

Now let L be any context-free language. By Theorems 1.2 and 3.1 there is a Chomsky normal form grammar Γ such that $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$. In the former case, we have shown how to obtain a pushdown automaton $\bar{\mathcal{M}}$ such that $L = L(\bar{\mathcal{M}})$. For the latter case we first modify the dfa \mathcal{M}_0 used in the proof of Theorem 8.2 so that it is nonrestarting. We know that this can be done without changing the regular language which \mathcal{M}_0 accepts by Theorem 4.1 in Chapter 8. By carrying out the construction of a pushdown automaton $\bar{\mathcal{M}}$ for which $L(\bar{\mathcal{M}}) = L(\Gamma)$ using the modified version of \mathcal{M}_0 , $\bar{\mathcal{M}}$ will have the property that none of its transitions has q_1 as its final symbol. That is, $\bar{\mathcal{M}}$ will never return to its initial state. Thus, if we define \mathcal{M}' to be exactly like $\bar{\mathcal{M}}$ except for having as its set of accepting states

$$F' = F \cup \{q_1\},$$

we see that $L(\mathcal{M}') = L(\bar{\mathcal{M}}) \cup \{0\} = L(\Gamma) \cup \{0\}$. Thus we have proved

Theorem 8.4. For every context-free language L , there is a pushdown automaton \mathcal{M} such that $L = L(\mathcal{M})$.

We will end this section by proving the converse of this result. Thus we must begin with a pushdown automaton and prove that the language it accepts is context-free. As a first step toward this goal, we will show that we can limit our considerations to atomic pushdown automata.

Theorem 8.5. Let \mathcal{M} be a pushdown automaton. Then there is an atomic pushdown automaton $\bar{\mathcal{M}}$ such that $L(\mathcal{M}) = L(\bar{\mathcal{M}})$.

Proof. For each transition

$$paU : Vq$$

of \mathcal{M} for which $a, U, V \neq 0$, we introduce two new states r, s and let $\bar{\mathcal{M}}$ have the transitions

$$pa0 : 0r,$$

$$r0U : 0s,$$

$$s00 : Vq.$$

If exactly one of a, U, V is 0, then only two transitions are needed for $\bar{\mathcal{M}}$, the transition containing three 0's is omitted and the two states appearing in it are identified. Otherwise, $\bar{\mathcal{M}}$ is exactly like \mathcal{M} . Clearly

$$L(\bar{\mathcal{M}}) = L(\mathcal{M}). \quad \blacksquare$$

Theorem 8.6. For every pushdown automaton \mathcal{M} , $L(\mathcal{M})$ is a context-free language.

Proof. Without loss of generality, by using Theorem 8.5 we can assume that \mathcal{M} is atomic. Let \mathcal{M} have states $Q = \{q_1, \dots, q_m\}$, initial state q_1 , final states F , tape alphabet T , and pushdown alphabet $\Omega = \{J_1, \dots, J_n\}$. Let $P = \{\langle , \rangle_i | i = 1, 2, \dots, n\}$. If $\Delta_1, \Delta_2, \dots, \Delta_m$ is a u -computation by \mathcal{M} (not necessarily accepting), where

$$\Delta_i = (l_i, p_i, \gamma_i), \quad i = 1, 2, \dots, m,$$

we can form the *record* w of this computation, where $w \in (T \cup P)^*$. Let R be the set of all records w of such computations $\Delta_1, \dots, \Delta_m$ for which $l_1 = 1$, $p_1 = q_1$ (the initial state of \mathcal{M}), and $p_m \in F$. (These are all but one of the requirements that the computation be accepting. The missing requirement is $\gamma_m = 0$.) We claim that R is a regular language. To prove this claim we simply construct an ndfa \mathcal{M}_0 which accepts R . \mathcal{M}_0 will have alphabet $T \cup P$, and the same states, initial state, and final states as \mathcal{M} . Its transition function δ is defined as follows:

$$\begin{aligned} \delta(q, a) &= \{p \in Q \mid \mathcal{M} \text{ has the transition } qa0 : 0p\} \text{ for } a \in T, \\ \delta(q, \langle \rangle) &= \{p \in Q \mid \mathcal{M} \text{ has the transition } q00 : J_i p\}, \\ \delta(q, \rangle_i) &= \{p \in Q \mid \mathcal{M} \text{ has the transition } q0J_i : 0p\}. \end{aligned}$$

Then \mathcal{M}_0 accepts w just in case $w \in R$, i.e., $R = L(\mathcal{M}_0)$. This shows that R is regular. Now, by the way in which records are defined we have that $\gamma_i = \gamma_i(w)$, $i = 1, 2, \dots, m$. Moreover, w is a record of an accepting computation by \mathcal{M} if and only if $w \in R$ and $\gamma_m = 0$, i.e., if and only if $w \in R$ and its record is balanced. By Theorem 8.1, the set of records of balanced accepting computations by \mathcal{M} is therefore

$$R \cap \text{PAR}_n(T).$$

Hence

$$L(\mathcal{M}) = \text{Er}_P(R \cap \text{PAR}_n(T)).$$

Finally, by Theorems 5.4 and 7.1 and Corollary 5.6, $L(\mathcal{M})$ is context-free. \blacksquare

Exercises

1. Let $L = \{a^{[n]}b^{[m]}a^{[n]} \mid m, n > 0\} \cup \{a^{[n]}c^{[n]} \mid n > 0\}$.
 - (a) Show that L is context-free.
 - (b) Construct a pushdown automaton which accepts L .
2. Let us call a *generalized pushdown automaton* a device which functions just like a pushdown automaton except that it can write any finite sequence of symbols on the stack in a single step. Show that for every generalized pushdown automaton \mathcal{M} , there is a pushdown automaton $\tilde{\mathcal{M}}$ such that $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$.
- 3.* Let

$$P = \left\{ \begin{array}{|c|} \hline u_i \\ \hline v_i \\ \hline \end{array} \mid i = 1, 2, \dots, k \right\}$$

be a set of dominoes on the alphabet A . Let $B = \{c_1, \dots, c_k\}$ be an alphabet such that $A \cap B = \emptyset$. Let $c \notin A \cup B$. Let

$$R = \{ycy^R \mid y \in A^*B^*\},$$

$$L_1 = \{u_{i_1}u_{i_2} \cdots u_{i_n}c_{i_n}c_{i_{n-1}} \cdots c_{i_2}c_{i_1}\},$$

$$L_2 = \{v_{i_1}v_{i_2} \cdots v_{i_n}c_{i_n}c_{i_{n-1}} \cdots c_{i_2}c_{i_1}\},$$

$$S_p = \{ycz^R \mid y \in L_1, z \in L_2\}.$$

Recall that by Theorem 6.5, the Post correspondence problem P has a solution if and only if $L_1 \cap L_2 \neq \emptyset$.

- (a) Show that the Post correspondence problem P has *no* solution if and only if $R \cap S_p = \emptyset$.
- (b) Show that $(A \cup B \cup \{c\})^* - R$ and $(A \cup B \cup \{c\})^* - S_p$ are both context-free. (*Hint:* Construct push-down automata.)
- (c) From (a) and (b) show how to conclude that there is no algorithm that can determine for a given context-free grammar Γ with terminals T whether $L(\Gamma) \cup \{0\} = T^*$.
- (d) Now show that there is no algorithm that can determine for a given context-free grammar Γ_1 and regular grammar Γ_2 whether
 - (i) $L(\Gamma_1) = L(\Gamma_2)$,
 - (ii) $L(\Gamma_1) \supseteq L(\Gamma_2)$.

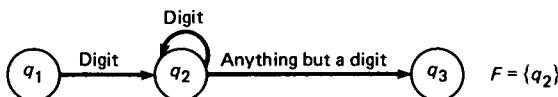
9. Compilers and Formal Languages

A *compiler* is a program which takes as input a program (known as the *source program*) written in a high-level language such as COBOL,

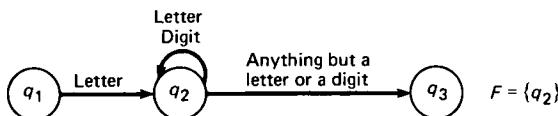
FORTRAN, Pascal, or PL/I and translates it into an equivalent program (known as the *object program*) in a low-level language such as an assembly language or a machine language. Just as in Chapters 2 and 5, we found it easier to write programs with the aid of macros, most programmers find programming in a high-level language faster, easier, and less tedious than in a low-level language. Thus the need for compilers.

The translation process is divided into a sequence of phases, of which the first two are of particular interest to us. *Lexical analysis*, which is the first phase of the compilation process, consists of dividing the characters of the source program into groups called *tokens*. Tokens are the logical units of an instruction and include keywords such as IF, THEN, and DO, operators such as + and *, predicates such as >, variable names, labels, constants, and punctuation symbols such as (and ;.

The reason that the lexical analysis phase of compilation is of interest to us is that it represents an application of the theory of finite automata and regular expressions. The lexical analyzer must identify tokens, determine types, and store this information into a *symbol table* for later use. Typically, compiler writers use nondeterministic finite automata to design these token recognizers. For example, the following is an ndfa which recognizes unsigned integer constants.



Similarly, a nondeterministic finite automaton which recognizes variable names might look like this:



We end our brief discussion of lexical analysis by noting that it is not always a simple task to properly determine the division into tokens. For example, in FORTRAN, the statements

DO 10 I = 1.11

and

DO 10 I = 1,11

look very similar but are in fact totally unrelated instructions. The first is an assignment statement which assigns to a variable named DO10I (embedded

blanks are ignored in FORTRAN) the value 1.11. The second is a **DO** loop which indicates that the body is to be performed 11 times. It is not until the “.” or “,” is encountered that the statement type can be determined.

At the completion of the lexical analysis phase of compilation, tokens have been identified, their types determined, and when appropriate, the value entered in the symbol table. At this point, the second phase of compilation, known as *syntactic analysis* or *parsing*, begins. It is in this second phase that context-free grammars play a central role.

For programming languages which are context-free, the parsing problem amounts to determining for a given context-free grammar Γ and word w

1. whether $w \in L(\Gamma)$, and
2. if $w \in L(\Gamma)$, how w could have been generated.

Intuitively, the parsing phase of the compilation process consists of the construction of derivation or parse trees whose leaves are the tokens identified by the lexical analyzer.

Thus, for example, if our grammar included the productions:

$$S \rightarrow \text{while-statement}$$

$$S \rightarrow \text{assignment-statement}$$

$$\text{while-statement} \rightarrow \text{while cond do } S$$

$$\text{cond} \rightarrow \text{cond} \vee \text{cond}$$

$$\text{cond} \rightarrow \text{rel},$$

$$\text{rel} \rightarrow \text{exp pred exp}$$

$$\text{exp} \rightarrow \text{exp} + \text{exp}$$

$$\text{exp} \rightarrow \text{var}$$

$$\text{exp} \rightarrow \text{const}$$

$$\text{pred} \rightarrow >$$

$$\text{pred} \rightarrow =$$

$$\text{assignment-statement} \rightarrow \text{var} \leftarrow \text{exp}$$

then the parse tree for the statement

$$\text{while } x > y \vee z = 2 \text{ do } w \leftarrow x + 4$$

is given by Fig. 9.1.

The parsing is usually accomplished by simulating the behavior of a pushdown automaton which accepts $L(\Gamma)$ either starting from the root of the tree or the leaves of the tree. In the former case, this is known as *top-down parsing* and in the latter case, *bottom-up parsing*.

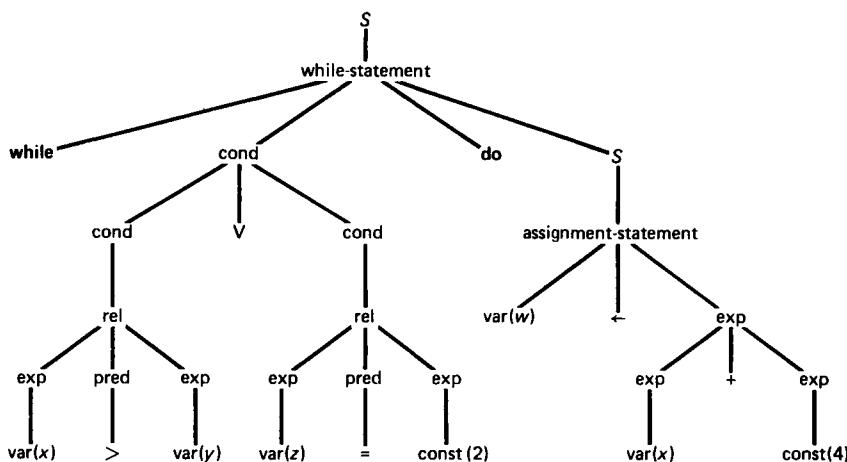


Fig. 9.1

Most programming languages are for the most part context-free. (A major exception is the coordination of declarations and uses.) A common technique involves the definition of a superset of the programming language which can be accepted by a deterministic pushdown automaton. This is desirable since there are particularly fast algorithms for parsing grammars associated with deterministic pushdown automata.

Exercises

1. Give a context-free grammar for generating valid FORTRAN arithmetic expressions over the alphabet $\{a, b, +, -, *, /, \uparrow, (,), \}\}$, where variable names are elements of $\{a, b\}^*$ of length at least 1. Is the grammar ambiguous? What are the implications of this?

Context-Sensitive Languages

1. The Chomsky Hierarchy

We are now going to place our work in the context of Noam Chomsky's hierarchy of grammars and languages. An arbitrary (phrase structure) grammar (recall Chapter 7, Section 5) is called a type 0 grammar. A *context-sensitive grammar* is called a type 1 grammar. A *regular grammar* (recall Chapter 9, Section 2) is called a type 3 grammar. Type 2 or *positive context-free grammars* (recall Chapter 9, Section 1) are simply grammars in which each production has the form $V \rightarrow h$, where V is a variable and $h \neq 0$. The inclusions suggested by the numbering obviously hold: every regular grammar is context-free, and every positive context-free grammar is context-sensitive. (Of course, grammars containing productions of the form $V \rightarrow 0$ cannot be context-sensitive.)

For each type of grammar, there is a corresponding class of languages:

A language L is $\begin{bmatrix} \text{regular} \\ \text{context-free} \\ \text{context-sensitive} \\ \text{r.e.} \end{bmatrix}$ or of type $\begin{bmatrix} 3 \\ 2 \\ 1 \\ 0 \end{bmatrix}$
 if there is a $\begin{bmatrix} \text{regular} \\ \text{positive context-free} \\ \text{context-sensitive} \\ \text{phrase structure} \end{bmatrix}$ grammar Γ

such that

$$L = L(\Gamma) \quad \text{or} \quad L = L(\Gamma) \cup \{0\}.$$

For regular languages this statement is just Theorem 2.3 in Chapter 9. For context-free languages, it is Theorem 1.2 in Chapter 9. For context-

sensitive languages we take it as a definition. For r.e. languages it is Theorem 5.2 in Chapter 7, and the special reference to $\{0\}$ is not needed. We have

Theorem 1.1. Every regular language is context-free. Every context-free language is context-sensitive. Every context-sensitive language is recursive.

Proof. The first two statements follow simply from the corresponding inclusions among the types of grammar. The third follows at once from Theorem 5.5 in Chapter 7. ■

We would like to show that the inclusions of Theorem 1.1 are proper, that is, that none of the four classes mentioned in the theorem is identical to any of the others. We have seen in Theorem 1.1 in Chapter 9, that the language $L = \{a^{[n]}b^{[n]} | n > 0\}$ is context-free but not regular. Similarly, we saw in Theorem 4.2 in Chapter 9 that the language $\{a^{[n]}b^{[n]}c^{[n]} | n > 0\}$ is not context-free, while Exercise 6 in Chapter 7 shows that it is context-sensitive. This takes care of the first two inclusions of Theorem 1.1. The following theorem takes care of the remaining one:

Theorem 1.2. There is a recursive language on the alphabet $\{1\}$ which is not context-sensitive.

Proof. We first code each context-sensitive grammar Γ with terminal alphabet $\{1\}$ by a string on the 5-letter alphabet $A = \{1, V, b, \rightarrow, /\}$. We do this simply by replacing each variable by a distinct string of the form $Vb^{[j]}$, using the arrow " \rightarrow " as usual between the left and right sides of productions, and using the slash "/" to separate productions. (Of course, not every string on this alphabet is actually the code for a context-sensitive grammar.) Now, the strings which code context-sensitive grammars may be placed in alphabetic order (or equivalently, in numerical order, regarding each string on A as the base 5 notation for an integer, as in Chapter 5). We let L_i be the context-sensitive language generated by the i th context-sensitive grammar in this enumeration, $i = 1, 2, 3, \dots$. Then we set

$$L = \{1^{[i]} | 1^{[i]} \notin L_i, i \neq 0\}.$$

This is, of course, a typical diagonal construction, and we easily show that L is not context-sensitive. For, if $L = L_{i_0}$, then

$$\begin{aligned} 1^{[i_0]} \in L &\quad \text{if and only if } 1^{[i_0]} \notin L_{i_0} \\ &\quad \text{if and only if } 1^{[i_0]} \notin L. \end{aligned}$$

To see that L is recursive we note that there is an algorithm which given i will return a context-sensitive grammar Γ_i that generates L_i . Then $1^{[i]}$ can be tested for membership in L_i using the algorithm developed in the *proof* of Theorem 5.5 in Chapter 7. ■

For each class of languages corresponding to types 0, 1, 2, 3, we are concerned with questions of the following kinds: What can we determine algorithmically about a language from a grammar which generates it? What kinds of device will accept precisely the languages belonging to the class? Under what operations are the classes closed? We have been dealing with these questions for languages of types 0, 2, and 3. Now, we will see what can be said about languages of type 1, i.e., context-sensitive languages. We begin by considering the question of closure under union. We will need the

Lemma. There is an algorithm that will transform a given context-sensitive grammar Γ into a context-sensitive grammar Δ such that the left sides of the productions of Δ contain no terminals and $L(\Gamma) = L(\Delta)$.

Proof. We “disguise” the terminals as variables exactly as in the proof of Theorem 3.1 in Chapter 9, obtaining the desired grammar Δ consisting of productions of the forms (3.1) and (3.2) in that proof. ■

Theorem 1.3. If L_1, L_2 are context-sensitive languages, then so is $L_1 \cup L_2$.

Proof. Assume $L_1 = L(\Gamma_1)$ or $L(\Gamma_1) \cup \{0\}$, $L_2 = L(\Gamma_2)$ or $L(\Gamma_2) \cup \{0\}$, where Γ_1 and Γ_2 are context-sensitive grammars with disjoint sets of variables of the form obtained in the Lemma. We construct Γ from Γ_1 and Γ_2 exactly as in the proof of Theorem 5.1 in Chapter 9, so that Γ is also context-sensitive and $L(\Gamma) = L(\Gamma_1) \cup L(\Gamma_2)$. Clearly, $L_1 \cup L_2 = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$. ■

2. Linear Bounded Automata

We are now going to deal with the question: which devices accept context-sensitive languages? We define a *linear bounded automaton* on the alphabet $C = \{s_1, s_2, \dots, s_n\}$ to be a *nondeterministic Turing machine* \mathcal{M} on the alphabet $C \cup \{\lambda, \rho\}$ such that the only quadruples \mathcal{M} contains beginning $q \lambda$ or $q \rho$ are of the forms $q \lambda R p$ and $q \rho L p$, respectively, such that \mathcal{M} has a *final state*, written \tilde{q} , where no quadruple of \mathcal{M} begins \tilde{q} , and finally such that for every quadruple $q a b p$ in \mathcal{M} , we have $b \neq \lambda, \rho$. Thus, when scanning λ , \mathcal{M} can only move right, and when scanning ρ , \mathcal{M} can only move left, and the symbols λ, ρ can never be printed in the course of a computation. Thus, the effect of the additional symbols λ and ρ is simply to prevent the machine from moving beyond the confines of the given string on the tape. Because of this we can code a configuration of \mathcal{M} by a triple $(i, q, \lambda w \rho)$, where $0 \leq i \leq |w| + 1$; i gives the position of the tapehead (i.e., of the scanned square), q is the current state; and $\lambda w \rho$ is the tape contents, $w \in (C \cup \{s_0\})^*$. (Recall that s_0 is the blank.) As usual, for configurations γ, δ we write $\gamma \vdash_{\mathcal{M}} \delta$ to mean

Table 2.1

Quadruple in \mathcal{M}	Corresponding transition
$q \ a \ b \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u , p, ubv)$
$q \ a \ R \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u +1, p, uav)$
$q \ a \ L \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u -1, p, uav)$

that one of the quadruples of \mathcal{M} permits the transition from γ to δ , and write $\gamma \xrightarrow{\mathcal{M}} \delta$ to mean that there is a sequence of configurations $\gamma = \gamma_1, \gamma_2, \dots, \gamma_k = \delta$ such that $\gamma_i \xrightarrow{\mathcal{M}} \gamma_{i+1}$ for $1 \leq i < k$. Table 2.1 shows which transitions are permitted by each quadruple in \mathcal{M} (here $a \in C \cup \{s_0, \lambda, \rho\}$, $b \in C \cup \{s_0\}$). (Of course, for $a = \lambda, \rho$, only quadruples of the second and third kind, respectively, can occur in \mathcal{M} .)

\mathcal{M} is said to *accept* a string $w \in C^*$ if

$$(1, q_1, \lambda w \rho) \xrightarrow{\mathcal{M}} (i, \tilde{q}, w'),$$

where q_1 is the initial state of \mathcal{M} and, of course, \tilde{q} is the final state. (Note carefully, that unlike the situation for Turing machines, a configuration will be regarded as “accepting” only if \mathcal{M} is in its final state \tilde{q} .) If $A \subseteq C$, we write $L_A(\mathcal{M})$ for the set of all $w \in A^*$ that are accepted by \mathcal{M} . The main theorem is

Theorem 2.1 (Landweber-Kuroda). The language $L \subseteq A^*$ is context-sensitive if and only if there is a linear bounded automaton \mathcal{M} such that $L = L_A(\mathcal{M})$.

We begin with

Lemma 1. There is an algorithm which transforms any given context-sensitive grammar Γ with terminals T into a linear bounded automaton \mathcal{M} such that $L(\Gamma) = L_T(\mathcal{M})$.

Proof. Let \mathcal{V} be the set of variables of Γ , and let $S \in \mathcal{V}$ be the start symbol. The alphabet of \mathcal{M} will be $T \cup \mathcal{V}$. Let the productions of Γ be $u_i \rightarrow v_i$, $i = 1, 2, \dots, m$, where

$$u_i = \alpha_1^{(i)} \alpha_2^{(i)} \cdots \alpha_{k_i}^{(i)} \quad \text{and} \quad v_i = \beta_1^{(i)} \beta_2^{(i)} \cdots \beta_{l_i}^{(i)}; \quad (2.1)$$

$$\alpha_1^{(i)}, \alpha_2^{(i)}, \dots, \alpha_{k_i}^{(i)}, \beta_1^{(i)}, \beta_2^{(i)}, \dots, \beta_{l_i}^{(i)} \in T \cup \mathcal{V},$$

and $k_i \leq l_i$. Then we set

$$\alpha_{k_i+1}^{(i)} = \alpha_{k_i+2}^{(i)} = \cdots = \alpha_{l_i}^{(i)} = s_0.$$

That is, we fill out the left side of each production with blanks. Since \mathcal{M} is operating nondeterministically, it can seek the word v_i on the tape and

replace it by u_i , thus undoing the work of the production. It will help in following the construction of the automaton \mathcal{M} if we think of it as operating in one of these four phases: *initialization*, *searching*, *production-undoing*, and *termination*. The states of \mathcal{M} will be the *initial state* q_1 , the *search state* σ , the *return state* $\bar{\sigma}$, the *undoing states* $p_j^{(i)}, q_j^{(i)}$ for $1 \leq i \leq m$ and $1 \leq j \leq l_i$ [l_i is as defined in Eqs. (2.1)], and the *termination states* $\tau, \bar{\tau}$.

PHASE 1 (Initialization). We place in \mathcal{M} the quadruples

$$\left. \begin{array}{ll} q_1 & a \ a \ \sigma \\ q_1 & a \ a \ \tau \end{array} \right\} \quad a \in \mathcal{V} \cup T \cup \{s_0\}.$$

Thus in Phase 1, \mathcal{M} operating nondeterministically “decides” to enter either the search or the termination phase.

PHASE 2 (Search). We place in \mathcal{M} the quadruples

$$\begin{aligned} \sigma & a & R & \sigma & a \neq \rho \\ \sigma & \rho & L & \bar{\sigma} \\ \sigma & \beta_1^{(i)} & \beta_1^{(i)} & p_1^{(i)} & 1 \leq i \leq m \\ \bar{\sigma} & a & L & \bar{\sigma} & a \neq \lambda \\ \bar{\sigma} & \lambda & R & q_1. \end{aligned}$$

In Phase 2, \mathcal{M} moves right along the tape searching for one of the initial symbols $\beta_1^{(i)}$ of the right side of a production. Finding one, \mathcal{M} may enter an *undoing state*. If \mathcal{M} encounters the right end marker ρ while still in state σ , it enters the *return state* $\bar{\sigma}$ and goes back to the beginning.

PHASE 3 (Production-undoing). We place in \mathcal{M} the quadruples, for $1 \leq j < l_i, 1 \leq i \leq m$,

$$\begin{aligned} p_j^{(i)} & \beta_j^{(i)} & \alpha_j^{(i)} & q_j^{(i)} \\ q_j^{(i)} & \alpha_j^{(i)} & R & p_{j+1}^{(i)} \\ p_{l_i}^{(i)} & \beta_{l_i}^{(i)} & \alpha_{l_i}^{(i)} & \bar{\sigma} \end{aligned}$$

together with the quadruples

$$p_j^{(i)} \ s_0 \ R \ p_j^{(i)}.$$

When operating in phase 3, \mathcal{M} has the opportunity to replace the right side of one of the productions on the tape by the left side (ignoring all blanks). If \mathcal{M} succeeds, it can enter the return state $\bar{\sigma}$, return to the left, and begin again.

PHASE 4 (Termination). We place in \mathcal{M} the quadruples

$$\begin{aligned} & \tau s_0 R \tau \\ & \tau S R \bar{\tau} \\ & \bar{\tau} s_0 R \bar{\tau} \\ & \bar{\tau} \rho L \tilde{q}. \end{aligned}$$

Thus if \mathcal{M} ever returns to state q_1 with the tape contents

$$\lambda s_0^{(i)} S s_0^{(j)} \rho \quad i, j \geq 0$$

(where, of course, S is the start symbol of Γ), then \mathcal{M} will have the opportunity to move all the way to the right in this phase and to enter the final state \tilde{q} .

Thus, \mathcal{M} will accept a word $w \in T^*$ just in case there is a derivation of w from S in Γ . ■

Lemma 2. If $L \subseteq A^*$ is a context-sensitive language, then there is a linear bounded automaton \mathcal{M} such that $L = L_A(\mathcal{M})$.

Proof. We have $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$ for a context-sensitive grammar Γ . In the first case, \mathcal{M} can be obtained as in Lemma 1. In the second case, we modify the automaton \mathcal{M} of Lemma 1 by adding the quadruple $q_1 \rho L \tilde{q}$. The modified automaton accepts 0 as well as the strings that \mathcal{M} accepts. ■

Now, we wish to discuss the converse situation: we are given a linear bounded automaton \mathcal{M} and alphabet A and wish to obtain a context-sensitive grammar Γ such that $L(\Gamma) = L_A(\mathcal{M}) - \{0\}$. The construction will be similar to the simulation, in Chapter 7, of a Turing machine by a semi-Thue process. However, the coding must be tighter because all the productions need to be non-length-decreasing.

Let \mathcal{M} be the given linear bounded automaton with alphabet C where $A \subseteq C$, initial state q_1 , and final state \tilde{q} . To begin with, we will only consider words $u \in C^*$ for which $|u| \geq 2$; such words can be written awb , where $w \in C^*$, $a, b \in C$. We wish to code a configuration $(i, q, \lambda awb\rho)$ of \mathcal{M} by a word of length $|awb| = |w| + 2$. To help us in doing this, we will use five variants on each letter $a \in C$:

$$a^\downarrow a^\uparrow \overleftarrow{a} \overrightarrow{a}.$$

The interpretation of these markings is

- $'a$: a on the left end of the word;
- a^\downarrow : a on the right end of the word;

- \hat{a} : a on the left end, but the symbol being scanned is λ , one square to the left of a ;
 \vec{a} : a on the right end, but the symbol being scanned is ρ , one square to the right of a .

Finally, the current state will ordinarily be indicated by a subscript on the scanned symbol. If however, the scanned symbol is λ or ρ , the subscript will be on the adjacent symbol, marked, as just indicated, by an arrow. The examples in Table 2.2 should make matters plain. Of course, this encoding only works for words $\lambda w\rho$ for which $|w| \geq 2$.

Table 2.2

Configuration	Code
$(3, q, \lambda ababc\rho)$	$'aba_qbc'$
$(1, q, \lambda ababc\rho)$	$'a_qbabc'$
$(5, q, \lambda ababc\rho)$	$'ababc'_q$
$(0, q, \lambda ababc\rho)$	\overleftarrow{a}_qbabc'
$(6, q, \lambda ababc\rho)$	$'abab\overrightarrow{c}_q'$

Now we will construct a semi-Thue process Σ such that given configurations γ, δ of \mathcal{M} and their codes $\bar{\gamma}, \bar{\delta}$, respectively, we shall have

$$\gamma \vdash_{\mathcal{M}} \delta \quad \text{if and only if} \quad \bar{\gamma} \Rightarrow_{\Sigma} \bar{\delta}.$$

As for Turing machines, we define Σ by introducing suitable productions corresponding to each quadruple of \mathcal{M} . The correspondence is shown in Table 2.3.

Now, since these productions simulate the behavior of \mathcal{M} in an obvious and direct manner, we see that \mathcal{M} will accept the string aub , $u \in C^*$, $a, b \in C$, just in case there is a derivation, from the initial word $'a_{q_1}ub'$ using these productions, of a word containing \tilde{q} as a subscript. To put this result in a more manageable form, we add to the alphabet of Σ the symbol S and add to Σ the “cleanup” productions

$$\alpha_{\tilde{q}} \rightarrow S, \quad \alpha S \rightarrow S, \quad S\alpha \rightarrow S, \quad (2.2)$$

where α can be any one of a , $'a$, a' , \overleftarrow{a} , or \overrightarrow{a} , for any $a \in C$. Since these productions will transform the codes for configurations with the final state \tilde{q} into the single symbol S , and since there is no other way to obtain the single symbol S using the productions of Σ , we have

Lemma 3. \mathcal{M} accepts the string aub , $a, b \in C$, $u \in C^*$, if and only if

$$'a_{q_1}ub' \xrightarrow{\Sigma} S.$$

Table 2.3

Quadruple of \mathcal{M}	Productions of Σ
$q \ a \ b \ p, \quad a, b \in C$	$a_q \rightarrow b_p$ $'a_q \rightarrow 'b_p$ $a_q^l \rightarrow b_p^l$
$q \ a \ R \ p, \quad a \in C$	$a_q b \rightarrow ab_p$ $'a_q b \rightarrow 'ab_p$ $a_q b^l \rightarrow ab_p^l$ $'a_q b^l \rightarrow 'ab_p^l$ $a_q^l \rightarrow \vec{a}_p$
$q \ \lambda \ R \ p$	$\vec{a}_q \rightarrow 'a_p, \quad \text{all } a \in C$
$q \ a \ L \ p, \quad a \in C$	$ba_q \rightarrow b_p a$ $ba_q^l \rightarrow b_p a^l$ $'ba_q \rightarrow 'b_p a$ $'ba_q^l \rightarrow 'b_p a^l$ $a_q \rightarrow \vec{a}_p$
$q \ p \ L \ p$	$\vec{a}_q \rightarrow a_p^l, \quad \text{all } a \in C$

Now let Ω be the semi-Thue process whose productions are the *inverses* of the productions of Σ . (See Chapter 7, Section 2.) Then we have

Lemma 4. \mathcal{M} accepts the string aub , $a, b \in C$, $u \in C^*$, if and only if

$$S \xrightarrow[\Omega]{} 'a_q ub^l.$$

Now we are ready to define a context-sensitive grammar Γ . Let the terminals of Γ be the members of A , let the variables of Γ be

1. the symbols from the alphabet of Ω that do not belong to A , and
2. symbols a^0 for each $a \in A$.

Finally, the productions of Γ are the productions of Ω together with

$$\left. \begin{array}{l} 'a_{q_1} \rightarrow a^0 \\ a^0 b \rightarrow ab^0 \\ a^0 b^l \rightarrow ab \end{array} \right\} \quad \text{for all } a, b \in A. \quad (2.3)$$

It is easy to check that Γ is in fact context-sensitive. [Of course, the productions (2.2) must be read from right to left, since it is the inverses of (2.2) that appear in Γ .] Moreover, using Lemma 4 and (2.3), we have

Lemma 5. Let $w \in A^*$. Then $w \in L(\Gamma)$ if and only if $|w| \geq 2$ and $w \in L_A(\mathcal{M})$.

Now let \mathcal{M} be a given linear bounded automaton, A a given alphabet, and let Γ be the context-sensitive grammar just constructed. Then, by Lemma 5, we have

$$L_A(\mathcal{M}) = L(\Gamma) \cup L_0,$$

where L_0 is the set of words $w \in A^*$ accepted by \mathcal{M} such that $|w| < 2$. But L_0 is finite, hence (Corollary 4.7 in Chapter 8) L_0 is a regular language, and so is certainly context-sensitive. Finally, using Theorem 1.3, we see that $L_A(\mathcal{M})$ is context-sensitive. This, together with Lemma 2, completes the proof of Theorem 2.1. \blacksquare

3. Closure Properties

Remarkably enough, it is still not known whether there is a context-sensitive language $L \subseteq A^*$ for which $A^* - L$ is not context-sensitive. The question of whether the complement of a context-sensitive language must also be context-sensitive remains one of the outstanding open problems in theoretical computer science. We have already seen that the context-sensitive languages are closed under union (Theorem 1.3), and now we consider *intersection*. Here, although the context-free languages are *not* closed under intersection (Theorem 5.2 in Chapter 9), we can prove

Theorem 3.1. If L_1 and L_2 are context-sensitive languages, then so is $L_1 \cap L_2$.

Proof. Let $L_1 = L_A(\mathcal{M}_1)$, $L_2 = L_A(\mathcal{M}_2)$, where $\mathcal{M}_1, \mathcal{M}_2$ are linear bounded automata. The idea of the proof is to test a string w for membership in $L_1 \cap L_2$ by first seeing whether \mathcal{M}_1 will accept w and then, if \mathcal{M}_1 does, to see whether \mathcal{M}_2 will also accept w . The difficulty is the usual one: \mathcal{M}_1 will destroy the input w in the process of testing it. When we were working with Turing machines, we were able to deal with this kind of problem by saving a copy of the input on a part of the tape which remained undisturbed. Since linear bounded automata have no extra space, the problem must be solved another way. The solution uses an important idea: we think of our tape as consisting of a number of separate “tracks,” in this case two tracks. We will construct a linear bounded automaton \mathcal{M} which will work as follows:

1. \mathcal{M} will copy the input so it appears on both the upper and the lower track of the tape;
2. \mathcal{M} will simulate \mathcal{M}_1 working on the upper track only;
3. if \mathcal{M}_1 has accepted, \mathcal{M} will then simulate \mathcal{M}_2 working on the lower track (on which the original input remains undisturbed).

Thus, let us assume that \mathcal{M}_1 and \mathcal{M}_2 both have the alphabet $C = \{s_1, s_2, \dots, s_n\}$. (Of course, in addition they may use the symbols λ, ρ, s_0 .) \mathcal{M} will be a linear bounded automaton using, in addition, the symbols $b_j^i, 0 \leq i, j \leq n$. We think of the presence of the symbol b_j^i as indicating that s_i is on the “upper track” while s_j is on the “lower track” at the indicated position. Finally we assume that q_1 is the initial state of \mathcal{M}_1 , that \bar{q} is its final state, and that q_2 is the initial state of \mathcal{M}_2 . We also assume that the sets of states of \mathcal{M}_1 and \mathcal{M}_2 are disjoint. \mathcal{M} is to have initial state q_0 and have the same final state as \mathcal{M}_2 . \mathcal{M} is to contain the following quadruples (for $0 \leq i \leq n$):

1. Initialization:

$$\begin{aligned} q_0 & s_i & b_i^i & \bar{q} \\ \bar{q} & b_i^i & R & q_0 \\ q_0 & \rho & L & \bar{\bar{q}} \\ \bar{\bar{q}} & b_i^i & L & \bar{\bar{q}} \\ \bar{\bar{q}} & \lambda & R & q_1. \end{aligned}$$

Here $\bar{q}, \bar{\bar{q}}$ are not among the states of \mathcal{M}_1 and \mathcal{M}_2 . These quadruples cause \mathcal{M} to copy the input on both “tracks” and then to return to the leftmost symbol of the input.

2. For each quadruple of \mathcal{M}_1 , the corresponding quadruples, obtained by replacing each s_i by $b_j^i, j = 0, 1, \dots, n$, are to be in \mathcal{M} . These quadruples cause \mathcal{M} to simulate \mathcal{M}_1 operating on the “upper” track. In addition, \mathcal{M} is to have the quadruples for $0 \leq i, j \leq n$:

$$\begin{aligned} \tilde{q} & b_j^i & R & \tilde{q} \\ \tilde{q} & \rho & L & \tilde{p} \\ \tilde{p} & b_j^i & s_j & \tilde{p} \\ \tilde{p} & s_j & L & \tilde{p} \\ \tilde{p} & \lambda & R & q_2. \end{aligned}$$

Here again \tilde{p} does not occur among the states of $\mathcal{M}_1, \mathcal{M}_2$. These quadruples cause \mathcal{M} to restore the “lower” track and then to enter the initial state of \mathcal{M}_2 scanning the leftmost input symbol.

3. Finally, \mathcal{M} is to contain all of the quadruples of \mathcal{M}_2 .

Since it is plain that

$$L_A(\mathcal{M}) = L_A(\mathcal{M}_1) \cap L_A(\mathcal{M}_2),$$

the proof is complete. ■

As an application, we obtain an unsolvability result about context-sensitive grammars.

Theorem 3.2. There is no algorithm for determining of a given context-sensitive grammar Γ whether $L(\Gamma) = \emptyset$.

Proof. Suppose there were such an algorithm. We can show that there would then be an algorithm for determining of two given *context-free* grammars Γ_1, Γ_2 whether $L(\Gamma_1) \cap L(\Gamma_2) = \emptyset$, thus contradicting Theorem 6.6 in Chapter 9. For, since Γ_1, Γ_2 are context-sensitive, the constructive nature of the proofs of Theorems 2.1 and 3.1 will enable us to obtain a context-sensitive grammar Γ with $L(\Gamma) = L(\Gamma_1) \cap L(\Gamma_2)$. ■

We conclude this short chapter by mentioning another major open problem concerning context-sensitive languages:

Is every context-sensitive language accepted by a *deterministic* linear bounded automaton?

Exercises

1. Prove that every context-free language is accepted by a deterministic linear bounded automaton.

2. Let $L \subseteq A^*$ be an r.e. language. Show that there is a context-sensitive language $L' \subseteq (A \cup \{c\})^*$ such that for all $w \in A^*$, we have

$$w \in L \quad \text{if and only if} \quad wc^{[i]} \in L' \text{ for some } i \geq 0.$$

3. Show that for every r.e. language L there is a context-sensitive grammar Γ such that the grammar obtained from Γ by adding a single production of the form $V \rightarrow 0$ generates L . (*Hint:* Use Exercise 2 and take c to be the variable V .)

Propositional Calculus

1. Formulas and Assignments

Let A be some given alphabet and let $\mathcal{A} \subseteq A^*$. Let $B = A \cup \{\neg, \wedge, \vee, \supset, \leftrightarrow, (,)\}$, where we assume that these additional symbols are not already in A . $\neg, \wedge, \vee, \supset, \leftrightarrow$ are called (*propositional*) *connectives*. Then by a *propositional formula over \mathcal{A}* we mean any element of B^* which either belongs to \mathcal{A} or is obtainable from elements of \mathcal{A} by repeated applications of the following operations on B^* :

1. transform α into $\neg\alpha$;
2. transform α and β into $(\alpha \wedge \beta)$;
3. transform α and β into $(\alpha \vee \beta)$;
4. transform α and β into $(\alpha \supset \beta)$;
5. transform α and β into $(\alpha \leftrightarrow \beta)$.

When the meaning is clear from the context, *propositional formulas over \mathcal{A}* will be called \mathcal{A} -*formulas* or even just *formulas* for short. In this context the elements of \mathcal{A} (which are automatically \mathcal{A} -formulas) are called *atoms*.

To make matters concrete we can take $A = \{p, q, r, s, |\}$, and let

$$\mathcal{A} = \{p^{[i]}, q^{[i]}, r^{[i]}, s^{[i]} | i \in N\}.$$

In this case the atoms are called propositional variables. We can think of the suffix $^{[i]}$ as a subscript and write $p_i = p^{[i]}$, $q_i = q^{[i]}$, etc. Here are a few examples of formulas:

$$\begin{aligned} & ((\neg p \supset q) \supset p), \\ & (((p \wedge q) \supset r) \wedge ((p_1 \wedge q_1) \supset r_1)) \supset \neg s), \\ & (((p_1 \vee \neg p_2) \vee p_3) \wedge (\neg p_1 \vee p_3)). \end{aligned}$$

Although the special case of propositional variables really suffices for

studying propositional formulas, it is useful in order to include later applications, to allow the more general case of an arbitrary language of atoms. (In fact our assumption that the atoms form a language is not really necessary.)

By an *assignment* on a given set of atoms \mathcal{A} we mean a function v which maps each atom into the set $\{\text{FALSE}, \text{TRUE}\} = \{0, 1\}$, where (recall Chapter 1, Section 4), as usual, we are identifying FALSE with 0 and TRUE with 1. Thus for each atom α we will have $v(\alpha) = 0$ or $v(\alpha) = 1$. Given an assignment v on a set of atoms \mathcal{A} , we now show how to define a value $\gamma^v \in \{0, 1\}$ for each \mathcal{A} -formula γ . The definition is by recursion and proceeds as follows:

1. If α is an atom, then $\alpha^v = v(\alpha)$;
2. If $\gamma = \neg\beta$, then $\gamma^v = \begin{cases} 1 & \text{if } \beta^v = 0 \\ 0 & \text{if } \beta^v = 1; \end{cases}$
3. $(\alpha \wedge \beta)^v = \begin{cases} 1 & \text{if } \alpha^v = \beta^v = 1 \\ 0 & \text{otherwise;} \end{cases}$
4. $(\alpha \vee \beta)^v = \begin{cases} 0 & \text{if } \alpha^v = \beta^v = 0 \\ 1 & \text{otherwise;} \end{cases}$
5. $(\alpha \supset \beta)^v = \begin{cases} 0 & \text{if } \alpha^v = 1 \text{ and } \beta^v = 0 \\ 1 & \text{otherwise;} \end{cases}$
6. $(\alpha \leftrightarrow \beta)^v = \begin{cases} 1 & \text{if } \alpha^v = \beta^v \\ 0 & \text{otherwise.} \end{cases}$

A set Ω of \mathcal{A} -formulas is said to be *truth-functionally satisfiable*, or just *satisfiable* for short, if there is an assignment v on \mathcal{A} such that $\alpha^v = 1$ for all $\alpha \in \Omega$; otherwise Ω is said to be (truth-functionally) *unsatisfiable*. If $\Omega = \{\gamma\}$ consists of a single formula, then we say that γ is (truth-functionally) *satisfiable* if Ω is; γ is (truth-functionally) *unsatisfiable* if Ω is unsatisfiable. γ is called a *tautology* if $\gamma^v = 1$ for all assignments v . It is obvious that:

Theorem 1.1. γ is a tautology if and only if $\neg\gamma$ is unsatisfiable.

We agree to write $\alpha = \beta$ for \mathcal{A} -formulas α, β to mean that for every assignment v on \mathcal{A} , $\alpha^v = \beta^v$. This convention amounts of thinking of an \mathcal{A} -formula as naming a mapping from $\{0, 1\}^n$ into $\{0, 1\}$ for some $n \in N$, so that two \mathcal{A} -formulas are regarded as the same if they determine the same mappings. [Thus, in high-school algebra one writes $x^2 - 1 = (x - 1)(x + 1)$, although $x^2 - 1$ and $(x - 1)(x + 1)$ are quite different as *expressions*, because they determine the same mappings on numbers.] With this understanding we are able to eliminate some of the connectives in favor of others in a systematic manner. In particular, the equations

$$(\alpha \supset \beta) = (\neg\alpha \vee \beta), \tag{1.1}$$

$$(\alpha \leftrightarrow \beta) = ((\alpha \supset \beta) \wedge (\beta \supset \alpha)) \tag{1.2}$$

Table 1.1

α	β	$\neg\alpha$	$(\neg\alpha \vee \beta)$	$(\alpha \supset \beta)$	$(\beta \supset \alpha)$	$(\alpha \leftrightarrow \beta)$
1	1	0	1	1	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
0	0	1	1	1	1	1

enable us to limit ourselves to the connectives \neg , \wedge , \vee . The truth of these two equations is easily verified by examining the "truth" tables in Table 1.1 which show all four possibilities for the pair α^v, β^v .

With our use of the equal sign all tautologies are equal to one another and likewise all unsatisfiable formulas are equal to one another. Since the equations

$$\alpha^v = 1 \quad \text{for all } v, \quad \beta^v = 0 \quad \text{for all } v$$

determine α to be a tautology and β to be unsatisfiable, it is natural to write 1 for any \mathcal{A} -formula which is a tautology and 0 for any \mathcal{A} -formula which is unsatisfiable. Thus $\alpha = 1$ means that α is a tautology, and $\alpha = 0$ means that α is unsatisfiable.

The system of \mathcal{A} -formulas, under the operations \neg , \wedge , \vee and involving the "constants" 0, 1 obeys algebraic laws, some of which are analogous to laws satisfied by the real numbers under the operations $-$, \cdot , $+$; but there are some striking differences as well. Specifically, we have, for all \mathcal{A} -formulas α, β, γ

absorption:

$$(\alpha \wedge 1) = \alpha$$

$$(\alpha \vee 0) = \alpha$$

contradiction; excluded middle:

$$(\alpha \wedge \neg\alpha) = 0$$

$$(\alpha \vee \neg\alpha) = 1$$

$$(\alpha \wedge 0) = 0$$

$$(\alpha \vee 1) = 1$$

idempotency:

$$(\alpha \wedge \alpha) = \alpha$$

$$(\alpha \vee \alpha) = \alpha$$

commutativity:

$$(\alpha \wedge \beta) = (\beta \wedge \alpha)$$

$$(\alpha \vee \beta) = (\beta \vee \alpha)$$

associativity:

$$(\alpha \wedge (\beta \wedge \gamma)) = ((\alpha \wedge \beta) \wedge \gamma)$$

$$(\alpha \vee (\beta \vee \gamma)) = ((\alpha \vee \beta) \vee \gamma)$$

distributivity:

$$(\alpha \wedge (\beta \vee \gamma)) = ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$$

$$(\alpha \vee (\beta \wedge \gamma)) = ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$$

De Morgan laws:

$$\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta)$$

$$\neg(\alpha \vee \beta) = (\neg\alpha \wedge \neg\beta)$$

double negation:

$$\neg\neg\alpha = \alpha$$

Table 1.2

α	β	$(\alpha \wedge \beta)$	$(\alpha \vee \beta)$
1	1	1	1
0	1	0	1
1	0	0	1
0	0	0	0

These equations, which are easily checked using truth tables, are the basis of the so-called Boolean algebra. In each row, the equations on the left and right can be obtained from one another by simply interchanging all occurrences of “ \vee ” with “ \wedge ” and of “0” with “1.” This is a special case of a general principle. The truth tables in Table 1.2 show that if we think of 0 as representing “TRUE,” and 1, “FALSE” (instead of the other way around), the tables for “ \wedge ” and “ \vee ” will simply be interchanged. Thus a being from another planet watching us doing propositional calculus might be able to guess that that was in fact what we were doing. But this being would have no way to tell which truth value we were representing by 0 and which by 1, and therefore could not say which of the two connectives represents “and” and which “or.” Therefore we have:

GENERAL PRINCIPLE OF DUALITY. *Any correct statement involving \wedge , \vee and 0, 1, can be translated into another correct statement in which 0 and 1 have been interchanged and \wedge and \vee have been interchanged.*

Of course, in carrying out the translation, notions defined in terms of 0, 1, \wedge , and \vee must be replaced by their *duals*. For example, the dual of “ α is a tautology” is “ α is unsatisfiable.” (The first is “ $\alpha^v = 1$ for all v ”; the second is “ $\alpha^v = 0$ for all v ”.) Thus the dual of the correct statement

if α is a tautology, so is $(\alpha \vee \beta)$

is the equally correct statement

if α is unsatisfiable, so is $(\alpha \wedge \beta)$.

Returning to our list of algebraic laws, we note that in particular the operations \wedge and \vee are commutative and associative. We take advantage of this associativity to write simply

$$\bigwedge_{i \leq k} \alpha_i = (\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k)$$

$$\bigvee_{i \leq k} \alpha_i = (\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_k)$$

without bothering to specify any particular grouping of the indicated

formulas. We freely omit parentheses that are not necessary to avoid ambiguity.

Exercises

1. For each of the following formulas tell whether it is (a) satisfiable, (b) a tautology, (c) unsatisfiable:

- (i) $((p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r)))$.
- (ii) $((p \supset (q \supset r)) \leftrightarrow ((p \wedge q) \supset r))$.
- (iii) $(p \wedge \neg q)$.
- (iv) $((p \vee q) \supset p)$.
- (v) $(\neg(p \supset q) \supset (p \wedge \neg q))$.

2. Apply the general principle of duality to each of the following true statements:

- (i) $(p \vee \neg p)$ is a tautology.
- (ii) $(p \supset (q \supset p))$ is a tautology.

3. Prove that if α and β are formulas, then $\alpha = \beta$ if and only if the formula $(\alpha \leftrightarrow \beta)$ is a tautology.

2. Tautological Inference

Let $\gamma_1, \gamma_2, \dots, \gamma_n, \gamma$ be \mathcal{A} -formulas. Then we write

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$$

and call γ a *tautological consequence* of the premises $\gamma_1, \dots, \gamma_n$ if for every assignment v on \mathcal{A} for which $\gamma_1^v = \gamma_2^v = \dots = \gamma_n^v = 1$, we have also $\gamma^v = 1$. This relation of tautological consequence is the most important concept in the propositional calculus. However, we can easily prove

Theorem 2.1. The relation $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$ is equivalent to each of the following:

1. the formula $((\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is a tautology;
2. the formula $(\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n \wedge \neg \gamma)$ is unsatisfiable.

Proof. $((\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is *not* a tautology just in case for some assignment v , $(\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n)^v = 1$ but $\gamma^v = 0$. That is, just in case for

some assignment v , $\gamma_1^v = \gamma_2^v = \cdots = \gamma_n^v = 1$ but $\gamma^v = 0$, which means simply that it is not the case that $\gamma_1, \gamma_2, \dots, \gamma_n \models \gamma$. Likewise

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n \wedge \neg\gamma)$$

is *satisfiable* if and only if for some assignment v , $\gamma_1^v = \gamma_2^v = \cdots = \gamma_n^v = (\neg\gamma)^v = 1$, i.e., $\gamma_1^v = \gamma_2^v = \cdots = \gamma_n^v = 1$, but $\gamma^v = 0$. ■

Thus the problem of tautological inference is reduced to testing a formula for satisfiability, or for being a tautology. Of course, in principle, such a test can be carried out by simply constructing a truth table. However, a truth table for a formula containing n different atoms will require 2^n rows. Hence, truth table construction may be quite unfeasible even for formulas of modest size.

Consider the example

$$((p \wedge q) \supset (r \wedge s)), ((p_1 \wedge q_1) \supset r_1), ((r_1 \wedge s) \supset s_1), p, q, q_1, p_1 \models s_1. \quad (2.1)$$

Since there are 8 atoms, a truth table would contain $2^8 = 256$ rows. In this example we can reason directly. If v makes all the premises TRUE, then $(p \wedge q)^v = (p_1 \wedge q_1)^v = 1$. Therefore, $(r \wedge s)^v = r_1^v = 1$, and in particular $s^v = 1$. Thus, $(r_1 \wedge s)^v = 1$ and finally, $s_1^v = 1$. We will use Theorem 2.1 to develop more systematic methods for doing such problems.

Exercises

1. Which of the following are correct:

- (i) $(p \supset q), p \models q$.
- (ii) $(p \supset q), q \models p$.
- (iii) $(p \supset q), \neg q \models \neg p$.
- (iv) $(p \supset (q \supset r)), (\neg s \vee p), q \models (s \supset r)$.

2. Apply Theorem 2.1 to Exercise 1.

3. Normal Forms

We will now describe some algebraic procedures for simplifying \mathcal{A} -formulas:

(I) ELIMINATE \supset AND \leftrightarrow .

Simply use Eq. (1.2) for each occurrence of \leftrightarrow . After all such occurrences have been eliminated, use Eq. (1.1) for each occurrence of \supset .

Assuming (I) accomplished, we move on to

(II) MOVE \neg INWARD.

For any occurrence of \neg which is not immediately to the left of an atom either

1. the occurrence immediately precedes another \neg , in which case the pair $\neg\neg$ can be eliminated using the law of double negation; or
2. the occurrence immediately precedes an \mathcal{A} -formula of the form $(\alpha \wedge \beta)$ or $(\alpha \vee \beta)$, in which case one of the De Morgan laws can be applied to move the \neg inside the parentheses.

After (II) has been applied some finite number of times, a formula will be obtained to which (II) can no longer be applied. Such a formula must have each \neg immediately preceding an atom.

As an example of the use of (I) and (II) consider the formula

$$(((p \leftrightarrow q) \supset (r \supset s)) \wedge (q \supset \neg(p \wedge r))). \quad (3.1)$$

Eliminating \leftrightarrow gives

$$(((p \supset q) \wedge (q \supset p)) \supset (r \supset s)) \wedge (q \supset \neg(p \wedge r)).$$

Eliminate \supset :

$$(\neg((\neg p \vee q) \wedge (\neg q \vee p)) \vee (\neg r \vee s)) \wedge (\neg q \vee \neg(p \wedge r)). \quad (3.2)$$

Move \neg inward:

$$(\neg(\neg p \vee q) \vee \neg(\neg q \vee p) \vee (\neg r \vee s)) \wedge (\neg q \vee \neg p \vee \neg r).$$

Move \neg inward:

$$((p \wedge \neg q) \vee (q \wedge \neg p) \vee \neg r \vee s) \wedge (\neg q \vee \neg p \vee \neg r). \quad (3.3)$$

A formula λ is called a *literal* if either λ is an atom or λ is $\neg\alpha$, where α is an atom. Note that if $\lambda = \neg\alpha$, for α an atom, then $\neg\lambda = \neg\neg\alpha = \alpha$. For α an atom it is convenient to write $\bar{\alpha}$ for $\neg\alpha$.

With this notation (3.3) becomes

$$((p \wedge \bar{q}) \vee (q \wedge \bar{p}) \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r}). \quad (3.4)$$

The distributive laws can be used to carry out further simplification, analogous to “multiplying out” in elementary algebra. However, the fact that there are two distributive laws available is a complication because the “multiplying out” can proceed in two directions. As we shall see, each direction gives rise to a specific so-called *normal form*.

A handy technique that makes use of the reader’s facility with elementary algebra is to actually replace the symbols \wedge , \vee by $+$, \cdot and then calculate as in ordinary algebra. Since there are two distributive laws available, correct results will be obtained either by replacing \wedge by $+$ and \vee by \cdot or vice versa.

Thus, writing \cdot for \wedge (and even omitting the \cdot as in elementary algebra) and $+$ for \vee , (3.4) can be written

$$\begin{aligned}
 & (p\bar{q} + q\bar{p} + \bar{r} + s) \cdot (\bar{q} + \bar{p} + \bar{r}) \\
 &= p\bar{q}\bar{q} + p\bar{q}\bar{p} + p\bar{q}\bar{r} + q\bar{p}\bar{q} + q\bar{p}\bar{p} + q\bar{p}\bar{r} + \bar{r}\bar{q} + \bar{r}\bar{p} + \bar{r}\bar{r} \\
 &\quad + s\bar{q} + s\bar{p} + s\bar{r} \\
 &= p\bar{q} + 0 + p\bar{q}\bar{r} + 0 + q\bar{p} + q\bar{p}\bar{r} + \bar{r}\bar{q} + \bar{r}\bar{p} + \bar{r} + s\bar{q} + s\bar{p} + s\bar{r} \\
 &= (p \wedge \bar{q}) \vee (p \wedge \bar{q} \wedge \bar{r}) \vee (q \wedge \bar{p}) \vee (q \wedge \bar{p} \wedge \bar{r}) \\
 &\quad \vee (\bar{r} \wedge \bar{q}) \vee (\bar{r} \wedge \bar{p}) \vee \bar{r} \vee (s \wedge \bar{q}) \vee (s \wedge \bar{p}) \vee (s \wedge \bar{r}), \tag{3.5}
 \end{aligned}$$

where we have used the principles of contradiction and absorption. Alternatively, writing $+$ for \wedge and \cdot for \vee , (3.4) can be written

$$\begin{aligned}
 & (p + \bar{q})(q + \bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (pq + p\bar{p} + \bar{q}q + \bar{q}\bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (pq + 1 + 1 + \bar{q}\bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= pq\bar{r}s + \bar{q}\bar{p}\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (p \vee q \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r}). \tag{3.6}
 \end{aligned}$$

Let λ_i be a sequence of *distinct* literals, $1 \leq i \leq n$. Then the formula $\bigvee_{i \leq n} \lambda_i$ is called an \vee -*clause* and the formula $\bigwedge_{i \leq n} \lambda_i$ is called an \wedge -*clause*. A pair of literals λ, λ' are called *mates* if $\lambda' = \neg \lambda$. We have

Theorem 3.1. Let λ_i be a literal for $1 \leq i \leq n$. Then the following are equivalent:

1. $\bigvee_{i \leq n} \lambda_i$ is a tautology;
2. $\bigwedge_{i \leq n} \lambda_i$ is unsatisfiable;
3. some pair λ_i, λ_j , $1 \leq i, j \leq n$, is a pair of mates.

Proof. If $\lambda_j = \neg \lambda_i$, then obviously, $\bigvee_{i \leq n} \lambda_i$ is a tautology and $\bigwedge_{i \leq n} \lambda_i$ is unsatisfiable. If, on the other hand, the λ_i contain no pair of mates, then there are assignments v, w such that $v(\lambda_i) = 1, w(\lambda_i) = 0$ for $1 \leq i \leq n$. Then $(\bigvee_{i \leq n} \lambda_i)^w = 0, (\bigwedge_{i \leq n} \lambda_i)^v = 1$, so that $\bigvee_{i \leq n} \lambda_i$ is not a tautology and $\bigwedge_{i \leq n} \lambda_i$ is satisfiable. ■

Let κ_i , $1 \leq i \leq n$, be a sequence of *distinct* \vee -*clauses*. Then the \mathcal{A} -formula $\bigwedge_{i \leq n} \kappa_i$ is said to be in conjunctive normal form (CNF). Dually, if κ_i , $1 \leq i \leq n$, is a sequence of *distinct* \wedge -*clauses*, then the \mathcal{A} -formula $\bigvee_{i \leq n} \kappa_i$ is in disjunctive normal form (DNF). Note that (3.6) is in CNF and (3.5) is in DNF. We say that (3.6) is a *CNF of (3.1)* and that (3.5) is a *DNF of (3.1)*. It should be clear that the procedures we have been describing will yield a CNF and a DNF for each \mathcal{A} -formula. Thus we have

Theorem 3.2. There is an algorithm which will transform any given \mathcal{A} -formula α into a formula β in CNF such that $\beta = \alpha$. There is a similar (in fact, dual) algorithm for DNF.

Because of Theorem 2.1, the following result is of particular importance.

Theorem 3.3. A formula in CNF is a tautology if and only if each of its \vee -clauses is a tautology. Dually, a formula in DNF is unsatisfiable if and only if each of its \wedge -clauses is unsatisfiable.

Proof. Let $\alpha = \bigwedge_{i \leq n} \kappa_i$, where each κ_i is an \vee -clause. If each κ_i is a tautology, then for any assignment v we have $\kappa_i^v = 1$ for $1 \leq i \leq n$, so that $\alpha^v = 1$; hence α is a tautology. If some κ_i is not a tautology, then there is an assignment v such that $\kappa_i^v = 0$; hence $\alpha^v = 0$ and α is not a tautology.

The proof for DNF is similar. Alternatively, we can invoke the general principle of duality. ■

Let us try to use these methods in applying Theorem 2.1 to example (2.1). First using Theorem 2.1(1):

We wish to know whether the following formula is a tautology:

$$\begin{aligned} (((p \wedge q) \supset (r \wedge s)) \wedge ((p_1 \wedge q_1) \supset r_1) \\ \wedge ((r_1 \wedge s) \supset s_1) \wedge p \wedge q \wedge q_1 \wedge p_1) \supset s_1. \end{aligned}$$

Use of (I) yields

$$\begin{aligned} (\neg((\neg(p \wedge q) \vee (r \wedge s)) \wedge (\neg(p_1 \wedge q_1) \vee r_1) \\ \wedge (\neg(r_1 \wedge s) \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1) \vee s_1). \end{aligned}$$

Use of (II) gives

$$\begin{aligned} (\neg(\neg(p \wedge q) \vee (r \wedge s)) \vee \neg(\neg(p_1 \wedge q_1) \vee r_1) \\ \vee \neg(\neg(r_1 \wedge s) \vee s_1) \vee \neg p \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1). \end{aligned}$$

Use of (II) again yields

$$\begin{aligned} ((p \wedge q \wedge \neg(r \wedge s)) \vee (p_1 \wedge q_1 \wedge \neg r_1) \vee (r_1 \wedge s \wedge \neg s_1) \vee \neg p \\ \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1). \end{aligned}$$

One final use of (II) gives

$$\begin{aligned} ((p \wedge q \wedge (\neg r \vee \neg s)) \vee (p_1 \wedge q_1 \wedge \neg r_1) \\ \vee (r_1 \wedge s \wedge \neg s_1) \vee \neg p \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1). \quad (3.7) \end{aligned}$$

To apply Theorem 3.3, it is necessary to find a CNF of (3.7). So we replace \wedge by $+$ and \vee by \cdot :

$$(p + q + \bar{r}\bar{s})(p_1 + q_1 + \bar{r}_1)(r_1 + s + \bar{s}_1)\bar{p}\bar{q}\bar{q}_1\bar{p}_1s_1 \quad (3.8)$$

and see that the CNF of (3.7) will consist of 27 clauses. Here are three “typical” clauses from this CNF:

$$\begin{aligned} & (p \vee p_1 \vee r_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1) \\ & (\bar{r} \vee \bar{s} \vee q_1 \vee r_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1) \\ & (q \vee \bar{r}_1 \vee \bar{s}_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1). \end{aligned}$$

Each of these clauses contains a pair of literals which are mates: p, \bar{p} in the first (and also p_1, \bar{p}_1), q_1, \bar{q}_1 in the second; and q, \bar{q} in the third (also s_1, \bar{s}_1). The same will be true for the remaining 24 clauses. But this is clearly not the basis for a very efficient algorithm. What if we try Theorem 2.1(2) on the same example? Then we need to show that the following formula is unsatisfiable:

$$\begin{aligned} & ((p \wedge q) \supset (r \wedge s)) \wedge ((p_1 \wedge q_1) \supset r_1) \wedge ((r_1 \wedge s) \supset s_1) \\ & \quad \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1. \end{aligned} \tag{3.9}$$

Using (I) we obtain

$$\begin{aligned} & ((\neg(p \wedge q) \vee (r \wedge s)) \wedge (\neg(p_1 \wedge q_1) \vee r_1) \\ & \quad \wedge (\neg(r_1 \wedge s) \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1). \end{aligned}$$

Using (II) we obtain

$$\begin{aligned} & ((\neg p \vee \neg q \vee (r \wedge s)) \wedge (\neg p_1 \vee \neg q_1 \vee r_1) \\ & \quad \wedge (\neg r_1 \vee \neg s \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1). \end{aligned} \tag{3.10}$$

To find a DNF formula equal to this we replace \wedge by · and \vee by +, obtaining

$$(\bar{p} + \bar{q} + rs)(\bar{p}_1 + \bar{q}_1 + r_1)(\bar{r}_1 + \bar{s} + s_1)pqq_1p_1\bar{s}_1.$$

But this is exactly the same as (3.8) except that each literal has been replaced by its mate! Once again we face essentially the same 27 clauses.

Suppose we seek a formula in CNF equal to (3.10) instead of a formula in DNF. We need only replace \wedge by + and \vee by ·:

$$\bar{p}\bar{q}(r + s) + \bar{p}_1\bar{q}_1r_1 + \bar{r}_1\bar{s}s_1 + p + q + q_1 + p_1 + \bar{s}_1.$$

In this manner, we get a formula in which almost all “multiplying out” has already occurred. The CNF is simply

$$\begin{aligned} & (\bar{p} \vee \bar{q} \vee r) \wedge (\bar{p} \vee \bar{q} \vee s) \wedge (\bar{p}_1 \vee \bar{q}_1 \vee r_1) \\ & \quad \wedge (\bar{r}_1 \vee \bar{s} \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \bar{s}_1. \end{aligned} \tag{3.11}$$

It consists of 9 short, easily obtained clauses.

A moment's reflection will show that this situation is entirely typical. Because the formula of Theorem 2.1(2) has the form

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n \wedge \neg\gamma),$$

we can get a CNF formula simply by obtaining a CNF for each of the (ordinarily short) formulas $\gamma_1, \gamma_2, \dots, \gamma_n, \neg\gamma$. However, to obtain a DNF which according to Theorem 3.3 is what we really want, we will have to multiply out $(n + 1)$ polynomials. If, say, each of $\gamma_1, \dots, \gamma_n, \neg\gamma$ is an \vee -clause consisting of k literals, then the DNF will consist of k^{n+1} \wedge -clauses. And the general principle of duality guarantees (as we have already seen in our particular example) that the same discouraging arithmetic will emerge should we attempt instead to use Theorem 2.1(1). In this case a DNF will generally be easy to get, whereas a CNF (which is what we really want) will require a good deal of computing time.

These considerations lead to the following problem:

Satisfiability Problem: Find an efficient algorithm for testing an \mathcal{A} -formula in CNF to determine whether it is truth-functionally satisfiable.

This problem has been of central importance in theoretical computer science, not only for the reasons already given, but also for others which will emerge in Chapter 15.

Exercises

1. Find CNF and DNF formulas equal to each of the following:

- (i) $((p \wedge (q \vee r)) \vee (q \wedge (p \vee r)))$;
- (ii) $((\neg p \vee (p \wedge \neg q)) \wedge (r \vee (\neg p \wedge q)))$.

2. Find a DNF formula which has the truth table

p	q	r	
1	1	1	0
0	1	1	1
1	0	1	1
0	0	1	1
1	1	0	0
0	1	0	1
1	0	0	0
0	0	0	0

[Hint: The second row of the table corresponds to the \wedge -clause $(\neg p \wedge q \wedge r)$. Each row for which the value is 1 similarly determines an \wedge -clause.]

3. Show how to generalize Exercise 2 to obtain a DNF formula corresponding to any given truth table.

4. Use a normal form to show the correctness of the inference

$$(p \Rightarrow q), (r \vee \neg q), \neg(p \wedge r) \vDash \neg p.$$

4. The Davis–Putnam Rules

In order to make it easier to state algorithms for manipulating formulas in CNF, it will be helpful to give a simple representation of such formulas as sets. From now on we use the word *clause* to mean \vee -clause. We represent the clause $\kappa = \bigvee_{j \leq m} \lambda_j$ as the set $\kappa = \{\lambda_j \mid j \leq m\}$, and we represent the formula $\alpha = \bigwedge_{i \leq n} \kappa_i$, where each κ_i is a clause, as the set $\alpha = \{\kappa_i \mid i \leq n\}$. In so doing we lose the order of the clauses and the order of the literals in each clause; however, by the commutative laws, this does not matter.

It is helpful to speak of the empty set of literals as the *empty clause*, written \square , and of the empty set of clauses as the *empty formula*, written simply \emptyset . Since it is certainly true, although vacuously so, that there is an assignment (in fact any assignment will do) which makes every clause belonging to the empty formula true, it is natural and appropriate to agree that the *empty formula* \emptyset is *satisfiable* (in fact, it is a tautology). On the other hand, there is no assignment which makes some literal belonging to the empty clause \square true (because there are no such literals). Thus, we should regard the empty clause \square as being *unsatisfiable*. Hence any formula α such that $\square \in \alpha$ will be unsatisfiable as well.

We will give some rules for manipulating formulas in CNF which are helpful in designing algorithms for testing such formulas for satisfiability. By Theorem 3.1, a clause κ is a tautology if and only if $\lambda, \neg\lambda \in \kappa$ for some literal λ . Now, if $\kappa \in \alpha$ and κ is a tautologous clause, then α is satisfiable if and only if $\alpha - \{\kappa\}$ is. Hence, we can assume that the sets of clauses with which we deal contain no clauses which are tautologies. The following terminology is helpful: a clause $\kappa = \{\lambda\}$, consisting of a single literal, is called a *unit*. If α is a set of clauses and λ is a literal, then a clause κ is called λ -positive if $\lambda \in \kappa$, κ is called λ -negative if $\neg\lambda \in \kappa$, and κ is called λ -neutral if κ is neither λ -positive nor λ -negative. Since tautologous clauses have been excluded, no clause can be both λ -positive and λ -negative. We write α_λ^+ for the set of λ -positive clauses of α , α_λ^- for the set of λ -negative clauses of α , and α_λ^0 for the set of λ -neutral clauses of α . Thus for every literal λ , we have the decomposition $\alpha = \alpha_\lambda^+ \cup \alpha_\lambda^- \cup \alpha_\lambda^0$. Finally, we write

$$\text{POS}_\lambda(\alpha) = \alpha_\lambda^0 \cup \{\kappa - \{\lambda\} \mid \kappa \in \alpha_\lambda^+\},$$

$$\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0 \cup \{\kappa - \{\neg\lambda\} \mid \kappa \in \alpha_\lambda^-\}.$$

Our main result is

Theorem 4.1 (Splitting Rule). Let α be a formula in CNF, and let λ be a literal. Then α is satisfiable if and only if at least one of the pair $\text{POS}_\lambda(\alpha)$ and $\text{NEG}_\lambda(\alpha)$ is satisfiable.

Proof. First let α be satisfiable, say $\alpha^v = 1$. Thus $\kappa^v = 1$ for all $\kappa \in \alpha$. That is, for each $\kappa \in \alpha$, there is a literal $\mu \in \kappa$ such that $\mu^v = 1$. Now, we must have either $\lambda^v = 1$ or $\lambda^v = 0$. Suppose first that $\lambda^v = 0$. We know that for each $\kappa \in \alpha_\lambda^+$, there is a literal $\mu \in \kappa$ such that $\mu^v = 1$. Thus this μ is not λ . Thus, for $\kappa \in \alpha_\lambda^+$, $(\kappa - \{\lambda\})^v = 1$. Hence, in this case, $\text{POS}_\lambda(\alpha)^v = 1$. If, instead, $\lambda^v = 1$, we can argue similarly that for each $\kappa \in \alpha_\lambda^-$, $(\kappa - \{\neg\lambda\})^v = 1$ and hence that $\text{NEG}_\lambda(\alpha)^v = 1$.

Conversely, let $\text{POS}_\lambda(\alpha)^v = 1$ for some assignment v . Then we define the assignment w by stipulating that

$$\lambda^w = 0; \quad \mu^w = \mu^v \quad \text{for all literals } \mu \neq \lambda, \neg\lambda.$$

Now, if $\kappa \in \alpha_\lambda^0$, then $\kappa^w = \kappa^v = 1$; if $\kappa \in \alpha_\lambda^+$, then $\kappa^w = (\kappa - \{\lambda\})^w = (\kappa - \{\lambda\})^v = 1$; finally, if $\kappa \in \alpha_\lambda^-$, then $\kappa^w = 1$ because $(\neg\lambda)^w = 1$. Thus, $\alpha^w = 1$.

If, instead, $\text{NEG}_\lambda(\alpha)^v = 1$ for some assignment v , we define w by

$$\lambda^w = 1; \quad \mu^w = \mu^v \quad \text{for all literals } \mu \neq \lambda, \neg\lambda.$$

Then if $\kappa \in \alpha_\lambda^0$, we have $\kappa^w = \kappa^v = 1$; if $\kappa \in \alpha_\lambda^-$, then $\kappa^w = (\kappa - \{\neg\lambda\})^w = (\kappa - \{\neg\lambda\})^v = 1$; finally, if $\kappa \in \alpha_\lambda^+$, then $\kappa^w = 1$ because $\lambda^w = 1$. Thus again $\alpha^w = 1$. ■

This theorem has the virtue of eliminating one literal, but at the price of considering two formulas instead of one. For this reason, it is of particular interest to find special cases in which we do not need to consider both $\text{POS}_\lambda(\alpha)$ and $\text{NEG}_\lambda(\alpha)$.

Thus, suppose that $\alpha_\lambda^- = \emptyset$. Then $\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0 \subseteq \text{POS}_\lambda(\alpha)$. Hence, in this case, for any assignment v we have $\text{POS}_\lambda(\alpha)^v = 1$ implies $\text{NEG}_\lambda(\alpha)^v = 1$. Therefore, we conclude

Corollary 4.2 (Pure Literal Rule). If $\alpha_\lambda^- = \emptyset$, then α is satisfiable if and only if $\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0$ is satisfiable.

For another useful special case, suppose that the unit clause $\{\lambda\} \in \alpha$. Then, since $\{\lambda\} - \{\lambda\} = \square$, we conclude that $\square \in \text{POS}_\lambda(\alpha)$. Hence, $\text{POS}_\lambda(\alpha)$ is unsatisfiable, and we have

Corollary 4.3 (Unit Rule). If $\{\lambda\} \in \alpha$, then α is satisfiable if and only if $\text{NEG}_\lambda(\alpha)$ is satisfiable.

To illustrate this last corollary by an example, let α be (3.11), which is a CNF of (3.9). Using the set representation,

$$\alpha = \{\{\bar{p}, \bar{q}, r\}, \{\bar{p}, \bar{q}, s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{p\}, \{q\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}. \quad (4.1)$$

Thus, there are 9 clauses, of which 5 are units. Using the unit clause $\{p\}$, Corollary 4.3 tells us that α is satisfiable if and only if $\text{NEG}_p(\alpha)$ is. That is, we need to test for satisfiability the set of clauses

$$\{\{\bar{q}, r\}, \{\bar{q}, s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{q\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Using the unit rule again, this time choosing the unit clause $\{q\}$, we reduce to

$$\{\{r\}, \{s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Using the unit clause $\{s\}$, we get

$$\{\{r\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, s_1\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Successive uses of the unit clauses $\{q_1\}, \{p_1\}, \{\bar{s}_1\}$ yield

$$\begin{aligned} & \{\{r\}, \{\bar{p}_1, r_1\}, \{\bar{r}_1, s_1\}, \{p_1\}, \{\bar{s}_1\}\}; \\ & \{\{r\}, \{r_1\}, \{\bar{r}_1, s_1\}, \{\bar{s}_1\}\}; \\ & \{\{r\}, \{r_1\}, \{\bar{r}_1\}\}. \end{aligned}$$

This last, containing the unit clauses $\{r_1\}$ and $\{\bar{r}_1\}$, is clearly unsatisfiable. Or, alternatively, applying the unit rule one last time, we obtain

$$\text{NEG}_{r_1}(\{\{r\}, \{r_1\}, \{\bar{r}_1\}\}) = \{\{r\}, \square\},$$

which is unsatisfiable because it contains the empty clause \square .

So we have shown by this computation that (4.1), and therefore (3.9), is unsatisfiable. And by Theorem 2.1, we then can conclude (once again) that the tautological inference (2.1) is valid.

A slight variant of the above computation would begin by applying Corollary 4.2, the pure literal rule, to (4.1), using the literal r . This has the effect of simply deleting the first clause. The rest of the computation might then go as above, but with the initial clause deleted at each stage.

For another example, recall (3.6), which was obtained as a CNF of (3.1). Written as a set of clauses this becomes

$$\beta = \{\{p, q, \bar{r}, s\}, \{\bar{q}, \bar{p}, \bar{r}, s\}, \{\bar{q}, \bar{p}, \bar{r}\}\}. \quad (4.2)$$

Here the pure literal rule can be applied using either of the literals \bar{r}, s . Thus, we have that β is satisfiable if and only if $\beta_{\bar{r}}^0$ is satisfiable, if and only if β_s^0 is satisfiable. And we have

$$\beta_{\bar{r}}^0 = \emptyset, \quad \beta_s^0 = \{\{\bar{q}, \bar{p}, \bar{r}\}\}.$$

From the first we see at once that β is satisfiable; if we wish to use the second, we can note by inspection that $(\beta_s^0)^v = 1$, where $v(q) = v(p) = v(r) = 0$, or we can use the pure literal rule a second time (using any of the three available literals) and once again arrive at the empty formula \emptyset .

We next turn to an example that has no unit clauses and to which the pure literal rule is not applicable:

$$\alpha = \{\{\bar{q}, p\}, \{r, p\}, \{\bar{p}, \bar{q}\}, \{\bar{p}, s\}, \{q, \bar{r}\}, \{q, \bar{s}\}\}.$$

Thus we are led to use the splitting rule forming, say,

$$\text{POS}_p(\alpha) = \{\{\bar{q}\}, \{r\}, \{q, \bar{r}\}, \{q, \bar{s}\}\},$$

$$\text{NEG}_p(\alpha) = \{\{\bar{q}\}, \{s\}, \{q, \bar{r}\}, \{q, \bar{s}\}\}.$$

Applying the pure literal rule once and then the unit rule twice to $\text{POS}_p(\alpha)$, we obtain successively

$$\{\{\bar{q}\}, \{r\}, \{q, \bar{r}\}\}, \quad \{\{r\}, \{\bar{r}\}\}, \quad \{\square\},$$

so that $\text{POS}_p(\alpha)$ is unsatisfiable. Doing the same to $\text{NEG}_p(\alpha)$, we obtain successively

$$\{\{\bar{q}\}, \{s\}, \{q, \bar{s}\}\}, \quad \{\{s\}, \{\bar{s}\}\}, \quad \{\square\},$$

so that $\text{NEG}_p(\alpha)$ is likewise unsatisfiable. By Theorem 4.1, we can thus conclude that α is unsatisfiable.

These examples suggest a rather systematic recursive procedure (sometimes known as the Davis–Putnam procedure) for testing a given formula α in CNF for satisfiability. The procedure as we shall describe it will not be completely deterministic; there will be situations in which one of a number of literals is to be selected. We will write the recursive procedure using two variables, γ for a set of clauses and \mathcal{S} for a stack of sets of clauses. We write $\text{TOP}(\mathcal{S})$ for the set of clauses at the top of the stack \mathcal{S} , $\text{POP}(\mathcal{S})$ for \mathcal{S} after $\text{TOP}(\mathcal{S})$ has been removed, $\text{PUSH}(\beta, \mathcal{S})$ for the stack obtained by putting β on the top of \mathcal{S} , and \emptyset for the empty stack. The procedure is as follows:

```

 $\gamma \leftarrow \alpha; \mathcal{S} \leftarrow \emptyset;$ 
 $\text{while } \gamma \neq \emptyset \text{ and } (\square \notin \gamma \text{ or } \mathcal{S} \neq \emptyset)$ 
     $\text{if } \square \in \gamma$ 
         $\text{then } \gamma \leftarrow \text{TOP}(\mathcal{S}); \mathcal{S} \leftarrow \text{POP}(\mathcal{S});$ 
     $\text{else if } \gamma_{\lambda}^- = \emptyset$ 
         $\text{then } \gamma \leftarrow \gamma_{\lambda}^0;$ 
     $\text{else if } \{\lambda\} \in \gamma$ 
         $\text{then } \gamma \leftarrow \text{NEG}_{\lambda}(\gamma);$ 
     $\text{else } \gamma \leftarrow \text{POS}_{\lambda}(\gamma); \mathcal{S} \leftarrow \text{PUSH}(\text{NEG}_{\lambda}(\gamma), \mathcal{S});$ 
 $\text{end while}$ 
 $\text{if } \gamma = \emptyset \text{ then return SATISFIABLE}$ 
 $\text{else return UNSATISFIABLE}$ 

```

Thus, this procedure will terminate returning SATISFIABLE whenever γ is the empty formula \emptyset , whether or not the stack \mathcal{S} is empty. (This is all right because the original formula will be satisfiable if any one of the formulas obtained by repeated uses of the splitting rule is satisfiable, and, of course, \emptyset is satisfiable.) The procedure will terminate returning UNSATISFIABLE if $\square \in \gamma$ and $\mathcal{S} = \emptyset$. (Here, γ is unsatisfiable, and no formulas remain in \mathcal{S} as the result of uses of the splitting rule.) If neither of these termination conditions is satisfied, the algorithm will first test for $\square \in \gamma$. If $\square \in \gamma$, it rejects (since γ is unsatisfiable) and “pops” the stack. Otherwise it attempts to apply first the pure literal rule and then the unit rule. If both attempts fail, it chooses (nondeterministically) some literal λ , takes $\text{POS}_\lambda(\gamma)$ as the new formula to work on, and “pushes” $\text{NEG}_\lambda(\gamma)$ onto the stack for future reference.

It is not difficult to see that the algorithm just given must always terminate. Let us say that a set of clauses α reduces to a set of clauses β if for each clause κ in β there is a clause $\bar{\kappa}$ in α such that $\kappa \subseteq \bar{\kappa}$. Then, at the beginning of each pass through the *while* loop, γ is a set of clauses to which α reduces and the stack consists of a list of sets of clauses to each of which α reduces. Since, for a given α , there are only a finite number of distinct configurations of this kind, and none can be repeated, the algorithm must eventually terminate.

Exercises

1. Use the Davis–Putnam rules to show the correctness of the inference in Exercise 3.4.
2. Use the Davis–Putnam rules to show the correctness of the following inference:

If John went swimming, then he lost his glasses and did not go to the movies. If John ate too much meat and did not go to the movies, then he will suffer indigestion. Therefore, if John ate too much meat and went swimming, then he will suffer indigestion.

3. Test the following set of clauses for satisfiability:

$$\begin{aligned} & \{p, \bar{q}, r, s\} \\ & \{\bar{p}, q, \bar{r}\} \\ & \{\bar{r}, s\} \\ & \{\bar{q}, r\} \\ & \{p, \bar{s}\}. \end{aligned}$$

5. Minimal Unsatisfiability and Subsumption

We begin with

Theorem 5.1. Let the clauses κ_1, κ_2 satisfy $\kappa_1 \subset \kappa_2$. Then if α is a formula in CNF such that $\kappa_1, \kappa_2 \in \alpha$, then α is satisfiable if and only if $\alpha - \{\kappa_2\}$ is satisfiable.

Proof. Clearly, if α is satisfiable, so is $\alpha - \{\kappa_2\}$.

Conversely, if $(\alpha - \{\kappa_2\})^v = 1$, then $\kappa_1^v = 1$, so that also $\kappa_2^v = 1$. Hence, $\alpha^v = 1$. ■

Thus, if in fact $\kappa_1, \kappa_2 \in \alpha$ and $\kappa_1 \subset \kappa_2$, we may simply drop κ_2 and test $\alpha - \{\kappa_2\}$ for satisfiability. The operation of dropping κ_2 in such a case is called *subsumption*. Unfortunately, there is no efficient algorithm known for testing a large set of clauses for the possibility of applying *subsumption*.

Definition. A finite set of clauses α is called *minimally unsatisfiable* if

1. α is unsatisfiable, and
2. for all $\beta \subset \alpha$, β is satisfiable.

Definition. A finite set of clauses α is said to be *linked* if whenever $\lambda \in \kappa_1$ and $\kappa_1 \in \alpha$, there is a clause $\kappa_2 \in \alpha$ such that $\neg \lambda \in \kappa_2$. That is, each literal in a clause of α has a *mate* in another clause of α .

Then it is very easy to prove

Theorem 5.2. Let α be minimally unsatisfiable. Then

1. for no $\kappa_1, \kappa_2 \in \alpha$ can we have $\kappa_1 \subset \kappa_2$, and
2. α is linked.

Proof. (1) is an immediate consequence of Theorem 5.1. To verify (2), suppose that α is minimally unsatisfiable but not linked. Then, there is a literal λ in a clause $\kappa \in \alpha$ such that the literal $\neg \lambda$ occurs in none of the clauses of α , i.e., $\alpha_{\lambda}^- = \emptyset$. Thus, by the pure literal rule, α_{λ}^0 is unsatisfiable. But since $\alpha_{\lambda}^0 \subset \alpha$, this is a contradiction. ■

6. Resolution

Let κ_1, κ_2 be clauses such that $\lambda \in \kappa_1$ and $\neg \lambda \in \kappa_2$. Then we write

$$\text{res}_{\lambda}(\kappa_1, \kappa_2) = (\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg \lambda\}).$$

The clause $\text{res}_{\lambda}(\kappa_1, \kappa_2)$ is then called the *resolvent* of κ_1, κ_2 with respect to the literal λ . The operation of forming resolvents has been the basis of a very

large number of computer programs designed to perform logical deductions. We have

Theorem 6.1. Let λ be an atom and let κ_1, κ_2 be clauses such that $\lambda \in \kappa_1$, $\neg\lambda \in \kappa_2$. Then

$$\kappa_1, \kappa_2 \vDash \text{res}_\lambda(\kappa_1, \kappa_2).$$

Proof. Let v be an assignment such that $\kappa_1^v = \kappa_2^v = 1$. Now if $\lambda^v = 1$, then $(\kappa_2 - \{\neg\lambda\})^v = 1$, while if $\lambda^v = 0$, then $(\kappa_1 - \{\lambda\})^v = 1$. In either case, therefore, $\text{res}_\lambda(\kappa_1, \kappa_2)^v = 1$. ■

Let α be a finite set of clauses. A sequence of clauses $\kappa_1, \kappa_2, \dots, \kappa_n = \kappa$ is called a *resolution derivation of κ from α* if for each i , $1 \leq i \leq n$, either $\kappa_i \in \alpha$ or there are $j, k < i$ and a literal λ such that $\kappa_i = \text{res}_\lambda(\kappa_j, \kappa_k)$. A resolution derivation of \square from α is called a *resolution refutation of α* . We define

$$\text{RES}_\lambda(\alpha) = \alpha_\lambda^0 \cup \{\text{res}_\lambda(\kappa_1, \kappa_2) | \kappa_1 \in \alpha_\lambda^+, \kappa_2 \in \alpha_\lambda^-\}.$$

We have

Theorem 6.2. Let α be a formula in CNF and let λ be a literal. Then α is satisfiable if and only if $\text{RES}_\lambda(\alpha)$ is satisfiable.

Proof. First let $\alpha^v = 1$. Then if $\kappa \in \alpha_\lambda^0$, we have also $\kappa \in \alpha$, so that $\kappa^v = 1$. Furthermore, if $\kappa = \text{res}_\lambda(\kappa_1, \kappa_2)$, with $\kappa_1 \in \alpha_\lambda^+, \kappa_2 \in \alpha_\lambda^-$, then $\kappa_1^v = 1, \kappa_2^v = 1$, so that, by Theorem 6.1, $\kappa^v = 1$. Since for all $\kappa \in \text{RES}_\lambda(\alpha)$, we have $\kappa^v = 1$, it follows that $\text{RES}_\lambda(\alpha)^v = 1$.

Conversely, let $\text{RES}_\lambda(\alpha)^v = 1$. We claim that either $\text{POS}_\lambda(\alpha)^v = 1$ or $\text{NEG}_\lambda(\alpha)^v = 1$. For, suppose that $\text{POS}_\lambda(\alpha)^v = 0$. Since $\alpha_\lambda^0 \subseteq \text{RES}_\lambda(\alpha)$, we have $(\alpha_\lambda^0)^v = 1$. So for some $\kappa_1 \in \alpha_\lambda^+$, we must have $(\kappa_1 - \{\lambda\})^v = 0$. However, for all $\kappa_2 \in \alpha_\lambda^-$ and this κ_1 , we must have $\text{res}_\lambda(\kappa_1, \kappa_2)^v = [(\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg\lambda\})]^v = 1$. Thus, for all $\kappa_2 \in \alpha_\lambda^-$ we have $(\kappa_2 - \{\neg\lambda\})^v = 1$, i.e., $\text{NEG}_\lambda(\alpha)^v = 1$. This proves our claim that either $\text{POS}_\lambda(\alpha)$ or $\text{NEG}_\lambda(\alpha)$ must be satisfiable. By Theorem 4.1, i.e., the splitting rule, α is satisfiable. ■

Theorem 6.2 suggests another procedure for testing a formula α in CNF for satisfiability. As with the Davis–Putnam rules, seek a literal of α to which the pure literal or unit rule can be applied. If none is to be found, choose a literal λ of α and compute $\text{RES}_\lambda(\alpha)$. Continue recursively.

As with the Davis–Putnam procedure, this procedure must eventually terminate in $\{\square\}$ or \emptyset ; this is because the number of literals is successively diminished. This procedure has the advantage of not requiring a stack of formulas, but the disadvantage that the problem may get considerably larger because of the use of the $\text{RES}_\lambda(\alpha)$ operation. Unfortunately, the present pro-

cedure is also called the Davis–Putnam procedure in the literature. To add to the confusion, it seems that computer implementations of the “Davis–Putnam procedure” have been almost exclusively of the procedure introduced in Section 4, whereas theoretical analyses of the computational complexity of the “Davis–Putnam procedure” have tended to deal with the procedure we have just introduced.

Theorem 6.3. Let α be a formula in CNF and suppose that there is a resolution derivation of the clause κ from α . Then $\alpha^v = 1$ implies $\kappa^v = 1$.

Proof. Let $\kappa_1, \kappa_2, \dots, \kappa_n = \kappa$ be a resolution derivation of κ from α . We shall prove that $\kappa_i^v = 1$ for $1 \leq i \leq n$, which will prove the result. To prove this by induction, we assume that $\kappa_j^v = 1$ for all $j < i$. (Of course for the case $i = 1$, this induction hypothesis is true vacuously.) Now, there are two cases. If $\kappa_i \in \alpha$, then $\kappa_i^v = 1$. Otherwise $\kappa_i = \text{res}_\lambda(\kappa_j, \kappa_k)$, where $j, k < i$. Hence, by the induction hypothesis, $\kappa_j^v = \kappa_k^v = 1$. So by Theorem 6.1, $\kappa_i^v = 1$. ■

Theorem 6.4 (Ground Resolution Theorem). The formula α in CNF is unsatisfiable if and only if there is a resolution refutation of α .

Proof. First let there be a resolution refutation of α , but suppose that nevertheless $\alpha^v = 1$. Then, by Theorem 6.3, $\square^v = 1$, which is impossible.

Conversely, let α be unsatisfiable. Let $\lambda_1, \lambda_2, \dots, \lambda_k$ be a list of all the atoms that occur in α . Let

$$\alpha_0 = \alpha, \quad \alpha_i = \text{RES}_{\lambda_i}(\alpha_{i-1}), \quad i = 1, 2, \dots, k.$$

Clearly each α_i contains only the atoms λ_j for which $i < j \leq k$. Hence α_k contains no atoms at all, and must be either \emptyset or $\{\square\}$. On the other hand, by Theorem 6.2, we have that α_i is unsatisfiable for $0 \leq i \leq k$. Hence $\alpha_k = \{\square\}$. Now, let the sequence $\kappa_1, \kappa_2, \dots, \kappa_m$ of clauses consist, first, of all of the clauses of $\alpha_0 = \alpha$, then, all of the clauses of α_1 , and so on through all of the clauses of α_k . But this last means that $\kappa_m = \square$. Moreover, it is clear from the definition of the RES_λ operation that $\kappa_1, \kappa_2, \dots, \kappa_m$ is a resolution derivation. ■

To illustrate the ground resolution theorem, we apply it to (4.1) to show, once again, that (3.9) is unsatisfiable. Here then is a resolution refutation of the formula α of (4.1):

$$\begin{aligned} &\{\bar{p}, \bar{q}, s\}, \{\bar{r}_1, \bar{s}, s_1\}, \{\bar{p}, \bar{q}, \bar{r}_1, s_1\}, \{p\}, \{\bar{q}, \bar{r}_1, s_1\}, \{q\}, \{\bar{r}_1, s_1\}, \\ &\{\bar{s}_1\}, \{\bar{r}_1\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{p}_1, \bar{q}_1\}, \{q_1\}, \{\bar{p}_1\}, \{p_1\}, \square. \end{aligned}$$

Exercises

1. Give a resolution refutation which shows the correctness of the inference of Exercise 3.4.
2. The same for the inference of Exercise 4.2.

7. The Compactness Theorem

Now, we will prove a theorem relating infinite sets of \mathcal{A} -formulas to their finite subsets.

Definition. A set Ω of \mathcal{A} -formulas is called *finitely satisfiable* if for every finite set $\Delta \subseteq \Omega$, the set Δ is truth-functionally satisfiable.

We have

Theorem 7.1. Let Ω be finitely satisfiable and let α be an \mathcal{A} -formula. Then either $\Omega \cup \{\alpha\}$ or $\Omega \cup \{\neg\alpha\}$ is finitely satisfiable.

Proof. Suppose to the contrary that Ω is finitely satisfiable but that neither $\Omega \cup \{\alpha\}$ nor $\Omega \cup \{\neg\alpha\}$ is finitely satisfiable. Then there are finite sets $\Delta_1, \Delta_2 \subseteq \Omega$ such that $\Delta_1 \cup \{\alpha\}$ and $\Delta_2 \cup \{\neg\alpha\}$ are both truth-functionally unsatisfiable. But $\Delta_1 \cup \Delta_2$ is a finite subset of Ω and hence there must be an assignment v such that for each $\beta \in \Delta_1 \cup \Delta_2$, we have $\beta^v = 1$. Now, either $\alpha^v = 1$ or $\alpha^v = 0$. In the first case $\Delta_1 \cup \{\alpha\}$ is satisfiable, and in the second case $\Delta_2 \cup \{\neg\alpha\}$ is satisfiable. This is a contradiction. ■

Now we will need to use a general property of infinite languages.

Enumeration Principle. Let L be an infinite subset of A^* , where A is an alphabet (and therefore is finite). Then there is an infinite sequence or *enumeration* w_0, w_1, w_2, \dots which consists of all the words in L each listed exactly once.

The truth of this enumeration principle can be seen in many ways. One is simply to imagine the elements of L written in order of increasing length, and to order words of the same length among themselves like the entries in a dictionary. Alternatively, one can regard the strings on A as notations for numbers in some base (as in Chapter 5) and arrange the elements of L in numerical order. (Actually, as it is not difficult to see, these two methods yield the same enumeration.) Of course, no claim is made that there is an algorithm for computing w_i from i . Such an algorithm can only exist if the language L is r.e.

Now, let $\alpha_0, \alpha_1, \alpha_2, \dots$ be an enumeration of the set of all \mathcal{A} -formulas. (By the enumeration principle, such an enumeration must exist.) Let Ω be a given finitely satisfiable set of \mathcal{A} -formulas. We define the sequence

$$\begin{aligned}\Omega_0 &= \Omega \\ \Omega_{n+1} &= \begin{cases} \Omega_n \cup \{\alpha_n\} & \text{if this set is finitely satisfiable} \\ \Omega_n \cup \{\neg \alpha_n\} & \text{otherwise.} \end{cases}\end{aligned}$$

By Theorem 7.1, we have

Lemma 1. Each Ω_n is finitely satisfiable.

Let $\tilde{\Omega} = \bigcup_{n=0}^{\infty} \Omega_n$. Then, we have

Lemma 2. $\tilde{\Omega}$ is finitely satisfiable.

Proof. Let us be given a finite set $\Delta \subseteq \tilde{\Omega}$. For each $\gamma \in \Delta$, $\gamma \in \Omega_n$ for some n . Hence $\Delta \subseteq \Omega_m$, where m is the maximum of those n . By Lemma 1, Δ is truth-functionally satisfiable. ■

Lemma 3. For each \mathcal{A} -formula α either $\alpha \in \tilde{\Omega}$ or $\neg \alpha \in \tilde{\Omega}$, but not both.

Proof. Let $\alpha = \alpha_n$. Then $\alpha \in \Omega_{n+1}$ or $\neg \alpha \in \Omega_{n+1}$, so that α or $\neg \alpha$ belongs to $\tilde{\Omega}$. If $\alpha, \neg \alpha \in \tilde{\Omega}$, then by Lemma 2, the finite set $\{\alpha, \neg \alpha\}$ would have to be truth-functionally satisfiable. But this is impossible. ■

Now we define an assignment v by letting $v(\lambda) = 1$ if $\lambda \in \tilde{\Omega}$ and $v(\lambda) = 0$ if $\lambda \notin \tilde{\Omega}$ for every atom λ . We have

Lemma 4. For each \mathcal{A} -formula α , $\alpha^v = 1$ if and only if $\alpha \in \tilde{\Omega}$.

Proof. As we already know, it suffices to restrict ourselves to formulas using the connectives \neg, \vee, \wedge . And, in fact, the De Morgan relation

$$(\beta_1 \vee \beta_2) = \neg(\neg \beta_1 \wedge \neg \beta_2)$$

shows that we can restrict ourselves even further, to the connectives \neg, \wedge . So, we assume that α is an \mathcal{A} -formula expressed in terms of the connectives \neg, \wedge . Our proof will be by induction on the total number of occurrences of these connectives in α .

If this total number is 0, then α is an atom, and the result follows from our definition of v . Otherwise we must have either $\alpha = \neg \beta$ or $\alpha = (\beta \wedge \gamma)$, where by the induction hypothesis we can assume the desired result for β and γ .

Case 1. $\alpha = \neg\beta$

Then, using Lemma 3,

$$\begin{aligned}\alpha^v &= 1 && \text{if and only if } \beta^v \neq 1 \\ && & \text{if and only if } \beta \notin \tilde{\Omega} \\ && & \text{if and only if } \alpha \in \tilde{\Omega}.\end{aligned}$$

Case 2. $\alpha = (\beta \wedge \gamma)$

If $\alpha^v = 1$, then $\beta^v = \gamma^v = 1$, so by the induction hypothesis, $\beta, \gamma \in \tilde{\Omega}$. If $\alpha \notin \tilde{\Omega}$, then by Lemma 3, $\neg\alpha \in \tilde{\Omega}$. But the finite set $\{\beta, \gamma, \neg\alpha\}$ is not satisfiable, contradicting Lemma 2. Thus, $\alpha \in \tilde{\Omega}$.

Conversely, if $\alpha \in \tilde{\Omega}$, then neither $\neg\beta$ nor $\neg\gamma$ can belong to $\tilde{\Omega}$, because the finite sets $\{\alpha, \neg\beta\}$, $\{\alpha, \neg\gamma\}$ are not satisfiable. Thus, by Lemma 3, $\beta, \gamma \in \tilde{\Omega}$. By the induction hypothesis $\beta^v = \gamma^v = 1$. Therefore, $\alpha^v = 1$. ■

Now, since $\Omega \subseteq \tilde{\Omega}$, we see that $\alpha^v = 1$ for each $\alpha \in \Omega$. Hence, Ω is truth-functionally satisfiable. Since we began with an arbitrary finitely satisfiable set of \mathcal{A} -formulas Ω , we have proved

Theorem 7.2 (Compactness Theorem for Propositional Calculus). Let Ω be a finitely satisfiable set of \mathcal{A} -formulas. Then Ω is truth-functionally satisfiable.

Exercises

1. Is the set of clauses

$$\{(p_i \vee \neg p_{i+1}) | i = 1, 2, 3, \dots\}$$

satisfiable? Why?

2. The same for the set

$$\{(p_i \vee \neg p_{i+1}), (\neg p_i \vee p_{i+1}) | i = 1, 2, 3, \dots\}.$$

3.* Let us be given a plane map containing infinitely many countries. Suppose there is no way to color this map with k colors so that adjacent countries are colored with different colors. Prove that there is a finite submap for which the same is true.

Quantification Theory

1. The Language of Predicate Logic

Although a considerable part of logical inference is contained in the propositional calculus, it is only with the introduction of the apparatus of *quantifiers* that one can encompass the full scope of logical deduction as it occurs in mathematics, and in science generally. We begin with an alphabet called a *vocabulary* consisting of two kinds of symbols, *relation symbols* and *function symbols*. Let \mathbf{W} be a vocabulary. For each symbol $t \in \mathbf{W}$, we assume there is an integer $\delta(t)$ called the *degree* of t . For t a function symbol, $\delta(t) \geq 0$, while for t a relation symbol, $\delta(t) > 0$. A function symbol t whose degree is 0 is also called a *constant symbol*. We assume that \mathbf{W} contains at least one relation symbol. (What we are calling a *vocabulary* is often called a *language* in the literature of mathematical logic. Obviously this terminology is not suitable for a book on theoretical computer science.) In addition to \mathbf{W} we shall use the alphabet

$$Q = \{\neg, \wedge, \vee, \supset, \leftrightarrow, \forall, \exists, (,), x, y, z, u, v, w, |, ,\},$$

where the boldface comma $,$ is one of the symbols which belong to Q . The words which belong to the language

$$\{x^{[i]}, y^{[i]}, z^{[i]}, u^{[i]}, v^{[i]}, w^{[i]} \mid i \in N\},$$

are called *variables*. Again we think of strings of the form $|^{[i]}$, $i > 0$, as subscripts, e.g., writing x_5 for $x|^{[5]}$. By a \mathbf{W} -term (or when the vocabulary \mathbf{W} is understood, simply a *term*) we mean an element of $(Q \cup \mathbf{W})^*$ which either is a constant symbol $c \in \mathbf{W}$ or a variable, or is obtained from constant symbols

and variables by repeated application of the operation on $(Q \cup W)^*$ which transforms $\mu_1, \mu_2, \dots, \mu_n$ into

$$f(\mu_1, \mu_2, \dots, \mu_n),$$

where f is a function symbol in W and $\delta(f) = n > 0$.

An *atomic W-formula* is an element of $(Q \cup W)^*$ of the form

$$r(\mu_1, \mu_2, \dots, \mu_n),$$

where $r \in W$ is a relation symbol, $\delta(r) = n$, and $\mu_1, \mu_2, \dots, \mu_n$ are terms. Finally, a *W-formula* (or simply a *formula*) is either an atomic *W-formula* or is obtained from atomic *W-formulas* by repeated application of the following operations on $(Q \cup W)^*$:

1. transform α into $\neg\alpha$;
2. transform α and β into $(\alpha \wedge \beta)$;
3. transform α and β into $(\alpha \vee \beta)$;
4. transform α and β into $(\alpha \supset \beta)$;
5. transform α and β into $(\alpha \leftrightarrow \beta)$;
6. transform α into $(\forall b)\alpha$, where b is a variable;
7. transform α into $(\exists b)\alpha$, where b is a variable.

If b is a variable, the expressions

$$(\forall b) \quad \text{and} \quad (\exists b)$$

are called *universal quantifiers* and *existential quantifiers*, respectively.

Let b be a variable, let λ be a formula or a term, and suppose that we have the decomposition $\lambda = rbs$, where the leftmost symbol of s is not $|$. (This means that b is not part of a longer variable. In fact, because λ is a formula or a term, s will have to begin either with $,$ or with $)$.) Then we say that the variable b occurs in λ . If more than one such decomposition is possible for a given variable b we speak, in an obvious sense, of the *first occurrence of b in λ* , *the second occurrence of b in λ* , etc., reading from left to right.

Next suppose that α is a formula and that we have the decomposition

$$\alpha = r(\forall b)\beta s \quad \text{or} \quad \alpha = r(\exists b)\beta s,$$

where β is itself a formula. Then the occurrence of b in the quantifiers shown, as well as all occurrences of b in β , are called **bound** occurrences of b in α . Any occurrence of b in α which is not bound is called a **free occurrence** of b in α . A *W-formula* α containing no free occurrences of variables is called a *W-sentence*, or simply a *sentence*. Any occurrence of a variable in a term is considered to be a *free occurrence*.

Thus, in the formula

$$(r(x) \supset (\exists y)s(u, y)),$$

x and u each have one occurrence, and it is free; y has two occurrences, and they are both bound. The formula

$$(\forall x)(\exists u)(r(x) \supset (\exists y)s(u, y))$$

is a sentence.

2. Semantics

In analogy with the propositional calculus, we wish to associate the truth values, 1 and 0, with sentences. To do this for a given sentence α will require an “interpretation” of the function and relation symbols in α .

By an *interpretation I of a vocabulary W*, we mean a nonempty set D , called the *domain of I*, together with the following:

1. an element c_I of D , for each constant symbol $c \in W$;
2. a function f_I from $D^{\delta(f)}$ into¹ D , for each function symbol $f \in W$ for which $\delta(f) > 0$; and
3. a function r_I from $D^{\delta(r)}$ into $\{0, 1\}$, for each relation symbol $r \in W$.

Let λ be a *term* or a *formula* and let b_1, b_2, \dots, b_n be a *list of distinct variables which includes all the variables that have free occurrences in λ* . Then, we write $\lambda = \lambda(b_1, \dots, b_n)$ as a *declaration* of our intention to regard b_1, \dots, b_n as acting like parameters taking on values. In such a case, if t_1, \dots, t_n are terms containing no occurrences of variables that have bound occurrences in λ , we write $\lambda(t_1, \dots, t_n)$ for the term or formula obtained from λ by simultaneously replacing b_1 by t_1 , b_2 by t_2, \dots, b_n by t_n .

Now let t be a W -term, $t = t(b_1, b_2, \dots, b_n)$, and let I be an interpretation of W , with domain D . Then we shall define a value $t^I[d_1, d_2, \dots, d_n] \in D$ for all $d_1, d_2, \dots, d_n \in D$. For the case $n = 0$, we write simply t^I . We define this notion recursively as follows:

1. If $t = t(b_1, b_2, \dots, b_n)$ and t is a variable, then t must be b_i for some i , $1 \leq i \leq n$, and we define $t^I[d_1, d_2, \dots, d_n] = d_i$;
2. If $t = t(b_1, b_2, \dots, b_n)$ and t is a constant symbol c in W , then we define $t^I[d_1, d_2, \dots, d_n] = c_I$;
3. If $t = t(b_1, b_2, \dots, b_n) = g(t_1, t_2, \dots, t_m)$, where g is a function symbol in W , $\delta(g) = m > 0$, then we first set $t_i = t_i(b_1, b_2, \dots, b_n)$, $i = 1, 2, \dots, m$, and we let $s_i = t_i^I[d_1, d_2, \dots, d_n]$, $i = 1, 2, \dots, m$. Finally, we define

$$t^I[d_1, d_2, \dots, d_n] = g_I(s_1, s_2, \dots, s_m).$$

¹ Recall from Chapter 1, Section 1, that D^n is the set of n -tuples of elements of D .

Continuing, if α is a **W**-formula, $\alpha = \alpha(b_1, b_2, \dots, b_n)$, and I is an interpretation of **W** with domain D , we shall define a value $\alpha^I[d_1, d_2, \dots, d_n] \in \{0, 1\}$, for all $d_1, d_2, \dots, d_n \in D$. Again, in the particular case $n = 0$ (which can only happen if α is a sentence), we simply write α^I . The recursive definition is as follows:

(1) If $\alpha = \alpha(b_1, b_2, \dots, b_n) = r(t_1, t_2, \dots, t_m)$, where r is a relation symbol in **W**, $\delta(r) = m$, then we first set $t_i = t_i(b_1, b_2, \dots, b_n)$, $i = 1, 2, \dots, m$, and then let $s_i = t_i^I[d_1, d_2, \dots, d_n]$, $i = 1, 2, \dots, m$. Finally, we define $\alpha^I[d_1, d_2, \dots, d_n] = r_I(s_1, s_2, \dots, s_m)$.

In (2)–(6) below, let $\beta = \beta(b_1, \dots, b_n)$, $\gamma = \gamma(b_1, \dots, b_n)$, where we assume that $\beta^I[d_1, \dots, d_n] = k$, $\gamma^I[d_1, \dots, d_n] = l$ with $k, l \in \{0, 1\}$, are already defined for all $d_1, d_2, \dots, d_n \in D$:

(2) If α is $\neg\beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k = 1. \end{cases}$$

(3) If α is $(\beta \wedge \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = l = 1 \\ 0 & \text{otherwise.} \end{cases}$$

(4) If α is $(\beta \vee \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 0 & \text{if } k = l = 0 \\ 1 & \text{otherwise.} \end{cases}$$

(5) If α is $(\beta \supset \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 0 & \text{if } k = 1 \text{ and } l = 0 \\ 1 & \text{otherwise.} \end{cases}$$

(6) If α is $(\beta \leftrightarrow \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = l \\ 0 & \text{otherwise.} \end{cases}$$

In (7) and (8) below, let $\beta = \beta(b_1, \dots, b_n, b)$, where we assume that $\beta^I[d_1, \dots, d_n, e]$ is already defined for all $d_1, \dots, d_n, e \in D$:

(7) If α is $(\forall b)\beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } \beta^I[d_1, \dots, d_n, e] = 1 \text{ for all } e \in D \\ 0 & \text{otherwise.} \end{cases}$$

(8) If α is $(\exists b)\beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha^I[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } \beta^I[d_1, \dots, d_n, e] = 1 \text{ for some } e \in D \\ 0 & \text{otherwise.} \end{cases}$$

It is important to be aware of the entirely nonconstructive nature of (7) and (8) of this definition. When the set D is infinite, the definition provides no algorithm for carrying out the required searches, and, indeed, in many important cases no such algorithm exists.

Let us consider some simple examples:

EXAMPLE 1. $\mathbf{W} = \{c, r, s\}$, where c is a constant symbol, and r and s are relation symbols, $\delta(r) = 3$, $\delta(s) = 2$. Let I have the domain $D = \{0, 1, 2, 3, \dots\}$, let $c_I = 0$, and let

$$r_I(x, y, z) = \begin{cases} 1 & \text{if } x + y = z \\ 0 & \text{otherwise,} \end{cases} \quad s_I(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise.} \end{cases}$$

If α is the sentence

$$(\forall x)(\forall y)(\forall z)(r(x, y, z) \supset s(x, z)),$$

then it is easy to see that $\alpha^I = 1$. For if $u, v, w \in D$ and $r_I(u, v, w) = 1$, then $u + v = w$, so that $u \leq w$ and therefore $s_I(u, w) = 1$. So if $\gamma = \gamma(x, y, z)$ is the formula $(r(x, y, z) \supset s(x, z))$, then $\gamma^I[u, v, w] = 1$.

On the other hand, if β is the sentence

$$(\forall x)(\exists y)r(x, y, c),$$

then $\beta^I = 0$. This is because $r_I(1, v, 0) = 0$ for all $v \in D$. Therefore

$$r(x, y, c)^I[1, v] = 0$$

for all $v \in D$. Thus, $(\exists y)r(x, y, c)^I[1] = 0$, and therefore, finally, $\beta^I = 0$.

EXAMPLE 2. $\mathbf{W}, \alpha, \beta$ are as in Example 1. I has the domain

$$\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\},$$

the set of all integers. c_I, r_I, s_I are defined as in Example 1. In this case, it is easy to see that $\alpha^I = 0$ and $\beta^I = 1$.

An interpretation I of the vocabulary \mathbf{W} is called a *model* of a \mathbf{W} -sentence α if $\alpha^I = 1$; I is called a *model* of the set Ω of \mathbf{W} -sentences if I is a model of each $\alpha \in \Omega$. Ω is said to be *satisfiable* if it has at least one model. An individual \mathbf{W} -sentence α is called *satisfiable* if $\{\alpha\}$ is satisfiable, i.e., if α has a model. α is called *valid* if every interpretation of \mathbf{W} is a model of α .

If $\alpha = \alpha(b_1, \dots, b_n)$, $\beta = \beta(b_1, \dots, b_n)$ are \mathbf{W} -formulas, we write $\alpha = \beta$ to mean that α and β are *semantically equivalent*, that is, that

$$\alpha^I[d_1, \dots, d_n] = \beta^I[d_1, \dots, d_n]$$

for all interpretations I of \mathbf{W} and all $d_1, \dots, d_n \in D$, the domain of I . Then, as is readily verified, all of the equations from Section 1 of Chapter 11 hold true as well in the present context. We also note the *quantificational De Morgan laws*:

$$\neg(\forall b)\alpha = (\exists b)\neg\alpha; \quad \neg(\exists b)\alpha = (\forall b)\neg\alpha. \quad (2.1)$$

Again, as in the case of the propositional calculus, we may eliminate the connectives \supset and \leftrightarrow by using appropriate equations from Chapter 11, Section 1. Once again, there is a “general principle of duality,” but we omit the details.

Now, let $\beta = \beta(b_1, \dots, b_n, b)$, and let the variable a have no occurrences in β . Then it is quite obvious that

$$\begin{aligned} (\exists b)\beta(b_1, \dots, b_n, b) &= (\exists a)\beta(b_1, \dots, b_n, a), \\ (\forall b)\beta(b_1, \dots, b_n, b) &= (\forall a)\beta(b_1, \dots, b_n, a). \end{aligned} \quad (2.2)$$

Continuing to assume that a has no occurrences in β , we have

$$\begin{aligned} ((\forall a)\alpha \wedge \beta) &= (\forall a)(\alpha \wedge \beta), \\ ((\exists a)\alpha \wedge \beta) &= (\exists a)(\alpha \wedge \beta), \\ ((\forall a)\alpha \vee \beta) &= (\forall a)(\alpha \vee \beta), \\ ((\exists a)\alpha \vee \beta) &= (\exists a)(\alpha \vee \beta). \end{aligned} \quad (2.3)$$

Exercises

1. Let \mathbf{W} be as in Example 1. For each of the following \mathbf{W} -sentences give an interpretation which is a model of the sentence as well as one which is not:

- (a) $(\forall x)(\exists y)(\forall z)(s(x, c) \supset r(x, y, z))$,
- (b) $(\exists y)(\forall x)(\forall z)(s(x, c) \supset r(x, y, z))$,
- (c) $(\forall x)(\forall y)(s(x, y) \supset s(y, x))$.

2. Give an interpretation which is a model of (a) in Exercise 1 but not of (b).

3. For a set of sentences Ω , let $\text{Mod}(\Omega)$ be the collection of all models of Ω . Prove that

$$\Omega_1 \subseteq \Omega_2 \quad \text{implies} \quad \text{Mod}(\Omega_2) \subseteq \text{Mod}(\Omega_1).$$

3. Logical Consequence

We are now ready to use the *semantics* just developed to define the notion of *logical consequence*. Let \mathbf{W} be a vocabulary, let Γ be a set of \mathbf{W} -sentences, and let γ be a \mathbf{W} -sentence. Then we write

$$\Gamma \vDash \gamma$$

and call γ a *logical consequence* of the *premises* Γ if every model of Γ is also a model of γ . If $\Gamma = \{\gamma_1, \dots, \gamma_n\}$, then we omit the braces $\{ , \}$, and write simply

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma.$$

Note that $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$ if and only if for every interpretation I of \mathbf{W} for which

$$\gamma_1^I = \gamma_2^I = \dots = \gamma_n^I = 1,$$

we also have $\gamma^I = 1$. (Intuitively, we may think of the various interpretations as “possible worlds.”) Then our definition amounts to saying that γ is a logical consequence of some premises if γ is true in every possible world in which the premises are all true.) As in the case of the propositional calculus, logical consequence can be determined by considering a single sentence. The proof of the corresponding theorem is virtually identical to that of Theorem 2.1 in Chapter 11 and is omitted.

Theorem 3.1. The relation $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$ is equivalent to each of the following:

1. the sentence $((\gamma_1 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is valid;
2. the sentence $(\gamma_1 \wedge \dots \wedge \gamma_n \wedge \neg \gamma)$ is unsatisfiable.

Once again we are led to a problem of satisfiability. We will focus our efforts on computational methods for demonstrating the unsatisfiability of a given sentence. We begin by showing how to obtain a suitable normal form for any given sentence.

As in Chapter 11, Section 3, we begin with the procedures

- (I) ELIMINATE \supset and \leftrightarrow .
- (II) MOVE \neg INWARD.

(I) is carried out exactly as in Chapter 11. For (II), we also need to use the quantificational De Morgan laws (2.1). Ultimately all \neg s will come to immediately precede relation symbols.

(III) RENAME VARIABLES.

Rename bound variables as necessary to ensure that no variables occur in two different quantifiers, using (2.2). Thus, the sentence

$$(\forall x)(\forall y)r(x, y) \vee ((\forall x)s(x) \wedge (\exists y)s(y))$$

might become

$$(\forall x)(\forall y)r(x, y) \vee ((\forall u)s(u) \wedge (\exists v)s(v)).$$

(IV) PULL QUANTIFIERS.

Using (2.3), bring all quantifiers to the left of the sentence. *Where possible, do so with existential quantifiers preceding universal quantifiers.* Thus, to continue our example, we would get successively

$$\begin{aligned} & (\forall x)(\forall y)r(x, y) \vee (\exists v)((\forall u)s(u) \wedge s(v)) \\ &= (\exists v)((\forall x)(\forall y)r(x, y) \vee ((\forall u)s(u) \wedge s(v))) \\ &= (\exists v)(\forall x)(\forall y)(r(x, y) \vee (s(u) \wedge s(v))). \end{aligned}$$

After applying (IV) as many times as possible, we obtain a sentence consisting of *a string of quantifiers followed by a formula containing no quantifiers*. Such a sentence is called a *prenex sentence*. A prenex sentence is also said to be in *prenex normal form*.

Let γ be a sentence of the form

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha,$$

where $n \geq 0$, and $\alpha = \alpha(b_1, b_2, \dots, b_n, b)$. Let g be a function symbol which is not in α with $\delta(g) = n$. If necessary, we enlarge the vocabulary \mathbf{W} to include this new symbol g . Then we write

$$\gamma_s = (\forall b_1) \cdots (\forall b_n)\alpha(b_1, b_2, \dots, b_n, g(b_1, b_2, \dots, b_n)).$$

γ_s is called the *Skolemization* of γ . [In the case $n = 0$, g is a constant symbol and the term

$$g(b_1, b_2, \dots, b_n)$$

is to be simply understood as standing for g .] Skolemization is important because of the following theorem:

Theorem 3.2. Let γ be a \mathbf{W} -sentence and let γ_s be its Skolemization. Then

- (1) every model of γ_s is a model of γ ;
- (2) if γ has a model, then so does γ_s ;
- (3) γ is satisfiable if and only if γ_s is satisfiable.

Proof. (3) obviously follows from (1) and (2).

To prove (1), let α, γ be as above and let

$$\beta = \beta(b_1, \dots, b_n) = \alpha(b_1, \dots, b_n, g(b_1, \dots, b_n)).$$

Let I be a model of γ_s so that $\gamma_s^I = 1$, and let the domain of I be D . Then, if d_1, \dots, d_n are arbitrary elements of D , we have $\beta^I[d_1, \dots, d_n] = 1$. Let $e = g_I(d_1, \dots, d_n)$. Thus $\alpha^I[d_1, \dots, d_n, e] = 1$, so that

$$(\exists b)\alpha(b_1, \dots, b_n, b)^I[d_1, \dots, d_n] = 1.$$

Hence finally, $\gamma^I = 1$.

To prove (2), let $\gamma^I = 1$, where I has the domain D . Again let d_1, \dots, d_n be any elements of D . Then, writing β for the formula $(\exists b)\alpha$, so that we may write $\beta = \beta(b_1, \dots, b_n)$, we have $\beta^I[d_1, \dots, d_n] = 1$. Thus, there is an element $e \in D$ such that $\alpha^I[d_1, \dots, d_n, e] = 1$. Hence, we have shown that for each $d_1, \dots, d_n \in D$, there is at least one element $e \in D$ such that $\alpha^I[d_1, \dots, d_n, e] = 1$. Thus, we may extend the interpretation I to the new function symbol g by defining $g_I(d_1, \dots, d_n)$ to be such an element¹ e , for each $d_1, \dots, d_n \in D$. Thus, for all $d_1, d_2, \dots, d_n \in D$, we have

$$\beta^I[d_1, \dots, d_n] = \alpha^I[d_1, \dots, d_n, g_I(d_1, \dots, d_n)] = 1.$$

Hence finally, $\gamma_s^I = 1$. ■

Since Theorem 3.2 shows that the leftmost existential quantifier in a prenex formula may be eliminated without affecting satisfiability, we can, by iterated Skolemization, obtain a sentence containing no existential quantifiers. We write this

(V) ELIMINATE EXISTENTIAL QUANTIFIERS.

In the example discussed under (IV), this would yield simply

$$(\forall x)(\forall y)(\forall u)(r(x, y) \vee (s(u) \wedge s(c))), \quad (3.1)$$

where c is a constant symbol.

For another example consider the sentence

$$(\forall x)(\exists u)(\forall y)(\forall z)(\exists v)r(x, y, z, u, v),$$

where r is a relation symbol, $\delta(r) = 5$. Then two Skolemizations yield

$$(\forall x)(\forall y)(\forall z)r(x, y, z, g(x), h(x, y, z)). \quad (3.2)$$

A sentence α is called *universal* if it has the form $(\forall b_1)(\forall b_2) \cdots (\forall b_n)\gamma$, where the formula γ contains no quantifiers. We may summarize the procedure (I)–(V) in

Theorem 3.3. There is an algorithm which will transform any given sentence β into a universal sentence α such that β is satisfiable if and only if α is satisfiable. Moreover, any model of α is also a model of β .

In connection with our procedure (I)–(V) consider the example

$$((\forall x)(\exists y)r(x, y) \wedge (\forall u)(\exists v)s(u, v)),$$

¹ Here we are using a nonconstructive set-theoretic principle known as the *axiom of choice*.

where r and s are relation symbols. By varying the order in which the quantifiers are pulled, we can obtain the prenex sentences

1. $(\forall x)(\exists y)(\forall u)(\exists v)(r(x, y) \wedge s(u, v)),$
2. $(\forall u)(\exists v)(\forall x)(\exists y)(r(x, y) \wedge s(u, v)),$
3. $(\forall x)(\forall u)(\exists y)(\exists v)(r(x, y) \wedge s(u, v)).$

Skolemizations will then yield the corresponding universal sentences:

1. $(\forall x)(\forall u)(r(x, g(x)) \wedge s(u, h(x, u))),$
2. $(\forall u)(\forall x)(r(x, g(u, x)) \wedge s(u, h(u))),$
3. $(\forall x)(\forall u)(r(x, g(x, u)) \wedge s(u, h(x, u))).$

But, for this example, one would expect that y should “depend” only on x and v only on u . In other words, we would expect to be able to use a universal sentence such as

4. $(\forall x)(\forall u)(r(x, g(x)) \wedge s(u, h(u))).$

As we shall see, it is important to be able to justify such simplifications.

Proceeding generally, let γ be a sentence of the form

$$\delta \wedge (\forall b_1) \cdots (\forall b_n)(\exists b)\alpha,$$

where $n \geq 0$ and $\alpha = \alpha(b_1, b_2, \dots, b_n, b)$. Let g be a function symbol which does not occur in γ with $\delta(g) = n$. Then we write

$$\gamma_s = \delta \wedge (\forall b_1) \cdots (\forall b_n)\alpha(b_1, \dots, b_n, g(b_1, \dots, b_n)).$$

γ_s is called a *generalized Skolemization* of γ . Then we have the following generalization of Theorem 3.2:

Theorem 3.4. Let γ be a **W**-sentence and let γ_s be a generalized Skolemization of γ . Then we have (1)–(3) of Theorem 3.2.

Proof. Again we need only verify (1) and (2). Let α, γ, δ be as above. To prove (1), let I be a model of γ_s with domain D . Let β be defined as in the proof of Theorem 3.2. Then $\delta^I = 1$ and $(\forall b_1) \cdots (\forall b_n)\beta^I = 1$. As in the proof of Theorem 3.2, we conclude that

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha^I = 1,$$

and so $\gamma^I = 1$.

Conversely, let $\gamma^I = 1$, where I has domain D . Then $\delta^I = 1$ and

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha^I = 1.$$

Precisely as in the proof of Theorem 3.2, we can extend the interpretation I to the symbol g in such a way that

$$(\forall b_1) \cdots (\forall b_n)\beta^I = 1.$$

Hence, $\gamma_s^t = 1$. ■

Henceforth we will consider the steps (IV) PULL QUANTIFIERS and (V) ELIMINATE EXISTENTIAL QUANTIFIERS to permit the use of generalized Skolemizations. Moreover, as we have seen, Theorem 3.3 remains correct if the universal sentence is obtained using generalized Skolemizations.

Exercises

1. Consider the inference

$$\begin{aligned} (\forall x)(p(x) \supset (\forall y)(s(y, x) \supset u(x))), \quad & (\exists x)(p(x) \wedge (\exists y)(s(y, x) \wedge h(y, x))) \\ \models & (\exists x)(\exists y)(u(x) \wedge h(y, x) \wedge s(y, x)). \end{aligned}$$

(a) Find a universal sentence whose unsatisfiability is equivalent to the correctness of this inference. Can you do this so that Skolemization only introduces constant symbols?

(b) Using (a) show that the inference is correct.

2. (a) Using generalized Skolemization find a universal sentence whose unsatisfiability is equivalent to the correctness of the inference

$$(\exists x)(\forall y)r(x, y) \models (\forall y)(\exists x)r(x, y).$$

(b) Show that the inference is correct.

3. The same as Exercise 2(a) for the inference

$$\begin{aligned} (\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)((P(x, y, u) \wedge P(y, z, v) \wedge P(x, v, w)) \supset P(u, z, w)), \\ (\forall x)(\forall y)(\exists z)P(z, x, y), \quad (\forall x)(\forall y)(\exists z)P(x, z, y) \\ \models (\exists x)(\forall y)P(y, x, y). \end{aligned}$$

4. Herbrand's Theorem

We have seen that the problem of logical inference is reducible to the problem of satisfiability, which in turn is reducible to the problem of satisfiability of universal sentences. In this section, we will prove Herbrand's theorem, which can be used together with algorithms for truth-functional satisfiability (discussed in Chapter 11) to develop procedures for this purpose.

Let α be a *universal* \mathbf{W} -sentence for some vocabulary \mathbf{W} , where we assume that α contains all the symbols in \mathbf{W} . If α contains at least one constant symbol, we call the set of all constant symbols in α the *constant set* of α . If α contains no constant symbols, we let a be some new constant symbol, which we add to \mathbf{W} , and we call $\{a\}$ the *constant set* of α . Then the language which consists of all \mathbf{W} -terms containing no variables is called the *Herbrand universe* of α . The set \mathcal{A} of atomic \mathbf{W} -formulas containing no variables is called the *atom set* of α . We will work with the set of *propositional formulas over \mathcal{A}* , i.e., of \mathcal{A} -formulas in the sense of Chapter 11, Section 1. *Each of these \mathcal{A} -formulas is also a \mathbf{W} -sentence that contains no quantifiers.*

Returning to the universal sentence (3.1), we see that its constant set is $\{c\}$, its Herbrand universe is likewise $\{c\}$, and its atom set is $\{r(c, c), s(c)\}$.

Next, examining the universal sentence (3.2), its constant set is $\{a\}$, but its Herbrand universe is infinite:

$$\mathbf{H} = \{a, g(a), h(a, a, a), g(g(a)), g(h(a, a, a)), h(a, a, g(a)), \dots\}.$$

Its atom set is likewise infinite:

$$\mathcal{A} = \{r(t_1, t_2, t_3, t_4, t_5) | t_1, t_2, t_3, t_4, t_5 \in \mathbf{H}\}.$$

Theorem 4.1. Let $\zeta = \zeta(b_1, b_2, \dots, b_n)$ be a \mathbf{W} -formula containing no quantifiers, so that the sentence

$$\gamma = (\forall b_1)(\forall b_2) \cdots (\forall b_n)\zeta$$

is universal. Let \mathbf{H} be the Herbrand universe of γ and let \mathcal{A} be its atom set. Then, γ is satisfiable if and only if the set

$$\Omega = \{\zeta(t_1, t_2, \dots, t_n) | t_1, t_2, \dots, t_n \in \mathbf{H}\} \quad (4.1)$$

of \mathcal{A} -formulas is truth-functionally satisfiable.

Proof. First let γ be satisfiable, say, $\gamma^I = 1$, and let D be the domain of I . We now define an assignment v on \mathcal{A} . Let r be a relation symbol of \mathbf{W} , $\delta(r) = m$, so that $r(t_1, \dots, t_m) \in \mathcal{A}$ for all $t_1, \dots, t_m \in \mathbf{H}$. Then we define

$$v(r(t_1, \dots, t_m)) = r_I(t_1^I, \dots, t_m^I).$$

We have

Lemma 1. For all \mathcal{A} -formulas α , $\alpha^I = \alpha^v$.

Proof. As in Chapter 11, we may assume that α contains only the connectives \neg, \wedge . Proceeding by induction, we see that if α is an atom, the result is obvious from our definition of v . Thus, we may suppose that $\alpha = \neg\beta$ or $\alpha = (\beta \wedge \gamma)$, where the result is known for β or, for β and γ , respectively.

In the first case, we have

$$\begin{aligned}\alpha^I = 1 &\quad \text{if and only if} & \beta^I = 0 \\ &\quad \text{if and only if} & \beta^v = 0 \\ &\quad \text{if and only if} & \alpha^v = 1.\end{aligned}$$

Similarly, in the second case

$$\begin{aligned}\alpha^I = 1 &\quad \text{if and only if} & \beta^I = \gamma^I = 1 \\ &\quad \text{if and only if} & \beta^v = \gamma^v = 1 \\ &\quad \text{if and only if} & \alpha^v = 1.\end{aligned}\blacksquare$$

Returning to the proof of the theorem, we wish to show that for all $\alpha \in \Omega$, $\alpha^v = 1$. By Lemma 1, it will suffice to show that $\alpha^I = 1$ for $\alpha \in \Omega$. Now, since $\gamma^I = 1$, we have

$$\zeta^I[d_1, \dots, d_n] = 1 \quad \text{for all } d_1, \dots, d_n \in D.$$

But clearly, for $t_1, \dots, t_n \in \mathbf{H}$,

$$\zeta(t_1, \dots, t_n)^I = \zeta^I[t_1^I, \dots, t_n^I] = 1.$$

We conclude that Ω is truth-functionally satisfiable.

Conversely, let us be given an assignment v on \mathcal{A} such that $\alpha^v = 1$ for all $\alpha \in \Omega$. We shall use v to construct an interpretation I of \mathbf{W} . The domain of I is simply the Herbrand universe \mathbf{H} . Furthermore,

- (1) If $c \in \mathbf{W}$ is a constant symbol, then $c_I = c$. (That is, a constant symbol is interpreted as itself.)
- (2) If $f \in \mathbf{W}$ is a function symbol, $\delta(f) = n > 0$, and $t_1, t_2, \dots, t_n \in \mathbf{H}$, then

$$f_I(t_1, t_2, \dots, t_n) = f(t_1, t_2, \dots, t_n) \in \mathbf{H}.$$

(Note carefully the use of boldface.)

- (3) If $r \in \mathbf{W}$ is a relation symbol, $\delta(r) = n$, and $t_1, t_2, \dots, t_n \in \mathbf{H}$, then

$$r_I(t_1, t_2, \dots, t_n) = v(r(t_1, t_2, \dots, t_n)).$$

[Note that the assignment v is only used in (3).] We have

Lemma 2. For every $t \in \mathbf{H}$, $t^I = t$.

Proof. Immediate from (1) and (2). \blacksquare

Lemma 3. For every \mathbf{W} -formula $\alpha = \alpha(b_1, \dots, b_n)$ containing no quantifiers, and all $t_1, \dots, t_n \in \mathbf{H}$, we have

$$\alpha^I[t_1, \dots, t_n] = v(\alpha(t_1, \dots, t_n)).$$

Proof. If α is an atom, the result follows at once from (3) and Lemma 2. For the general case it now follows because the same recursive rules are used for the propositional connectives, whether we are evaluating interpretations or assignments. ■

Returning to the proof of the theorem, we wish to show that $\gamma^I = 1$. For this, recalling that \mathbf{H} is the domain of I , it suffices to show that

$$\zeta^I[t_1, \dots, t_n] = 1 \quad \text{for all } t_1, \dots, t_n \in \mathbf{H}.$$

By Lemma 3, this amounts to showing that

$$v(\zeta(t_1, \dots, t_n)) = 1 \quad \text{for all } t_1, \dots, t_n \in \mathbf{H}.$$

But this last is precisely what we have assumed about v . ■

The usefulness of the theorem we have just proved results from combining it with the compactness theorem (Theorem 7.2 in Chapter 11).

Theorem 4.2 (Herbrand's Theorem). Let $\zeta, \gamma, \mathbf{H}, \mathcal{A}$, and Ω be as in Theorem 4.1. Then γ is unsatisfiable if and only if there is a truth-functionally unsatisfiable \mathbf{W} -formula of the form $\bigwedge_{\beta \in \Sigma} \beta$ for some finite subset Σ of Ω .

Proof. If there is a truth-functionally unsatisfiable \mathcal{A} -formula $\bigwedge_{\beta \in \Sigma} \beta$, where $\Sigma \subseteq \Omega$, then for every assignment v on \mathcal{A} , there is some $\beta \in \Sigma$ such that $\beta^v = 0$. Hence Σ , and therefore also Ω , is not truth-functionally satisfiable; hence by Theorem 4.1, γ is unsatisfiable.

Conversely, if γ is unsatisfiable, then by Theorem 4.1, Ω is not truth-functionally satisfiable. Thus, by the compactness theorem (Theorem 7.2 in Chapter 11), Ω is not finitely satisfiable; i.e., there is a finite set $\Sigma \subseteq \Omega$ such that Σ is not truth-functionally satisfiable. Then, the sentence $\bigwedge_{\beta \in \Sigma} \beta$ is truth-functionally unsatisfiable. ■

This theorem leads at once to a family of procedures for demonstrating the unsatisfiability of a universal sentence γ . Write $\Omega = \bigcup_{n=0}^{\infty} \Sigma_n$, where $\Sigma_0 = \emptyset$, $\Sigma_n \subseteq \Sigma_{n+1}$, the Σ_n are all finite, and where there is an algorithm which transforms each Σ_n into Σ_{n+1} . (This can easily be managed, e.g., by simply writing the elements of Ω as an infinite sequence.) Then we have the procedure

```

 $n \leftarrow 0$ 
WHILE  $\bigwedge_{\beta \in \Sigma_n} \beta$  IS TRUTH-FUNCTIONALLY SATISFIABLE DO
   $n \leftarrow n + 1$ 
END

```

If γ is unsatisfiable, the procedure will eventually terminate; otherwise it will continue forever. The test for truth-functional satisfiability of $\bigwedge_{\beta \in \Sigma_n} \beta$ can be performed using the methods of Chapter 11, e.g., the Davis–Putnam rules. Using this discussion, we are able to conclude

Theorem 4.3. For every vocabulary \mathbf{W} the set of unsatisfiable sentences is recursively enumerable. Likewise the set of valid sentences is r.e.

Proof. Given a sentence α , we apply our algorithms to obtain a universal sentence γ which is satisfiable if and only if α is. We then apply the above procedure based on Herbrand's theorem. It will ultimately halt if and only if α is unsatisfiable. This procedure shows that the set of unsatisfiable sentences is r.e.

Since a sentence α is valid if and only if $\neg\alpha$ is unsatisfiable, the same procedure shows that the set of valid sentences is r.e. ■

One might have hoped that the set of unsatisfiable \mathbf{W} -sentences would in fact be recursive. But as we shall see later (Theorem 8.1), this is not the case. Thus, as we shall see, we *cannot hope* for an algorithm that, beginning with sentences $\gamma_1, \gamma_2, \dots, \gamma_n, \gamma$ as input, will return YES if $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$, and NO otherwise. The best we can hope for is a general procedure that will halt and return YES whenever the given logical inference is correct, but that may fail to terminate otherwise. And in fact, using Theorem 3.1 and an algorithm of the kind used in the proof of Theorem 4.3, we obtain just such a procedure.

Now let us consider what is involved in testing the truth-functional satisfiability of $\bigwedge_{\beta \in \Sigma} \beta$, where Σ is a finite subset of the set Ω defined in (4.1). If we wish to use the methods developed in Chapter 11, we need to obtain a CNF of $\bigwedge_{\beta \in \Sigma} \beta$. But, if for each $\beta \in \Sigma$, we have a CNF formula β^0 such that $\beta = \beta^0$, then $\bigwedge_{\beta \in \Sigma} \beta^0$ is clearly a CNF of $\bigwedge_{\beta \in \Sigma} \beta$. *This fact makes CNF useful in this context.*

In fact we can go further. We can apply the algorithms of Chapter 11, Section 3, to obtain CNF formulas directly for $\zeta = \zeta(b_1, \dots, b_n)$. When we do this we are in effect enlarging the set of formulas to which we apply the methods of Chapter 11, by allowing atoms which contain variables. Each formula can then be thought of as representing all of the \mathbf{W} -formulas obtained by replacing each variable by an element of the Herbrand universe \mathbf{H} . In this context formulas containing no variables are called *ground formulas*. We also speak of *ground literals*, *ground clauses*, etc.

If the CNF formula obtained in this manner from $\zeta(b_1, \dots, b_n)$ is given by the set of clauses

$$\{\kappa_i(b_1, \dots, b_n) \mid i = 1, 2, \dots, r\}, \quad (4.2)$$

then each $\beta \in \Sigma$ will have a CNF

$$\{\kappa_i(t_1, \dots, t_n) \mid i = 1, 2, \dots, r\},$$

where t_1, \dots, t_n are suitable elements of \mathbf{H} . Hence, there will be a CNF of $\bigwedge_{\beta \in \Sigma} \beta$ representable in the form

$$\{\kappa_i(t_1^j, \dots, t_n^j) \mid i = 1, \dots, r, j = 1, \dots, s\}, \quad (4.3)$$

where $t_1^j, \dots, t_n^j \in \mathbf{H}$, $j = 1, 2, \dots, s$. Thus, what we are seeking is an unsatisfiable set of clauses of the form (4.3). Of course, such a set can be unsatisfiable without being *minimally* unsatisfiable in the sense of Chapter 11, Section 5. In fact, there is no reason to expect a minimally unsatisfiable set of clauses which contains, say, $\kappa_1(t_1, \dots, t_n)$ to also contain $\kappa_2(t_1, \dots, t_n)$. Thus, we are led to treat the clauses in the set (4.2) independently of one another, seeking substitutions of elements of \mathbf{H} for the variables b_1, \dots, b_n so as to obtain a truth-functionally inconsistent set \mathbf{R} of clauses. Each of the clauses in (4.2) can give rise by substitution to one or more of the clauses of \mathbf{R} .

Let us consider some examples:

EXAMPLE 1. Consider this famous inference: *All men are mortal; Socrates is a man; therefore, Socrates is mortal.* An appropriate vocabulary would be $\{m, t, s\}$, where m, t are relation symbols of degree 1 (which we think of as standing for the properties of *being a man*, and of *being mortal*, respectively), and s is a constant symbol (which we think of as naming Socrates). The inference becomes

$$(\forall x)(m(x) \supset t(x)), m(s) \vDash t(s).$$

Thus, we wish to prove the unsatisfiability of the sentence

$$((\forall x)(m(x) \supset t(x)) \wedge m(s) \wedge \neg t(s)).$$

Going to prenex form, we see that no Skolemization is needed:

$$(\forall x)((\neg m(x) \vee t(x)) \wedge m(s) \wedge \neg t(s)).$$

The Herbrand universe is just $\{s\}$. In this simple case, Herbrand's theorem tells us that we have to prove the truth-functional unsatisfiability of

$$((\neg m(s) \vee t(s)) \wedge m(s) \wedge \neg t(s));$$

that is, we are led directly to a ground formula in CNF. Using the set representation of Chapter 11, Section 4, we are dealing with the set of clauses

$$\{\{\bar{m}(s), t(s)\}, \{m(s)\}, \{\bar{t}(s)\}\}.$$

Using the Davis–Putnam rules (or, in this case equivalently, resolution), we obtain successively

$$\{\{t(s)\}, \{\bar{t}(s)\}\}, \quad \text{and} \quad \{\square\};$$

hence the original inference was valid.

EXAMPLE 2. Another inference: *Every shark eats a tadpole; all large white fish are sharks; some large white fish live in deep water; any tadpole eaten by a deep water fish is miserable; therefore, some tadpoles are miserable.*

Our vocabulary is $\{s, b, t, r, m, e\}$, where all of these are relation symbols of degree 1, except e , which is a relation symbol of degree 2. $e(x, y)$ is to represent “ x eats y .” s stands for the property of *being a shark*, b of *being a large white fish*, t of *being a tadpole*, r of *living in deep water*, and m of *being miserable*. The inference translates as

$$\begin{aligned} & (\forall x)(s(x) \supset (\exists y)(t(y) \wedge e(x, y))), \\ & (\forall x)(b(x) \supset s(x)), \\ & (\exists x)(b(x) \wedge r(x)), \\ & (\forall x)(\forall y)((r(x) \wedge t(y) \wedge e(x, y)) \supset m(y)) \\ & \qquad \models (\exists y)(t(y) \wedge m(y)). \end{aligned}$$

Thus, we need to demonstrate the unsatisfiability of the sentence

$$\begin{aligned} & ((\forall x)(s(x) \supset (\exists y)(t(y) \wedge e(x, y)))) \\ & \wedge (\forall x)(b(x) \supset s(x)) \\ & \wedge (\exists x)(b(x) \wedge r(x)) \\ & \wedge (\forall x)(\forall y)((r(x) \wedge t(y) \wedge e(x, y)) \supset m(y)) \\ & \wedge \neg(\exists y)(t(y) \wedge m(y))). \end{aligned}$$

We proceed as follows:

I. ELIMINATE \supset :

$$\begin{aligned} & ((\forall x)(\neg s(x) \vee (\exists y)(t(y) \wedge e(x, y)))) \\ & \wedge (\forall x)(\neg b(x) \vee s(x)) \\ & \wedge (\exists x)(b(x) \wedge r(x)) \\ & \wedge (\forall x)(\forall y)(\neg(r(x) \wedge t(y) \wedge e(x, y)) \vee m(y)) \\ & \wedge \neg(\exists y)(t(y) \wedge m(y))). \end{aligned}$$

II. MOVE \neg INWARD:

$$\begin{aligned} & ((\forall x)(\neg s(x) \vee (\exists y)(t(y) \wedge e(x, y)))) \\ & \wedge (\forall x)(\neg b(x) \vee s(x)) \\ & \wedge (\exists x)(b(x) \wedge r(x)) \\ & \wedge (\forall x)(\forall y)(\neg r(x) \vee \neg t(y) \vee \neg e(x, y) \vee m(y)) \\ & \wedge (\forall y)(\neg t(y) \vee \neg m(y))). \end{aligned}$$

III. RENAME VARIABLES:

$$\begin{aligned}
 & ((\forall x)(\neg s(x) \vee (\exists y_1)(t(y_1) \wedge e(x, y_1)))) \\
 & \quad \wedge (\forall z)(\neg b(z) \vee s(z)) \\
 & \quad \wedge (\exists u)(b(u) \wedge r(u)) \\
 & \quad \wedge (\forall v)(\forall w)(\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\forall y)(\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

IV. PULL QUANTIFIERS (trying to pull existential quantifiers first):

$$\begin{aligned}
 & (\exists u)(\forall x)(\exists y_1)(\forall z)(\forall v)(\forall w)(\forall y) \\
 & \quad ((\neg s(x) \vee (t(y_1) \wedge e(x, y_1)))) \\
 & \quad \wedge (\neg b(z) \vee s(z)) \\
 & \quad \wedge b(u) \wedge r(u) \\
 & \quad \wedge (\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

V. ELIMINATE EXISTENTIAL QUANTIFIERS:

$$\begin{aligned}
 & (\forall x)(\forall z)(\forall v)(\forall w)(\forall y) \\
 & \quad ((\neg s(x) \vee (t(g(x)) \wedge e(x, g(x)))) \\
 & \quad \wedge (\neg b(z) \vee s(z)) \\
 & \quad \wedge b(c) \wedge r(c) \\
 & \quad \wedge (\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

Thus we are led to the clauses

$$\begin{aligned}
 & \{\bar{s}(x), t(g(x))\}, \\
 & \{\bar{s}(x), e(x, g(x))\}, \\
 & \{\bar{b}(z), s(z)\}, \\
 & \{b(c)\}, \\
 & \{r(c)\}, \\
 & \{\bar{r}(v), \bar{t}(w), \bar{e}(v, w), m(w)\}, \\
 & \{\bar{t}(y), \bar{m}(y)\}.
 \end{aligned}$$

The Herbrand universe is

$$H = \{c, g(c), g(g(c)), \dots\}.$$

To find substitutions for the variables in **H**, we have recourse to Theorem 5.2 (2) in Chapter 11. To search for a minimally unsatisfiable set of ground clauses, we should seek substitutions that will lead to every literal having a mate (in another clause). By inspection, we are led to the substitution

$$x = c, \quad z = c, \quad v = c, \quad w = g(c), \quad y = g(c).$$

We thus obtain the set of ground clauses

$$\begin{aligned} & \{\bar{s}(c), t(g(c))\}, \\ & \{\bar{s}(c), e(c, g(c))\}, \\ & \{\bar{b}(c), s(c)\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Although this set of clauses is linked, we must still test for satisfiability. Using the Davis–Putnam rules we obtain, first using the unit rule on $\{b(c)\}$,

$$\begin{aligned} & \{\bar{s}(c), t(g(c))\}, \\ & \{\bar{s}(c), e(c, g(c))\}, \\ & \{s(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Using the unit rule on $\{s(c)\}$ and then on $\{r(c)\}$ gives

$$\begin{aligned} & \{t(g(c))\}, \\ & \{e(c, g(c))\}, \\ & \{\bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Using the unit rule on $\{t(g(c))\}$ and then on $\{e(c, g(c))\}$ gives

$$\{m(g(c))\},$$

$$\{\bar{m}(g(c))\}.$$

Finally, we obtain the set of clauses consisting of the empty clause:

□.

In Examples 1 and 2 each clause of (4.2) gave rise to just one clause in the truth-functionally unsatisfiable set of clauses obtained. That is, we obtain a truth-functionally unsatisfiable set of clauses of the form (4.3) with $s = 1$. Our next example will be a little more complicated.

EXAMPLE 3. We consider the inference

$$\begin{aligned} & (\forall x)(\exists y)(r(x, y) \vee r(y, x)), \\ & (\forall x)(\forall y)(r(x, y) \supset r(y, y)) \\ & \quad \models (\exists z)r(z, z). \end{aligned}$$

Thus, we wish to demonstrate the unsatisfiability of the sentence

$$(\forall x)(\exists y)(r(x, y) \vee r(y, x)) \wedge (\forall x)(\forall y)(r(x, y) \supset r(y, y)) \wedge \neg(\exists z)r(z, z).$$

We proceed as follows:

I, II, III. ELIMINATE \supset ; MOVE \neg INWARD; RENAME VARIABLES:

$$(\forall x)(\exists y)(r(x, y) \vee r(y, x)) \wedge (\forall u)(\forall v)(\neg r(u, v) \vee r(v, v)) \wedge (\forall z)\neg r(z, z).$$

IV. PULL QUANTIFIERS:

$$(\forall x)(\exists y)(\forall u)(\forall v)(\forall z)((r(x, y) \vee r(y, x)) \wedge (\neg r(u, v) \vee r(v, v)) \wedge \neg r(z, z)).$$

V. ELIMINATE EXISTENTIAL QUANTIFIERS:

$$\begin{aligned} & (\forall x)(\forall u)(\forall v)(\forall z)((r(x, g(x)) \vee r(g(x), x)) \\ & \quad \wedge (\neg r(u, v) \vee r(v, v)) \wedge \neg r(z, z)). \end{aligned}$$

We thus obtain the set of clauses

$$\{r(x, g(x)), r(g(x), x)\},$$

$$\{\bar{r}(u, v), r(v, v)\},$$

$$\{\bar{r}(z, z)\}.$$

The Herbrand universe is

$$\mathbf{H} = \{a, g(a), g(g(a)), \dots\}.$$

How can we find a mate for $r(x, g(x))$? Not by using $\bar{r}(z, z)$ —whichever element $t \in \mathbf{H}$ we substitute for x , $r(x, g(x))$ will become $r(t, g(t))$, which cannot be obtained from $r(z, z)$ by replacing z by any element of \mathbf{H} . Thus the only potential mate for $r(x, g(x))$ is $\bar{r}(u, v)$. We tentatively set $u = x, v = g(x)$ so that the second clause becomes

$$\{\bar{r}(x, g(x)), r(g(x), g(x))\}.$$

But now, $\bar{r}(u, v)$ is also the only available potential mate for $r(g(x), x)$. Thus, we are led to also substitute $v = x, u = g(x)$ in the second clause, obtaining

$$\{\bar{r}(g(x), x), r(x, x)\}.$$

Both $r(g(x), g(x))$ and $r(x, x)$ can be matched with $\bar{r}(z, z)$ to produce mates. We thus arrive at the set of clauses

$$\begin{aligned} &\{r(x, g(x)), r(g(x), x)\}, \\ &\{\bar{r}(x, g(x)), r(g(x), g(x))\}, \\ &\{\bar{r}(g(x), x), r(x, x)\}, \\ &\{\bar{r}(x, x)\}, \\ &\{\bar{r}(g(x), g(x))\}. \end{aligned}$$

Now we can replace x by any element of \mathbf{H} to obtain a linked set of ground clauses. For example, we can set $x = a$; but any other substitution for x will do. Actually, it is just as easy to work with the nonground clauses as listed, since the propositional calculus processing is quite independent of which element of \mathbf{H} we substitute for x . In fact after four applications of the unit rule (or of resolution) we obtain \square , which shows that the original inference was correct.

Exercises

1. Describe the Herbrand universe of the universal sentence obtained in Exercise 3.1.
2. The same for Exercise 3.2.
3. The same for Exercise 3.3.

5. Unification

We continue our consideration of Example 3 of the previous section. Let us analyze what was involved in attempting to “mate” our literals. Suppose we want to mate $r(x, g(x))$ with $\bar{r}(z, z)$. The first step is to observe that both literals have the same relation symbol r , and that r is negated in one and only one of the two literals. Next we were led to the equations

$$x = z, \quad g(x) = z.$$

The first equation is easily satisfied by setting $x = z$. But then the second equation becomes $g(z) = z$, and clearly no substitution from the Herbrand universe can satisfy this equation. Thus, we were led to consider instead the pair of literals $r(x, g(x))$, $\bar{r}(u, v)$. The equations we need to solve are then

$$x = u, \quad g(x) = v.$$

Again we satisfy the first equation by letting $x = u$; the second equation becomes $g(u) = v$, which can be satisfied by letting $v = g(u)$. So the literals become $r(u, g(u))$ and $\bar{r}(u, g(u))$.

This example illustrates the so-called *unification algorithm* for finding substitutions which will transform given literals $r(\lambda_1, \dots, \lambda_n)$, $\bar{r}(\mu_1, \dots, \mu_n)$ into mates of one another. The procedure involves comparing two terms μ, λ and distinguishing four cases:

1. One of μ, λ (say, μ) is a variable and λ does not contain this variable. Then replace μ by λ throughout.
2. One of μ, λ (say, μ) is a variable, $\lambda \neq \mu$, but λ contains μ . Then report: NOT UNIFIABLE.
3. μ, λ both begin with function symbols, but not with the same function symbol. Again report: NOT UNIFIABLE.
4. μ, λ begin with the same function symbol, say

$$\mu = g(v_1, \dots, v_k), \quad \lambda = g(\eta_1, \dots, \eta_k).$$

Then use this same procedure recursively on the pairs

$$v_1 = \eta_1, \quad v_2 = \eta_2, \quad \dots, \quad v_k = \eta_k.$$

In applying the unification algorithm to

$$r(\lambda_1, \dots, \lambda_n), \quad \bar{r}(\mu_1, \dots, \mu_n),$$

we begin with the pairs of terms

$$\lambda_1 = \mu_1, \quad \lambda_2 = \mu_2, \quad \dots, \quad \lambda_n = \mu_n$$

and apply the above procedure to each. Naturally, substitutions called for by step 1 must be made in all of the terms before proceeding.

To see that the process always terminates, it is only necessary to note that whenever step 1 is applied, the total number of variables present decreases.

EXAMPLE. Let us attempt to unify

$$r(g(x), y, g(g(z))) \quad \text{with} \quad \bar{r}(u, g(u), g(v)).$$

We are led to the equations

$$g(x) = u, \quad y = g(u), \quad g(g(z)) = g(v).$$

The first equation leads to letting

$$u = g(x),$$

and the remaining equations then become

$$y = g(g(x)) \quad \text{and} \quad g(g(z)) = g(v).$$

The second is satisfied by letting

$$y = g(g(x)),$$

which does not affect the third equation. The third equation leads recursively to

$$g(z) = v,$$

which is satisfied by simply setting v equal to the left side of this equation. The final result is

$$r(g(x), g(g(x)), g(g(z))), \bar{r}(g(x), g(g(x)), g(g(z))).$$

Numerous systematic procedures for showing sentences to be unsatisfiable based on the unification algorithm have been studied. These procedures work directly with clauses containing variables and do not require that substitutions from the Herbrand universe actually be carried out. In particular, there are *linked conjunct procedures* which are based on searches for a linked set of clauses, followed by a test for truth-functional unsatisfiability. However, most computer implemented procedures have been based on *resolution*. In these procedures, when a pair of literals have been mated by an appropriate substitution, they are immediately eliminated by resolution. We illustrate the use of resolution on Examples 2 and 3 of the previous section.

Beginning with the clauses of Example 2, applying the unification algorithm to the pair of literals $s(z)$, $\bar{s}(x)$, and then using resolution, we get

$$\begin{aligned} & \{\bar{b}(x), t(g(x))\}, \\ & \{\bar{b}(x), e(x, g(x))\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(v), \bar{t}(w), \bar{e}(v, w), m(w)\}, \\ & \{\bar{t}(y), \bar{m}(y)\}. \end{aligned}$$

Next, unifying

$$e(x, g(x)) \quad \text{and} \quad \bar{e}(v, w)$$

and using resolution, we get

$$\begin{aligned} & \{\bar{b}(x), t(g(x))\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{b}(x), \bar{r}(x), \bar{t}(g(x)), m(g(x))\}, \\ & \{\bar{t}(y), \bar{m}(y)\}. \end{aligned}$$

Another stage of unification and resolution yields

$$\begin{aligned} & \{t(g(c))\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{t}(g(c)), m(g(c))\}, \\ & \{\bar{t}(y), \bar{m}(y)\}, \end{aligned}$$

and then

$$\begin{aligned} & \{r(c)\}, \\ & \{\bar{r}(c), m(g(c))\}, \\ & \{\bar{m}(g(c))\}. \end{aligned}$$

Finally, we get

$$\begin{aligned} & \{r(c)\}, \\ & \{\bar{r}(c)\}, \end{aligned}$$

and, then, to complete the proof,

□.

The combination of unification with resolution can be thought of as a single step constituting a kind of generalized resolution. Thus, resolution in the sense of Chapter 11, that is, resolution involving only ground clauses, will now be called *ground resolution*, while the unmodified word *resolution* will be used to represent this more general operation. In the ground case we used the notation $\text{res}_\lambda(\kappa_1, \kappa_2)$ for the *resolvent* of κ_1, κ_2 with respect to the literal λ , namely,

$$(\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg\lambda\}).$$

In the general case, let $\lambda \in \kappa_1, \neg\mu \in \kappa_2$, where the unification algorithm can be successfully applied to λ and $\neg\mu$. Thus, there are substitutions for the variables which yield new clauses $\tilde{\kappa}_1, \tilde{\kappa}_2$ such that if the substitutions transform λ into $\tilde{\lambda}$, they also transform $\neg\mu$ into $\neg\tilde{\lambda}$. Then we write

$$\text{res}_{\lambda, \mu}(\kappa_1, \kappa_2) = (\tilde{\kappa}_1 - \{\tilde{\lambda}\}) \cup (\tilde{\kappa}_2 - \{\neg\tilde{\lambda}\}).$$

Let α be a finite set of clauses. Then a sequence of clauses $\kappa_1, \kappa_2, \dots, \kappa_n$ is called a *resolution derivation* of $\kappa_n = \kappa$ from α if for each $i, 1 \leq i \leq n$, either $\kappa_i \in \alpha$ or there are $j, k < i$ and literals λ, μ such that $\kappa_i = \text{res}_{\lambda, \mu}(\kappa_j, \kappa_k)$. As in Chapter 11, a resolution derivation of \square from α is called a *resolution refutation* of α . The key theorem is

Theorem 5.1 (J. A. Robinson's General Resolution Theorem). Let $\zeta = \zeta(b_1, \dots, b_n)$ be a W-formula containing no quantifiers, and let ζ be in CNF. Let

$$\gamma = (\forall b_1)(\forall b_2) \cdots (\forall b_n)\zeta.$$

Then, the sentence γ is unsatisfiable if and only if there is a resolution refutation of the clauses of ζ .

We shall not prove this theorem here, but will content ourselves with showing how it applies to Example 3 of the previous section. The clauses were

1. $\{r(x, g(x)), r(g(x), x)\}$
2. $\{\bar{r}(u, v), r(v, v)\}$
3. $\{\bar{r}(z, z)\}$.

A resolution refutation is obtained as follows:

4. $\{r(g(x), x), r(g(x), g(x))\}$ (resolving 1 and 2);
5. $\{r(x, x), r(g(x), g(x))\}$ (resolving 2 and 4);
6. $\{r(g(x), g(x))\}$ (resolving 3 and 5);
7. \square (resolving 3 and 6).

Exercises

1. Prove the correctness of the inferences of Exercises 3.1–3.3 by obtaining minimally unsatisfiable sets of clauses.
2. Prove the correctness of the inferences of Exercises 3.1–3.3 by obtaining resolution refutations.
3. (a) Prove that the problem of the validity of the sentence

$$(\exists x)(\exists y)(\forall z)((r(x, y) \supset (r(y, z) \wedge r(z, z))) \\ \wedge ((r(x, y) \wedge s(x, y)) \supset (s(x, z) \wedge s(z, z))))$$

leads to the list of clauses

$$\{r(x, y)\}, \\ \{s(x, y), \bar{r}(y, h(x, y)), \bar{r}(h(x, y), h(x, y))\}, \\ \{\bar{r}(y, h(x, y)), \bar{r}(h(x, y), h(x, y)), \bar{s}(x, h(x, y)), \bar{s}(h(x, y), h(x, y))\}.$$

(Hint: Use Theorem 5.1 in Chapter 11.)

- (b) Prove the validity of the sentence in (a) by giving a resolution refutation.

6. Compactness and Countability

In this section we give two applications of the circle of ideas surrounding Herbrand's theorem which are extremely important in mathematical logic, although they have not so far been involved in the applications of logic to computer science.

Theorem 6.1 (Compactness Theorem for Predicate Logic). Let Ω be a set of W-sentences each finite subset of which is satisfiable. Then Ω is satisfiable.

Proof. If Ω is finite, there is nothing to prove. If Ω is infinite, we can use the enumeration principle from Chapter 11, Section 7, to obtain an enumeration $\beta_0, \beta_1, \beta_2, \dots$ of the elements of Ω . Let us write

$$\gamma_n = \bigwedge_{i \leq n} \beta_i, \quad n = 0, 1, 2, \dots$$

Let steps (I)–(V) of Section 3 be applied to each of $\beta_0, \beta_1, \beta_2, \dots$ to obtain universal sentences

$$\alpha_i = (\forall b_1^{(i)}) \cdots (\forall b_{m_i}^{(i)}) \zeta_i(b_1^{(i)}, \dots, b_{m_i}^{(i)}).$$

Then by Theorem 3.3, for each i , α_i is satisfiable if and only if β_i is satisfiable, and moreover any model of α_i is also a model of β_i . Now let us apply the same steps (I)–(V) to the sentence γ_n . We see that if we use generalized Skolemization we can do this in such a way that the universal sentence δ_n we obtain, corresponding to γ_n in the sense of Theorem 3.3, consists of universal quantifiers followed by the formula

$$\bigwedge_{i \leq n} \zeta_i.$$

Now, by hypothesis, each γ_n is satisfiable. Hence, by Theorem 3.3, so is each δ_n . For each n , let \mathbf{H}_n be the Herbrand universe of δ_n . Thus,

$$\mathbf{H}_0 \subseteq \mathbf{H}_1 \subseteq \mathbf{H}_2 \dots$$

Let $\mathbf{H} = \bigcup_{n \in N} \mathbf{H}_n$. By Theorem 4.1, the sets

$$\Sigma_n = \left\{ \bigwedge_{i \leq n} \zeta_i(t_1^{(i)}, \dots, t_{m_i}^{(i)}) \mid t_1^{(i)}, \dots, t_{m_i}^{(i)} \in \mathbf{H}_n, i = 0, 1, \dots, n \right\}$$

are truth-functionally satisfiable. We wish to show that the set

$$\Gamma = \{\zeta_i(t_1, \dots, t_{m_i}) \mid t_1, t_2, \dots \in \mathbf{H}\}$$

is itself truth-functionally satisfiable. By the compactness theorem for propositional calculus (Theorem 7.2 in Chapter 11) it suffices to prove this for every finite subset Δ of Γ . But for any finite subset Δ of Γ , there is a largest value of the subscript i which occurs, and all the t_j which occur are in some \mathbf{H}_k . Let l be the larger of this subscript k and this largest value of i . Then Δ is itself a subset of

$$\Lambda_l = \{\zeta_i(t_1, \dots, t_{m_i}) \mid t_1, t_2, \dots \in \mathbf{H}_l, 0 \leq i \leq l\}.$$

Moreover, since Σ_l is truth-functionally satisfiable, so is Λ_l , and therefore Δ . This shows that Γ is truth-functionally satisfiable.

Now, let \mathcal{A} be the set of all atoms which occur in the formulas that belong to Γ . Let v be an assignment on \mathcal{A} such that $\beta^v = 1$ for all $\beta \in \Gamma$. Then we use v to construct an interpretation I of \mathbf{W} with domain \mathbf{H} precisely as in the proof of Theorem 4.1. Then Lemmas 2 and 3 of that proof hold and precisely as in that case we have

$$\zeta_i^I[t_1, \dots, t_{m_i}] = 1 \quad \text{for all } t_1, \dots, t_{m_i} \in \mathbf{H} \text{ and } i \in N.$$

Hence, $\alpha_i^I = 1$ for all $i \in N$. Since any model of α_i is also a model of β_i , we have $\beta_i^I = 1$ for all $i \in N$. Thus, I is a model of Ω . ■

Now let us begin with a set Ω of \mathbf{W} -sentences which has a model I . Then of course I is a model of every finite subset of Ω . Thus, the *method of proof*

of the previous theorem can be applied to Ω . Of course, this would be pointless if our aim were merely to obtain a model of Ω ; we already have a model I of Ω . But the method of proof of Theorem 6.1 gives us a model of Ω whose domain \mathbf{H} is a language on an alphabet. Thus, we have proved

Theorem 6.2 (Skolem–Löwenheim Theorem). Let Ω be a satisfiable set of \mathbf{W} -sentences. Then Ω has a model whose domain is a language on some alphabet.

What makes this important and interesting is that any language satisfies the enumeration principle of Chapter 11, Section 7. Infinite sets which possess an enumeration are called *countably infinite*. This brings us to the usual form of the Skolem–Löwenheim theorem:

Corollary 6.3. Let Ω be a satisfiable set of \mathbf{W} -sentences. Then Ω has a model whose domain is countably infinite.

Many infinite sets which occur in mathematics are *not* countable. In fact, the diagonal method, which was used in obtaining unsolvability results in Part 1 of this book, was originally developed by Cantor to prove that the set of real numbers is not countable. What the Skolem–Löwenheim theorem shows is that no set of sentences can characterize an infinite uncountable set in the sense of excluding countable models.

We close this section with another useful form of the compactness theorem.

Theorem 6.4. If $\Gamma \vDash \gamma$, then there is a finite subset Δ of Γ such that $\Delta \vDash \gamma$.

Proof. Since every model of Γ is a model of γ , the set $\Gamma \cup \{\neg\gamma\}$ has no models; that is, it is not satisfiable. Thus, by Theorem 6.1, there is a finite subset Δ of Γ such that $\Delta \cup \{\neg\gamma\}$ is unsatisfiable. Thus every model of Δ is a model of γ , i.e., $\Delta \vDash \gamma$. ■

Exercise

- Let Ω_1, Ω_2 be sets of sentences such that $\Omega_1 \cup \Omega_2$ is unsatisfiable. Prove that there is a sentence α such that $\Omega_1 \vDash \alpha$, and $\Omega_2 \vDash \neg\alpha$.

*7. Gödel's Incompleteness Theorem

Let Γ be a recursive set of \mathbf{W} -sentences for some given vocabulary \mathbf{W} . We think of Γ as being considered for use as a set of “axioms” for some part of mathematics. The requirement that Γ be recursive is natural, because,

by Church's Thesis, it simply amounts to requiring that there be some algorithmic method of determining whether or not an alleged "axiom" really is one. Often Γ will be finite. We define $T_\Gamma = \{\gamma \mid \Gamma \vdash \gamma\}$ and call T_Γ the *axiomatizable theory* on \mathbf{W} whose axioms are the sentences belonging to the set Γ . Of course, it is quite possible to have different sets of axioms which define the same theory.

If T is an axiomatizable theory, we write

$$\vdash_T \gamma$$

(read: " T proves γ ") to mean that $\gamma \in T$. We also write $\not\vdash_T \gamma$ to mean that $\gamma \notin T$. The most important fact about axiomatizable theories is given by the following theorem:

Theorem 7.1. An axiomatizable theory is r.e.

Proof. By Theorems 3.1 and 6.4, $\gamma \in T_\Gamma$ if and only if

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n \wedge \neg \gamma)$$

is unsatisfiable for some $\gamma_1, \gamma_2, \dots, \gamma_n \in \Gamma$. Since Γ is recursive, it is certainly r.e. Thus, by Theorem 7.3 in Chapter 7, there is a recursive function g on N whose range is Γ . For a given sentence γ , let

$$\delta(n, \gamma) = (g(0) \wedge g(1) \wedge \cdots \wedge g(n) \wedge \neg \gamma)$$

for all $n \in N$. Clearly, $\delta(n, \gamma)$ is a recursive function of n and γ . Moreover, the sentence γ belongs to T_Γ if and only if there is an $n \in N$ such that $\delta(n, \gamma)$ is unsatisfiable. But by Theorem 4.3, the set of unsatisfiable \mathbf{W} -sentences is r.e. Hence there is a partially computable function h which is defined for a given input if and only if that input is an unsatisfiable \mathbf{W} -sentence. Let h be computed by program \mathcal{P} and let $p = \#(\mathcal{P})$. Then the following "dovetailing" program halts if and only if the input γ belongs to T_Γ , thereby showing that T_Γ is r.e.:

```
[A]      Z ← δ(T, γ)
          T ← T + 1
          IF ~ STP(1)(Z, p, T) GOTO A
```

We shall see in the next section that there is a Γ such that T_Γ is not recursive.

Now let \mathbf{W} be some vocabulary intended for use in expressing properties of the natural numbers. By a *numeral system* for \mathbf{W} , we mean a *recursive function* v on N such that for each $n \in N$, $v(n)$ is a \mathbf{W} -term containing no variables, and such that for all $n, m \in N$, $n \neq m$ implies $v(n) \neq v(m)$. When v can be understood from the context, we write \bar{n} for $v(n)$. \bar{n} is called the

numeral corresponding to n and may be thought of as a notation for n using the vocabulary \mathbf{W} . A popular choice is

$$\bar{n} = S(S(\cdots S(0) \cdots)),$$

where S is a function symbol of degree 1, 0 is a constant symbol, and the number of occurrences of S is n .

Let $\alpha = \alpha(b)$ be a \mathbf{W} -formula and let \mathbf{T} be an axiomatic theory on \mathbf{W} . Then, given a numeral system for \mathbf{W} , we can associate with α the set

$$U = \{n \in N \mid \vdash_{\mathbf{T}} \alpha(\bar{n})\}. \quad (7.1)$$

In this case, we say that the formula α *represents* the set U in \mathbf{T} . If we begin with a set $U \subseteq N$, we can ask the question: Is there a \mathbf{W} -formula α which represents U in \mathbf{T} ? We have

Theorem 7.2. If there is a formula α which represents the set U in an axiomatizable theory \mathbf{T} , then U is r.e.

Proof. Let \mathbf{T} be an axiomatizable theory, and let α represent U in \mathbf{T} . By Theorem 7.1, we know that there is a program \mathcal{P} which will halt for given input y if and only if $\vdash_{\mathbf{T}} y$. Given $n \in N$, we need only compute $\alpha(\bar{n})$ [which we can do because $v(n) = \bar{n}$ is recursive], and feed it as input to \mathcal{P} . The new program thus defined halts for given input $n \in N$ if and only if $\vdash_{\mathbf{T}} \alpha(\bar{n})$. By (7.1), U is r.e. ■

In fact, there are many axiomatizable theories in which all r.e. sets are representable. To see the negative force of Theorem 7.2, we rewrite it as follows:

Corollary 7.3. Let \mathbf{T} be an axiomatizable theory. Then if $U \subseteq N$ is not r.e., there is no formula which represents U in \mathbf{T} .

This corollary is a form of Gödel's incompleteness theorem. To obtain a more striking form of the theorem, let us say that the formula α *quasi-represents* the set U in \mathbf{T} if

$$\{n \in N \mid \vdash_{\mathbf{T}} \alpha(\bar{n})\} \subseteq U. \quad (7.2)$$

We can think of such a formula α as intended to express the proposition " $n \in U$ " using the vocabulary \mathbf{W} . Comparing (7.1) and (7.2) and considering Corollary 7.3, we have

Corollary 7.4. Let \mathbf{T} be an axiomatizable theory and let $U \subseteq N$ be a set which is not r.e. Let the formula α quasi-represent U in \mathbf{T} . Then, there is a number n_0 such that $n_0 \in U$ but $\not\vdash_{\mathbf{T}} \alpha(\bar{n}_0)$.

As we can say loosely, the sentence $\alpha(\bar{n}_0)$ is "true" but not provable. Corollary 7.4 is another form of Gödel's incompleteness theorem. We conclude with our final version:

Theorem 7.5. Let T be an axiomatizable theory, and let S be an r.e. set which is not recursive. Let $\alpha = \alpha(x)$ be a formula such that α represents S in T , and $\neg\alpha$ quasi-represents \bar{S} in T . Then there is a number n_0 such that $\vdash_T \alpha(\bar{n}_0)$ and $\vdash_T \neg\alpha(\bar{n}_0)$.

Proof. We take $U = \bar{S}$ in Corollary 7.4 to obtain a number n_0 such that $n_0 \in \bar{S}$, but $\vdash_T \neg\alpha(\bar{n}_0)$. Since $n_0 \notin S$ and α represents S in T , we must also have $\vdash_T \alpha(\bar{n}_0)$. ■

In this last case, it is usual to say that $\alpha(\bar{n}_0)$ is undecidable in T .

*8. Unsolvability of the Satisfiability Problem in Predicate Logic

In 1928, the great mathematician David Hilbert called the problem of finding an algorithm for testing a given sentence to determine whether it is satisfiable “the main problem of mathematical logic.” This was because experience had shown that all of the inferences in mathematics could be expressed within the logic of quantifiers. Thus, an algorithm meeting Hilbert’s requirements would have provided, in principle, algorithmic solutions to all the problems in mathematics. So, when unsolvable problems were discovered in the 1930s, it was only to be expected that Hilbert’s satisfiability problem would also turn out to be unsolvable.

Theorem 8.1 (Church–Turing). There is a vocabulary W such that there is no algorithm for testing a given W -sentence to determine whether it is satisfiable.

Proof. Our plan will be to translate the word problem for a Thue process into predicate logic in such a way that a solution to Hilbert’s satisfiability problem would also yield a solution to the word problem for the given process.

Thus, using Theorem 3.5 in Chapter 7, let Π be a Thue process on the alphabet $\{a, b\}$ with an unsolvable word problem. Let Π have the productions $g_i \rightarrow h_i$, $i = 1, 2, \dots, K$, together with their inverses, where we may assume that for each i , $g_i, h_i \neq 0$ (recall Theorem 3.5 in Chapter 7). We introduce the vocabulary $W = \{a, b, \bullet, \cong\}$, where a, b are constant symbols, \bullet is a function symbol, and \cong is a relation symbol, with $\delta(\bullet) = \delta(\cong) = 2$. We will make use of the interpretation I with domain $\{a, b\}^* - \{0\}$ which is defined as follows:

$$a_I = a,$$

$$b_I = b,$$

$$\bullet_I(u, v) = uv,$$

$$\cong_I(u, v) = 1 \quad \text{if and only if } u \xrightarrow[\Pi]{} v.$$

For ease of reading, we shall write \bullet and \cong in “infix” position. Thus, we shall write, for example,

$$((x \bullet a) \cong y) \quad \text{instead of} \quad \cong (\bullet(x, a), y).$$

For each word $w \in \{a, b\}^* - \{0\}$, we now define a **W-term** w^* as follows:

$$\begin{aligned} a^* &= a, & b^* &= b, \\ (ua)^* &= (u^* \bullet a), & (ub)^* &= (u^* \bullet b). \end{aligned} \tag{8.1}$$

We have

Lemma 1. For every word $w \in \{a, b\}^* - \{0\}$, we have $(w^*)^I = w$.

Proof. The proof is by an easy induction on $|w|$, using (8.1) and the definition of the interpretation I . ■

Let Γ be the set of **W-sentences** obtained by prefixing the appropriate universal quantifiers to each **W-formula** in the following list:

- (1) $(x \cong x)$,
- (2) $((x \cong y) \supset (y \cong x))$,
- (3) $((x \cong y) \wedge (y \cong z)) \supset (x \cong z))$,
- (4) $((x \cong y) \wedge (u \cong v)) \supset ((x \bullet u) \cong (y \bullet v))$,
- (5) $((x \bullet y) \bullet z) \cong (x \bullet (y \bullet z))$,
- $(5+i)$ $(g_i^* \cong h_i^*)$, $1 \leq i \leq K$.

We have

Lemma 2. The interpretation I is a model of the set of sentences Γ .

Proof. The sentences of Γ all express in logical notation basic facts about concatenation of strings and about derivations in Thue processes. Detailed verification is left to the reader. ■

Lemma 3. If $\Gamma \vDash (u^* \cong v^*)$, then $u \xrightarrow{\Pi} v$.

Proof. By the definition of logical inference and Lemma 2, we have $(u^* \cong v^*)^I = 1$. Hence,

$$u = (u^*)^I \xrightarrow{\Pi} (v^*)^I = v. \quad \blacksquare$$

We next wish to establish the converse of Lemma 3. For this it will suffice to show that if $u \xrightarrow{\Pi} v$, then the sentence

$$\bigwedge_{\alpha \in \Gamma} \alpha \wedge \neg(u^* \cong v^*)$$

is unsatisfiable (recall Theorem 3.1). The Herbrand universe is

$$H = \{a, b, a \bullet a, a \bullet b, b \bullet a, b \bullet b, a \bullet (a \bullet a), \dots\}.$$

Let us call a W-sentence α a *Herbrand instance* of a W-formula β if α can be obtained from β by replacing each of its free variables by an element of H. α is said to be *rooted* if it is a tautological consequence of the sentences (5 + i) together with Herbrand instances of the formulas listed in (1)–(5) above. Obviously, if the sentence β is rooted, then $\Gamma \models \beta$.

Lemma 4. If $w = uv$, where $u \neq 0$ and $v \neq 0$, then

$$(w^* \cong (u^* \bullet v^*)) \quad (8.2)$$

is rooted.

Proof. The proof is by induction on $|v|$. If $|v| = 1$, we can assume without loss of generality that $v = a$. But in this case, the sentence (8.2) is a Herbrand instance of (1).

Supposing the result known for v , we need to establish it for va and vb . We give the proof for va , that for vb being similar. So let $w = uv$, where we can assume that (8.2) is rooted. We need to show that the sentence

$$((wa)^* \cong (u^* \bullet (va)^*))$$

is likewise rooted. By (8.1) this amounts to showing that

$$((w^* \bullet a) \cong (u^* \bullet (v^* \bullet a)))$$

is rooted. But this follows from the induction hypothesis, noting that the following sentences are rooted: (For each of these sentences, the number of the corresponding formula of which it is a Herbrand instance is given.)

$$(a \cong a) \quad (1)$$

$$(((w^* \cong (u^* \bullet v^*)) \wedge (a \cong a)) \supset ((w^* \bullet a) \cong ((u^* \bullet v^*) \bullet a))) \quad (4)$$

$$(((u^* \bullet v^*) \bullet a) \cong (u^* \bullet (v^* \bullet a))) \quad (5)$$

$$\begin{aligned} (((w^* \bullet a) \cong ((u^* \bullet v^*) \bullet a)) \wedge (((u^* \bullet v^*) \bullet a) \cong (u^* \bullet (v^* \bullet a)))) \\ \supset ((w^* \bullet a) \cong (u^* \bullet (v^* \bullet a))). \end{aligned} \quad (3)$$

■

Lemma 5. If $u \underset{\Pi}{\Rightarrow} v$, then $(u^* \cong v^*)$ is rooted.

Proof. For some i , $1 \leq i \leq K$, we have either $u = pg_i q$, $v = ph_i q$, or $u = ph_i q$, $v = pg_i q$, where $p, q \in \{a, b\}^*$. We may assume that in fact $u = pg_i q$, $v = ph_i q$, because in the other case we could use the following Herbrand instance of (2):

$$((v^* \cong u^*) \supset (u^* \cong v^*)).$$

The proof now divides into three cases.

Case I: $p = q = 0$. Then the sentence $(u^* \cong v^*)$ is just $(5 + i)$ and is therefore in Γ .

Case II: $p = 0, q \neq 0$. Using $(5 + i)$ and the following Herbrand instance of (4):

$$(((g_i^* \cong h_i^*) \wedge (q^* \cong q^*)) \supset ((g_i^* \bullet q^*) \cong (h_i^* \bullet q^*))),$$

we see that the sentence

$$((g_i^* \bullet q^*) \cong (h_i^* \bullet q^*))$$

is rooted. Using Lemma 4 and Herbrand instances of (2) and (3) we obtain the result.

Case III: $p, q \neq 0$. Using Case II, the sentence $((g_i q)^* \cong (h_i q)^*)$ is rooted. Using the Herbrand instance of (4):

$$(((p^* \cong p^*) \wedge ((g_i q)^* \cong (h_i q)^*)) \supset ((p^* \bullet (g_i q)^*) \cong (p^* \bullet (h_i q)^*))),$$

we see that

$$((p^* \bullet (g_i q)^*) \cong (p^* \bullet (h_i q)^*))$$

is rooted. The result now follows using Lemma 4 and Herbrand instances of (2) and (3). ■

Lemma 6. If $u \xrightarrow{\Pi} v$, then $(u^* \cong v^*)$ is rooted.

Proof. The proof is by induction on the length of a derivation of v from u . If this length is 1, then $v = u$, and we may use a Herbrand instance of (1). To complete the proof, we may assume that $u \xrightarrow{\Pi} w \xrightarrow{\Pi} v$, where it is known that $(u^* \cong w^*)$ is rooted. By Lemma 5, $(w^* \cong v^*)$ is rooted. We then get the result by using the following Herbrand instance of (3):

$$(((u^* \cong w^*) \wedge (w^* \cong v^*)) \supset (u^* \cong v^*)).$$

Combining Lemmas 3 and 6, we obtain

Lemma 7. $u \xrightarrow{\Pi} v$ if and only if $\Gamma \vDash (u^* \cong v^*)$.

Now it is easy to complete the proof of our theorem. If we possessed an algorithm for testing a given W-sentence for satisfiability, we could use it to test the sentence

$$\bigwedge_{\alpha \in \Gamma} \alpha \wedge \neg(u^* \cong v^*)$$

and therefore, by Theorem 3.1, to test the correctness of the logical inference $\Gamma \vDash (u^* \cong v^*)$. This would in turn lead to an algorithm for solving the word problem for Π , which we know is unsolvable. ■

A final remark: We really have been working with the axiomatizable theory T_Γ . Thus what Lemma 7 states is just that

$$u \xrightarrow[\Pi]{} v \quad \text{if and only if} \quad \vdash_{\Gamma} (u^* \cong v^*). \quad (8.3)$$

Hence we conclude that the theory T_Γ is not recursive. [If it were, we could use (8.3) to solve the word problem for Π .] Thus we have proved

Theorem 8.2. There are axiomatizable theories which are not recursive.

Loop Programs

1. The Language L and Primitive Recursive Functions

In this chapter, we study a programming language L whose programs compute exactly the primitive recursive functions. In addition to zero, increment, and assignment statements, L contains a LOOP instruction which functions much like a DO-loop in FORTRAN. For this reason, programs written in L are called *loop programs*. It will turn out that L offers us a natural way to estimate the complexity of computations performed by programs written in the language.

The language L has a syntax not unlike the language \mathcal{S} of Chapter 2. In particular, L will use the same list of variables that \mathcal{S} does.

The language L has the following five types of instructions:

1. $V \leftarrow 0$ zero,
2. $V \leftarrow V + 1$ increment,
3. $V \leftarrow V'$ assignment,
4. LOOP V loop,
5. END end,

where V, V' are any variables. The first three instruction types have the same semantics as the corresponding instruction or macros of \mathcal{S} . The LOOP and END statements always occur in pairs, and must be matched like left and right parentheses. LOOP-END pairs specify that the intervening code is to be executed a certain number of times before the instruction following the END is executed. The number of times this block of instructions is to be executed is the number stored in X when the LOOP X instruction is first encountered. Thus, even if the value of X is changed during the course of the

computation, the number of loop iterations is unaffected. This is important because it guarantees that *every loop program eventually halts*. To emphasize the point, note that the program

```
X ← 0
X ← X + 1
LOOP X
X ← X + 1
END
Y ← X
```

terminates with Y having the value 2, rather than entering an infinite loop. As in Chapter 2, we will say that this function computes the constant function $f(x) = 2$, and as in the case of the language \mathcal{L} , variables other than input variables are assumed to be initialized to 0. In general, every loop program computes a total function.

For a slightly more complicated example, consider the following program, where we have indented to facilitate reading:

```
Z ← 0
LOOP X1
  LOOP X2
    Z ← Z + 1
  END
END
Y ← Z
```

The inner loop of this program adds 1 to the value of Z , x_2 times. Since this inner loop is performed x_1 times, this program computes $x_1 \cdot x_2$.

This is an example of a loop program in which the maximum depth of nesting of LOOP-END pairs is 2. In general, we define L_n to be the class of loop programs with LOOP-END pairs nested to a depth of at most n . \mathcal{L}_n is then the class of functions computable by programs in L_n . Thus the two examples above are L_1 and L_2 programs, respectively. Note that L_0 is the class of programs containing no LOOP instructions.

We shall prove now that the class of functions computed by loop programs is exactly the class of primitive recursive functions.

Theorem 1.1. Every primitive recursive function belongs to $\bigcup_{n=0}^{\infty} \mathcal{L}_n$.

Proof. Recalling the definition in Chapter 3, Section 3, we will first show that the initial functions are actually in \mathcal{L}_0 , and then show that $\bigcup_{n=0}^{\infty} \mathcal{L}_n$ is closed under composition and recursion.

For the initial functions, we note that the following programs of L_0 suffice:

$$\begin{array}{c} \frac{y \leftarrow s(x)}{Z \leftarrow X} \\ \frac{y \leftarrow n(x)}{Y \leftarrow 0} \\ \frac{y \leftarrow u_i^n(x_1, \dots, x_n)}{Y \leftarrow X_i} \end{array}$$

$$\begin{array}{l} Z \leftarrow Z + 1 \\ Y \leftarrow Z \end{array}$$

Note that in each case *the values of the input variables have been preserved.*

A loop program to compute the composition

$$f(g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m))$$

would simply be

$$\begin{array}{l} Z_1 \leftarrow g_1(X_1, \dots, X_m) \\ \vdots \\ Z_n \leftarrow g_n(X_1, \dots, X_m) \\ Y \leftarrow f(Z_1, \dots, Z_n) \end{array}$$

where each statement of the program may be thought of as a macro to perform the desired function. It follows from the (not explicitly stated) induction hypothesis that such a program exists in L . (The existence of macro expansions corresponding to any function computable by a program of L follows as in Chapter 2, Section 5.) The program thus obtained is in L_k , where k is the maximum depth of nesting in any of the macro expansions. Thus, the composition of functions in \mathcal{L}_k yields a function in \mathcal{L}_k .

Our final task is to deal with recursion. In Chapter 3, Section 2, we considered two types of recursion. The simpler type has the form

$$\begin{aligned} h(0) &= k, \\ h(z + 1) &= g(z, h(z)). \end{aligned}$$

In this case, $h(x)$ is programmed in L as

$$\begin{aligned} &Y \leftarrow 0 \\ &\left. \begin{array}{l} Y \leftarrow Y + 1 \\ \vdots \\ Y \leftarrow Y + 1 \end{array} \right\} k \text{ times} \\ &Z \leftarrow 0 \\ &\text{LOOP } X \\ &\quad Y \leftarrow g(Z, Y) \\ &\quad Z \leftarrow Z + 1 \\ &\text{END} \end{aligned}$$

Similarly, if

$$h(x_1, \dots, x_n, 0) = f(x_1, \dots, x_n),$$

$$h(x_1, \dots, x_n, z + 1) = g(z, h(x_1, \dots, x_n, z), x_1, \dots, x_n),$$

then the function $h(x_1, \dots, x_n, x_{n+1})$ is computed by the program

```

 $Y \leftarrow f(X_1, \dots, X_n)$ 
 $Z \leftarrow 0$ 
LOOP  $X_{n+1}$ 
 $Y \leftarrow g(Z, Y, X_1, \dots, X_n)$ 
 $Z \leftarrow Z + 1$ 
END

```

For this case, if $f \in \mathcal{L}_k$ and $g \in \mathcal{L}_m$, then clearly $h \in \mathcal{L}_{\max(k, m+1)}$. ■

As we shall see, the converse of Theorem 1.1 is also true. In proving this converse, it will be helpful to consider loop programs which contain only local variables: Z_1, Z_2, Z_3, \dots . We also assume that whenever a block P is contained between a “LOOP V ” and “END” pair, the variable V does not occur in the block P . This clearly involves no loss in generality since the program segment:

LOOP V

(1.1)

END

is equivalent to

$V' \leftarrow V$
LOOP V'

P

END

where V' may be selected to be a variable which occurs nowhere else in the program.

Now, considering programs of L containing only local variables, we think of the local variables contained in the program as being assigned some initial values; since these local variables will in general have different values when the program terminates, we may think of the program as yielding a function for each local variable. That is, if P is a program containing the local variables Z_1, Z_2, \dots, Z_n we may write

$$\begin{aligned}
 Z_1 &\leftarrow f_1(Z_1, \dots, Z_n) \\
 Z_2 &\leftarrow f_2(Z_1, \dots, Z_n) \\
 &\vdots \\
 Z_n &\leftarrow f_n(Z_1, \dots, Z_n)
 \end{aligned} \tag{1.2}$$

to indicate the transformation in the values of the local variables effected by P . If we now add a LOOP-END pair as in (1.1), the new program Q will give a corresponding transformation, which we may write as

$$\begin{aligned} Z_1 &\leftarrow g_1(Z_1, \dots, Z_n, V) \\ Z_2 &\leftarrow g_2(Z_1, \dots, Z_n, V) \\ &\vdots \\ Z_n &\leftarrow g_n(Z_1, \dots, Z_n, V) \end{aligned}$$

The key result is

Lemma 1. If the functions f_1, \dots, f_n , are primitive recursive, then so are the functions g_1, \dots, g_n .

Proof. We write \vec{z} to stand for z_1, z_2, \dots, z_n . Then we have

$$g_i(\vec{z}, 0) = z_i$$

$$g_i(\vec{z}, t + 1) = f_i(g_1(\vec{z}, t), \dots, g_n(\vec{z}, t)),$$

$$i = 1, 2, \dots, n.$$

Thus, the functions g_i are expressed in terms of the functions f_i by means of a kind of interlocking, or simultaneous, recursion. To obtain the result, we use Gödel numbers (recall Chapter 3, Section 8), setting

$$\tilde{g}(\vec{z}, u) = [g_1(\vec{z}, u), \dots, g_n(\vec{z}, u)].$$

Thus we have

$$\tilde{g}(\vec{z}, 0) = [z_1, \dots, z_n],$$

$$\tilde{g}(\vec{z}, t + 1) = [k_1, \dots, k_n],$$

where $k_i = f_i((\tilde{g}(\vec{z}, t))_1, \dots, (\tilde{g}(\vec{z}, t))_n)$, $i = 1, 2, \dots, n$. Thus, $\tilde{g}(\vec{z}, u)$ is primitive recursive. Since

$$g_i(\vec{z}, u) = (\tilde{g}(\vec{z}, u))_i, \quad i = 1, 2, \dots, n,$$

we conclude that each g_i is primitive recursive. ■

Lemma 2. Let P be a program in L containing only the variables Z_1, Z_2, \dots, Z_n . Let P transform the values of Z_1, \dots, Z_n as in (1.2). Then, f_1, f_2, \dots, f_n are all primitive recursive.

Proof. Suppose first that $P \in L_0$. Then P contains no LOOP instructions and hence effectively can only set a variable equal to 0 or to the value of another variable, or add 1 to its value some finite number of times. Thus, for each $i = 1, 2, \dots, n$, we must have

$$f_i(Z_1, \dots, Z_n) = Z_j + k$$

or

$$f_i(Z_1, \dots, Z_n) = k$$

for some $k \in N$. These functions are obviously primitive recursive.

Assuming the result for all $P \in L_m$, let us prove it for a program $P \in L_{m+1}$. Such a program can be decomposed as in Fig. 1.1, where Q_0, Q_1, \dots, Q_r and P_1, P_2, \dots, P_r all are in L_m . By the induction hypothesis, the functions corresponding to $Q_0, Q_1, \dots, Q_r, P_1, P_2, \dots, P_r$ are all primitive recursive.

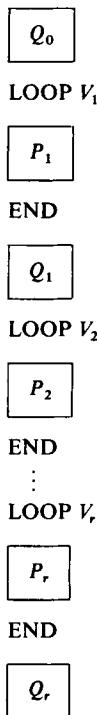
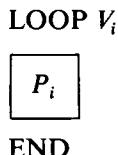


Fig. 1.1

By Lemma 1 the same is true of the functions corresponding to the programs



The result now follows from the fact that the class of primitive recursive functions is closed under composition. ■

We can now easily prove

Theorem 1.2. $\bigcup_{n=0}^{\infty} \mathcal{L}_n$ is precisely the class of primitive recursive functions.

Proof. By Theorem 1.1, we need only show that each program in L computes a primitive recursive function. Let a program P of L compute the function $h(x_1, \dots, x_k)$. By a simple substitution of variables, P can be transformed to a program of the form shown in Fig. 1.2, in which Q contains only local variables Z_1, \dots, Z_m and $k < s \leq m$. Using the notation of Eqs. (1.2), we then have

$$h(x_1, \dots, x_k) = f_s(x_1, \dots, x_k, 0, \dots, 0).$$

The result then follows from Lemma 2. ■

```

 $Z_1 \leftarrow X_1$ 
 $Z_2 \leftarrow X_2$ 
 $\vdots$ 
 $Z_k \leftarrow X_k$ 
 $Q$ 
 $Y \leftarrow Z_s$ 

```

Fig. 1.2

2. Running Time

Having shown that L is a language for computing primitive recursive functions, we now focus our attention on the running time of such programs. We define the running time to be the total number of zero, increment, and assignment statements executed. (The loop and end statements are not counted for simplicity, although the results which we shall prove hold under a variety of different ways of counting time.) Thus, if P is a loop program with input variables X_1, \dots, X_n , then $T_P(x_1, \dots, x_n)$ is the *running time* of P defined as the total number of executions of zero, increment, and assignment statements in the given computation.

We now show that the running time T_P can be computed in L by a program with no greater depth of nesting than the original program P .

Theorem 2.1. If $P \in L_n$, then $T_P \in \mathcal{L}_n$.

Proof. To compute T_P for a given program P , we simply modify P by inserting a counter T which is incremented each time a zero, increment, or

assignment statement is executed in P . This is accomplished by placing a “ $T \leftarrow T + 1$ ” instruction immediately following each such statement in P . Thus T keeps count of the number of executions of these instructions, and at the end of the computation, T contains the value of T_P . This modified program obviously has the same depth of nesting as P . ■

Our goal is to obtain an a priori bound on the running time of a given program. Thus, using T_P , the running time itself, as this bound is really not satisfactory. Knowledge of T_P merely tells the programmer that the program runs as long as it runs, which the programmer presumably already knew. What we want to find are bounding functions which are easy to understand in themselves and which therefore transmit information to the programmer about a given program’s running time.

We write $g^{(m)}(x)$ for the result of composition of g with itself m times. In particular,

$$\begin{aligned} g^{(0)}(x) &= x \\ g^{(1)}(x) &= g(x), \\ g^{(2)}(x) &= g(g(x)), \end{aligned}$$

etc. In general, we have the recursion

$$\begin{aligned} g^{(0)}(x) &= x, \\ g^{(m+1)}(x) &= g(g^{(m)}(x)). \end{aligned}$$

We define

$$\begin{aligned} f_0(x) &= \begin{cases} x + 1 & \text{if } x = 0 \text{ or } x = 1 \\ x + 2 & \text{otherwise,} \end{cases} \\ f_{n+1}(x) &= f_n(x)(1). \end{aligned}$$

We will use the functions $f_n(x)$ to solve the problem of providing a priori bounds for the running time of loop programs. As we shall see, these are very rapidly growing functions; for the first few, we have

$$\begin{aligned} f_1(x) &= 2x && \text{if } x \neq 0, \\ f_2(x) &= 2^x \\ f_3(x) &= 2^{2^{\cdot^{\cdot^{\cdot^2}}}} \quad \left\{ \begin{array}{l} \text{x times.} \end{array} \right. \end{aligned}$$

We now derive some of the basic properties of these functions:

Lemma 1. $f_{n+1}(x + 1) = f_n(f_{n+1}(x))$.

Proof.

$$\begin{aligned} f_{n+1}(x + 1) &= f_n^{(x+1)}(1) \\ &= f_n(f_n^{(x)}(1)) \\ &= f_n(f_{n+1}(x)). \end{aligned}$$
■

Lemma 2. $f_0^{(k)}(x) \geq k$.

Proof. By induction on k . This result is clear for $k = 0$. Assuming the result for k and noting that $f_0(x) \geq x + 1$ for all x , we have

$$\begin{aligned} f_0^{(k+1)}(x) &= f_0(f_0^{(k)}(x)) \\ &\geq f_0^{(k)}(x) + 1 \geq k + 1. \end{aligned}$$
■

Lemma 3. $f_n(x) > x$.

Proof. We prove this lemma by induction on n . For $n = 0$, we note that

$$f_0(x) \geq x + 1 > x.$$

Assuming the result for $n = k$, that is, that $f_k(x) > x$, for all x , we prove *by induction on x* that

$$f_{k+1}(x) > x \tag{2.1}$$

for all x . (Thus, the entire proof has the form of a “double induction.”) For $x = 0$, we have

$$f_{k+1}(0) = f_k^{(0)}(1) = 1 > 0.$$

Assuming (2.1) for $x = m$, and using Lemma 1 and *both* induction hypotheses we have

$$\begin{aligned} f_{k+1}(m + 1) &= f_k(f_{k+1}(m)) \\ &> f_{k+1}(m) > m. \end{aligned}$$

Since $f_{k+1}(m)$ and m are integers, it follows that $f_{k+1}(m + 1) > m + 1$. ■

Lemma 4. $f_n(x + 1) > f_n(x)$.

Proof. For $n = 0$, the result follows from the definition of f_0 . Using Lemmas 1 and 3 we have

$$f_{k+1}(x + 1) = f_k(f_{k+1}(x)) > f_{k+1}(x).$$
■

Lemma 5. $f_n(x) \leq f_{n+1}(x)$.

Proof. Although the proof is *not* by induction, we begin by remarking that the result is obvious for $x = 0$ since both sides of the inequality are 1. By Lemma 3, $f_{n+1}(x) \geq x + 1$, so by Lemma 4,

$$f_{n+1}(x + 1) = f_n(f_{n+1}(x)) \geq f_n(x + 1).$$
■

Lemma 6. $f_n^{(k)}(x) < f_n^{(k+1)}(x)$.

Proof. Using Lemma 3,

$$f_n^{(k)}(x) < f_n(f_n^{(k)}(x)) = f_n^{(k+1)}(x). \quad \blacksquare$$

These last three lemmas tell us that $f_n^{(k)}(x)$ is increasing in each of x , n , and k . We now sharpen Lemma 6 to get

Lemma 7. $f_n^{(k+1)}(x) \geq 2f_n^{(k)}(x)$ for $n \geq 1$.

Proof. By induction on k . For $k = 0$, using Lemma 5,

$$\begin{aligned} 2f_n^{(0)}(x) &= 2x \\ &\leq f_1(x) \leq f_n(x) = f_n^{(1)}(x). \end{aligned}$$

Assuming the result for k , we have

$$\begin{aligned} 2f_n^{(k+1)}(x) &= 2f_n^{(k)}(f_n(x)) \\ &\leq f_n^{(k+1)}(f_n(x)) = f_n^{(k+2)}(x). \end{aligned} \quad \blacksquare$$

Lemma 8. $f_n^{(k)}(x) + x \leq f_n^{(k+1)}(x)$ for $n \geq 1$.

Proof. Using Lemma 7, we have for $k = 0$,

$$f_n^{(1)}(x) \geq 2f_n^{(0)}(x) = f_n^{(0)}(x) + x.$$

For $k > 0$, using Lemmas 7, 6, and 3, we have

$$\begin{aligned} f_n^{(k+1)}(x) &\geq 2f_n^{(k)}(x) \\ &= f_n^{(k)}(x) + f_n^{(k)}(x) \\ &\geq f_n^{(k)}(x) + f_n^{(1)}(x) > f_n^{(k)}(x) + x. \end{aligned} \quad \blacksquare$$

Lemma 9. $f_1^{(k)}(x) \geq 2^k \cdot x$.

Proof. By induction on k . For $k = 0$, $f_1^{(0)}(x) = x = 2^0 \cdot x$. Assuming the result for k , we have, using Lemmas 4 and 7,

$$\begin{aligned} f_1^{(k+1)}(x) &= f_1(f_1^{(k)}(x)) \\ &\geq f_1(2^k \cdot x) \geq 2f_1^{(0)}(2^k \cdot x) \\ &= 2 \cdot 2^k \cdot x = 2^{k+1} \cdot x. \end{aligned} \quad \blacksquare$$

Lemma 8 can be used to make an important observation. Suppose that u is the largest value of the inputs x_1, \dots, x_m to a program P and that it is known that $T_P(x_1, \dots, x_m) \leq f_n^{(k)}(u)$. Then, since the value of no variable can be increased by more than 1 during a single step we have that during this computation, *all variables in P are restricted to values*

$$\leq u + f_n^{(k)}(u) \leq f_n^{(k+1)}(u).$$

We shall call this fact the *growth limitation property*.

We now prove one of our main results:

Theorem 2.2 (Bounding Theorem). Let $P \in L_n$. Then there is a k such that

$$T_P(x_1, \dots, x_m) \leq f_n^{(k)}(\max(x_1, \dots, x_m)).$$

Proof. By induction on n . For $n = 0$, there are no loops in P , and hence

$$\begin{aligned} T_P(x_1, \dots, x_m) &= k \\ &\leq f_0^{(k)}(\max(x_1, \dots, x_m)), \end{aligned}$$

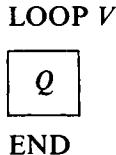
where we have used Lemma 2.

Now assume the result is known for $n - 1$, and let $P \in L_n$. Let u denote $\max(x_1, \dots, x_m)$. We consider three cases:

Case 1. $P \in L_{n-1}$. Then, using Lemma 5 and the induction hypothesis,

$$T_P(x_1, \dots, x_m) \leq f_{n-1}^{(k)}(u) \leq f_n^{(k)}(u).$$

Case 2. P has the form



where $Q \in L_{n-1}$. The case $n = 1$, $Q \in L_0$ must be handled separately. In this case, $T_Q = q$ for some constant q . Choose k such that $q \leq 2^k$. Then, with v be the value of V ,

$$\begin{aligned} T_P(x_1, \dots, x_m) &= qv \\ &\leq qu \leq 2^k \cdot u \leq f_1^{(k)}(u) \end{aligned}$$

using Lemma 9. (Recall that if V is not one of the input variables, it is initialized to 0.)

Thus, we may assume that $n > 1$. By the induction hypothesis we have

$$T_Q(x_1, \dots, x_m) \leq f_{n-1}^{(j)}(u) \tag{2.2}$$

for some j . Now, a computation by P is obtained by running Q v times, each time with possibly larger values for Q 's inputs. Thus, in order to get the desired bound on P 's running time, we must first get bounds on the value of the variables after each execution of Q . By the growth limitation property we see that the value of each variable at the end of one execution of Q is

bounded by $f_{n-1}^{(j+1)}(u)$. We therefore know that after running Q twice, the total running time will be bounded by

$$f_{n-1}^{(j)}(u) + f_{n-1}^{(j)}(f_{n-1}^{(j+1)}(u)) = f_{n-1}^{(j)}(u) + f_{n-1}^{(2j+2)}(u).$$

By the growth limitation property, the values of the variables after two successive executions of Q will be $\leq f_{n-1}^{(2j+3)}(u)$.

Repeating this reasoning, we find that after i successive executions of Q , the value of the variables and the running time are both bounded by $f_{n-1}^{(i(j+2)-1)}(u)$. Using this fact, and letting $Q[r]$ denote the program consisting of r successive copies of the program Q , we have that

$$\begin{aligned} T_P(x_1, \dots, x_m) &\leq T_{Q[u]}(x_1, \dots, x_m) \\ &\leq f_{n-1}^{(u \cdot (j+2))}(u) \\ &\leq f_{n-1}^{(u \cdot (j+2))}(f_n(u)) \quad (\text{by Lemmas 3 and 4}) \\ &= f_{n-1}^{(u \cdot (j+2))}(f_{n-1}^{(u)}(1)) \quad (\text{by definition}) \\ &= f_{n-1}^{(u \cdot (j+3))}(1) \\ &\leq f_{n-1}^{(u \cdot 2(j+2))}(1) \quad (\text{by Lemma 6}) \\ &\leq f_{n-1}^{(f_n^{(j+2)}(u))}(1) \quad (\text{by Lemma 9}) \\ &\leq f_{n-1}^{(f_n^{(j+2)}(u))}(1) \quad (\text{by Lemmas 5 and 6}) \\ &= f_n(f_n^{(j+2)}(u)) \quad (\text{by definition}) \\ &= f_n^{(j+3)}(u). \end{aligned}$$

Letting $k = j + 3$, we have

$$T_P(x_1, \dots, x_m) \leq f_n^{(k)}(u).$$

Case 3. For the general case, we decompose the program P as in Fig. 1.1. Then P is broken up into, say s subprograms, each of which falls under either Case 1 or Case 2. The outputs of each of these subprograms are inputs to the next. Thus, using Cases 1 and 2 and the growth limitation property, there are numbers $k_1, \dots, k_s \geq 1$ such that

$$\begin{aligned} T_P(x_1, \dots, x_m) &\leq f_n^{(k_1)}(u) + f_n^{(k_2)}(f_n^{(k_1)}(u)) + \dots + f_n^{(k_s)}(\dots(f_n^{(k_1)}(u))) \\ &\leq f_n^{(k_1+k_2+\dots+k_s+s-1)}(u), \end{aligned}$$

using Lemma 8 repeatedly. ■

Using the growth limitation property, we then have

Corollary 2.3. If $g \in \mathcal{L}_n$, then there is a constant k such that

$$g(x_1, \dots, x_m) \leq f_n^{(k)}(\max(x_1, \dots, x_m)).$$

3. \mathcal{L}_n as a Hierarchy

The next issue to be investigated is whether we always gain something in computational power by permitting an additional level of looping. The answer is “Yes.” We shall prove that for all $n \in N$, $\mathcal{L}_{n+1} - \mathcal{L}_n \neq \emptyset$.

Theorem 3.1. For $n \geq 1$, $f_n \in \mathcal{L}_n$.

Proof. By induction on n . For $n = 1$

$$\begin{aligned} f_1(0) &= 1, \\ f_1(x) &= 2x \quad \text{for } x > 0. \end{aligned}$$

Thus, the following program computes f_1 :

```

 $Y \leftarrow Y + 1$ 
LOOP X
 $X \leftarrow X + 1$ 
 $Y \leftarrow X$ 
END

```

By definition, $f_{k+1}(x) = f_k^{(x)}(1)$, and therefore the following program computes $f_{k+1}(x)$:

```

 $Y \leftarrow Y + 1$ 
 $Z \leftarrow Z + 1$ 
LOOP X
 $Y \leftarrow f_k(Z)$ 
 $Z \leftarrow Y$ 
END

```

By the induction hypothesis $f_k \in \mathcal{L}_k$ and thus this program is in L_{k+1} . ■

We now wish to show that $f_n \notin \mathcal{L}_{n-1}$. We first introduce some terminology. We say that a predicate is true *almost everywhere* (a.e.) provided that it is true for all but a finite set of integers.

We say a function f *majorizes* a function g and write $f > g$ provided that

$$f(x) > g(x) \quad \text{a.e.}$$

That is, in such a case, “ultimately” f is larger than g .

Theorem 3.2. For all n, k , $f_{n+1} > f_n^{(k)}$.

Proof. For $n = 0$, we have

$$\begin{aligned} f_0^{(k)}(x) &= x + 2k \quad \text{for } x \geq 2, \\ f_1(x) &= 2x. \end{aligned}$$

But $2x > x + 2k$ since $x > 2k$ a.e. for any fixed k .

For $n \neq 0$, we proceed by induction on k . For $k = 0$, we have using Lemma 3,

$$f_{n+1}(x) > x = f_n^{(0)}(x).$$

Suppose that $f_{n+1} > f_n^{(k)}$. Then, for x sufficiently large,

$$\begin{aligned} f_n^{(k+1)}(x) &< f_n^{(k+1)}(2x - 4) && \text{(by Lemma 4)} \\ &= f_n^{(k+1)}(f_1(x - 2)) \\ &\leq f_n^{(k+1)}(f_n(x - 2)) && \text{(by Lemmas 4 and 5)} \\ &= f_n^{(2)}(f_n^{(k)}(x - 2)) \\ &\leq f_n^{(2)}(f_{n+1}(x - 2)) && \text{(by the induction hypothesis)} \\ &= f_n^{(2)}(f_n^{(x-2)}(1)) \\ &= f_n^{(x)}(1) \\ &= f_{n+1}(x). \end{aligned}$$
■

We can now assemble the pieces and prove the following key result:

Theorem 3.3. $f_{n+1} \in \mathcal{L}_{n+1} - \mathcal{L}_n$.

Proof. By Theorem 3.1, $f_{n+1} \in \mathcal{L}_{n+1}$. Suppose that also $f_{n+1} \in \mathcal{L}_n$. By Corollary 2.3, it would then follow that

$$f_{n+1}(x) \leq f_n^{(k)}(x)$$

for some value of k and all x .

But by Theorem 3.2, we have that

$$f_n^{(k)}(x) < f_{n+1}(x) \quad \text{a.e.,}$$

and hence it would follow that

$$f_{n+1}(x) < f_{n+1}(x) \quad \text{a.e.,}$$

which is clearly impossible. ■

Thus we know that, for each n , there are primitive recursive functions which can be computed by a program in L_{n+1} , but not by one in L_n .

In Chapter 3 (Corollary 3.4) we showed that every primitive recursive function is computable. As our next result, we shall exhibit a function which is computable but not primitive recursive. Our function is $A(n, x) = f_n(x)$.

(The letter A is used because this function is a minor variant of a function introduced by W. Ackermann.) We know by Lemma 1 that

$$f_{i+1}(x + 1) = f_i(f_{i+1}(x));$$

thus,

$$A(i + 1, x + 1) = A(i, A(i + 1, x)).$$

In addition, we have that

$$A(i, 0) = 1,$$

$$A(0, x) = \begin{cases} x + 1 & \text{if } x = 0, 1 \\ x + 2 & \text{if } x > 1. \end{cases}$$

We shall show that $A(u, x)$ is computable by showing how to construct a program in the language \mathcal{S} which computes it. As a preliminary step, we consider the calculation of some values. Since A grows very quickly, we content ourselves with calculating $A(2, 2)$. To begin with, we have

$$A(2, 2) = A(1, A(2, 1)).$$

In order to continue, we must first compute $A(2, 1) = a$ and then $A(1, a)$. We use a stack to store the left-hand arguments while the right-hand argument is being evaluated. We use the notation (a_1, a_2, \dots, a_m) to denote the stack which contains these m items, the topmost being a_1 , with a_i on top of a_{i+1} , $i = 1, \dots, m - 1$. Thus, we place 1 on the stack and proceed to evaluate $A(2, 1) = A(1, A(2, 0))$. Again the left-hand argument is pushed onto the stack, giving us $(1, 1)$ and then we evaluate $A(2, 0)$. $A(2, 0) = 1$, so that we must pop the topmost element off the stack and now evaluate $A(1, 1)$. $A(1, 1) = A(0, A(1, 0))$. We push 0 onto the stack, yielding $(0, 1)$, and evaluate $A(1, 0) = 1$. Popping 0 off the top of the stack, we are left with (1) , and must calculate $A(0, 1) = 2$. Again popping the stack we get $A(1, 2) = A(0, A(1, 1))$. Pushing 0 onto the stack gives us (0) . We have already found that $A(1, 1) = A(0, A(1, 0)) = A(0, 1) = 2$, so we now compute $A(0, 2) = 4$, and hence we have found that

$$\begin{aligned} A(2, 2) &= A(1, A(2, 1)) = A(1, 2) \\ &= A(0, A(1, 1)) = A(0, 2) = 4. \end{aligned}$$

Theorem 3.4. $A(n, x)$ is computable.

Proof. To code the stack, we use two variables L and S with L denoting the length of the stack, and S containing the actual values that have been placed on the stack. In order to push q onto the stack we set

$$\begin{aligned} L &\leftarrow L + 1 \\ S &\leftarrow \langle q, S \rangle \end{aligned}$$

where we are using the pairing function defined in Chapter 3.

To pop the stack, we set

$$\begin{aligned} L &\leftarrow L - 1 \\ i &\leftarrow l(S) \\ S &\leftarrow r(S) \end{aligned}$$

The program to compute $A(i, X)$ is then

[A]	IF $X \neq 0$ GOTO D $X \leftarrow 1$	$[A(i, 0) = 1; \text{ whenever } A(i, x)$ is calculated, it is stored in x .]
[B]	IF $L = 0$ GOTO G $L \leftarrow L - 1$ $i \leftarrow l(S)$ $S \leftarrow r(S)$ GOTO A	[Is the stack empty? If not, pop it.]
[C]	$X \leftarrow X + 2$ GOTO B	[Execute when $i = 0, x \neq 0, 1$. $A(0, x) = x + 2$ if $x > 1$.]
[D]	IF $i \neq 0$ GOTO F IF $X \neq 1$ GOTO C $X \leftarrow 2$ GOTO B	$[A(0, 1) = 2.]$
[F]	$X \leftarrow X - 1$ $L \leftarrow L + 1$ $S \leftarrow \langle i - 1, S \rangle$ GOTO A	[Execute when $i \neq 0, x \neq 0$; $A(i, x) = A(i - 1, A(i, x - 1))$. Put $(i - 1)$ on the stack and update x to $x - 1$.]
[G]	$X \leftarrow X$	[Execute when the stack is empty and the value of $A(i, x)$ has been stored in X .]

We now complete our results about the function A by proving the following result:

Theorem 3.5. The functions $A(n, x)$ and $A(x, x)$ are both computable, but not primitive recursive.

Proof. We have seen that $A(n, x)$ is computable, and therefore, so is $A(x, x)$.

Let $g(x) = A(x, x) = f_x(x)$, and suppose that g were primitive recursive. Then, by Theorem 1.1, $g \in \mathcal{L}_m$ for some m . By Corollary 2.3, there would be some $k \in N$ such that

$$g(x) \leq f_m^{(k)}(x)$$

for all x . By Theorem 3.2,

$$f_m^{(k)}(x) < f_{m+1}(x) \quad \text{a.e.}$$

Hence,

$$g(x) < f_{m+1}(x) \quad \text{a.e.}$$

Thus there certainly exists $n_0 > m + 1$ such that

$$f_{n_0}(n_0) = g(n_0) < f_{m+1}(n_0).$$

But, by Lemma 5 of Section 2, since $n_0 > m + 1$, we have

$$f_{n_0}(n_0) \geq f_{m+1}(n_0),$$

which is a contradiction.

This proves that $A(x, x)$ is not primitive recursive. Of course if $A(n, x)$ were primitive recursive, $A(x, x)$ would be also. Thus our proof is complete. ■

4. A Converse to the Bounding Theorem

We shall prove a rather surprising converse to the bounding theorem. We begin with

Lemma 1. The function

$$h(x_1, \dots, x_m) = \sum_{i=1}^m x_i$$

is in the class \mathcal{L}_1 .

Proof. The program of Fig. 4.1 clearly computes h . ■

```

 $Y \leftarrow X_1$ 
LOOP  $X_2$ 
     $Y \leftarrow Y + 1$ 
END
LOOP  $X_3$ 
     $Y \leftarrow Y + 1$ 
END
:
LOOP  $X_m$ 
     $Y \leftarrow Y + 1$ 
END

```

Fig. 4.1. An L_1 program which computes $\sum_{i=1}^m x_i$.

Lemma 2. For $n \geq 1$, and all $k \in N$, the function $g(x_1, \dots, x_m) = f_n^{(k)}(\sum_{i=1}^m x_i)$ is in the class \mathcal{L}_n .

Proof. The following program, which by Theorem 3.1, is in L_n , computes g :

$$\begin{aligned} Y &\leftarrow \sum_{i=1}^m X_i \\ Y &\leftarrow f_n(Y) \\ Y &\leftarrow f_n(Y) \\ &\vdots \\ Y &\leftarrow f_n(Y) \end{aligned} \quad \left. \right\} k$$

Of course, each macro $Y \leftarrow f_n(Y)$ can be taken to abbreviate

$$\begin{aligned} Z_1 &\leftarrow Y \\ Z_2 &\leftarrow f_n(Z_1) \\ Y &\leftarrow Z_2 \end{aligned}$$

■

Theorem 4.1. Let P be a loop program which computes $g(x_1, \dots, x_m)$, and let $n \geq 2$, and let there be a constant k such that

$$T_P(x_1, \dots, x_m) \leq f_n^{(k)}(\max(x_1, \dots, x_m)).$$

Then

$$g \in \mathcal{L}_n.$$

Proof. Let P be S_1, \dots, S_z where each S_i , $1 \leq i \leq z$, is an instruction of L . We construct a new program P' in L_n which computes g , in which corresponding to each S_i of P there will be a subprogram Σ_i of P' , to be described below. Associated with each subprogram Σ_i , there will be a variable T_i which we think of as a binary switch. Thus T_i will always have the value 0 or 1. At any particular time, exactly one of these switches will be nonzero, indicating which instruction of P is currently being simulated. Thus initially T_1 is set to 1, and $T_i = 0$ for $2 \leq i \leq z$. Finally, controlling the execution of the program P' is a counter C which is initialized in a subprogram Q to $f_n^{(k)}(\sum_{i=1}^m x_i)$. By Lemma 2, we know that we may assume that Q is in L_n . Since by Lemma 4 of Section 2,

$$f_n^{(k)}(\max(x_1, \dots, x_m)) \leq f_n^{(k)}\left(\sum_{i=1}^m x_i\right),$$

our hypothesis guarantees that the initial value of the counter C will be at least as large as the total number of steps needed to carry out the computation of $g(x_1, \dots, x_m)$ by P . In the program P' , we make free use of the macro

$$V \leftarrow 1$$

which, of course, abbreviates the expansion

$$\begin{aligned} V &\leftarrow 0 \\ V &\leftarrow V + 1 \end{aligned}$$

The program P' is

```

 $Q$ 
 $T_1 \leftarrow 1$ 
LOOP  $C$ 
 $\Sigma_1$ 
 $\Sigma_2$ 
 $\vdots$ 
 $\Sigma_z$ 
END

```

We now define the subprograms Σ_i of P' .

1. If S_i is a zero, increment, or assignment statement, then Σ_i is the subprogram

```

LOOP  $T_i$ 
 $T_i \leftarrow 0$ 
 $T_{i+1} \leftarrow 1$ 
 $S_i$ 
END

```

At entry, $T_i = 1$ so the code in the loop is performed once. The switch indicating that Σ_i is to be executed is turned off, the switch for Σ_{i+1} is turned on, and finally S_i is executed.

2. If S_i is LOOP K , and the corresponding END is S_j , then Σ_i is the subprogram

```

LOOP  $T_i$ 
 $T_i \leftarrow 0$ 
 $U_i \leftarrow 1$ 
 $V_i \leftarrow K$ 
END
LOOP  $U_i$ 
 $U_i \leftarrow 0$ 
 $T_{j+1} \leftarrow 1$ 
 $W_i \leftarrow 0$ 
END
LOOP  $V_i$ 
 $V_i \leftarrow W_i$ 
 $W_i \leftarrow W_i + 1$ 
 $T_{j+1} \leftarrow 0$ 
 $T_{i+1} \leftarrow 1$ 
END

```

Σ_i is made up of three separate loops. The function of the first loop is to turn off the switch corresponding to S_i , turn on the switch for the second loop, and initialize a counter to K , the loop count of S_i .

In the second loop, after turning off the switch for that section, and turning on the switch for the statement following the END statement corresponding to S_i , a counter is initialized to 0.

The third loop controls the sequencing through the body of the loop of P being simulated. The control counter for the loop V_i was initialized to K in the first loop. Each time through this loop, its value is decremented by one. If the value of V_i is 0 at entry to this loop, the next subprogram to be executed will be Σ_{j+1} , the subprogram corresponding to the instruction of P following the S_i loop. If the value of V_i is not 0, the loop body has not yet been executed K times, so the switch for subprogram Σ_{j+1} is turned off, and the switch corresponding to the first statement in the loop body is turned on.

3. The code for the subprogram Σ_j , where S_j is the END statement corresponding to the LOOP statement S_i , is

```

LOOP  $T_j$ 
     $T_j \leftarrow 0$ 
     $T_{j+1} \leftarrow 1$ 
END
LOOP  $V_i$ 
     $T_{j+1} \leftarrow 0$ 
     $U_i \leftarrow 1$ 
END

```

Σ_j is made up of two loops. The first loop simply turns off the switch corresponding to Σ_j and turns on the switch for the subprogram corresponding to the instruction following the END statement in P . The second loop of Σ_j is controlled by a variable, V_i , whose value was set in Σ_i (the LOOP statement subprogram). This variable contains the number of times the loop must still be performed. If V_i is 0, the loop body has been executed K times and since the switch for Σ_{j+1} was set in the loop above, the subprogram corresponding to the instruction following the END statement will be executed next. If V_i is not 0, the loop body has not yet been executed K times, and so the switch for Σ_{j+1} is turned off, and the switch for the second loop of Σ_i is turned on.

Notice that the outermost LOOP C instruction, in conjunction with the switch mechanism, really controls the proper execution of the interior LOOP statements. By initializing C to $f_n^{(k)}(\sum_{i=1}^m x_i)$, we know that the body of the program can be executed as often as is necessary.

Since each of the subprograms Σ_i is in L_2 , and $Q \in L_n$, it follows that for $n \geq 2$, $P' \in L_n$. ■

*5. Doing without Branch Instructions

It is currently thought to be part of good programming practice to avoid the use of branch instructions. It is therefore of some interest to describe a programming language for computing partially computable functions that does not require GOTOs.

The language we suggest, W , is an extension of L . In addition to the instruction types (1)–(5) available in L , W has available the “while” instruction type:

6. WHILE $V \neq 0$ DO

Syntactically, *while* instructions are paired with END’s in the same way that loop instructions are. Semantically a program segment

WHILE $V \neq 0$ DO



END

causes P to be executed (causing possible changes in the value of V) over and over again, until V has the value 0. If V never gets the value 0, the computation never terminates. The above segment can be simulated in \mathcal{S} by

[A] IF $V = 0$ GOTO B



GOTO A

[B] ...

Hence, all functions computed by programs in W are clearly partially computable, a fact which would in any case be a consequence of Church’s thesis. We now note that the converse is also true:

Theorem 5.1. Every partially computable function can be computed by a program of the language W .

Proof. We use the normal form theorem (Theorem 7.6 in Chapter 7) according to which any partially computable function $f(x_1, \dots, x_n)$ can be written in the form

$$l(\min_z [g(x_1, \dots, x_n, z) = 0]),$$

where l is one of our primitive recursive pairing functions, and g is a primitive recursive function. Then f can be computed by the following program of W

(where we have used the fact that primitive recursive functions are computable by programs of L and hence of W):

```

 $Z \leftarrow 0$ 
 $V \leftarrow g(X_1, \dots, X_n, 0)$ 
WHILE  $V \neq 0$  DO
     $Z \leftarrow Z + 1$ 
     $V \leftarrow g(X_1, \dots, X_n, Z)$ 
END
 $Y \leftarrow l(Z)$ 
```

■

Note that by this construction no more than one while statement is ever needed for computing any particular function. Of course, in practice it is typically convenient to use several while statements in a single program.

Exercises

1. Show that there is a program P of L such that there is no algorithm that determines for a given number y whether there is an input to P which will produce y as output.
2. Show that $x \cdot y$ is in $\mathcal{L}_2 - \mathcal{L}_1$.
3. The same for x^y . (*Hint:* Write a program of L to find a and b where $y = 2^a + b$ and $b < 2^a$.)

Abstract Complexity

1. The Blum Axioms

In this chapter we will develop an abstract theory of the amount of resources needed to carry out computations. In practical terms resources can be measured in various ways: storage space used, time, some weighted average of central processor time and peripheral processor time, some combinations of space and time used, or even monetary cost. The theorems proved in this chapter are quite independent of which of these “measures” we use. We shall work with two very simple assumptions known as the *Blum axioms* after Manuel Blum who introduced them in his doctoral dissertation. These assumptions are satisfied by any of the “measures” mentioned above (if given precise definitions in any natural manner) as well as by many others.

Definition. A 2-ary partial function C on N is called a *complexity measure* if it satisfies the *Blum axioms*:

1. $C(x, i)\downarrow$ if and only if $\Phi_i(x)\downarrow$;
2. The predicate $C(x, i) \leq y$ is recursive. (This predicate is of course false if $C(x, i)\uparrow$.)

We write $C_i(x) = C(x, i)$. We think of $C_i(x)$ as the *complexity of the computation* which occurs when the program whose number is i is fed the input x . It is not very difficult to see that various natural ways of measuring complexity of computation do satisfy the Blum axioms. What is remarkable is that some very interesting and quite nontrivial results can be derived from such meager assumptions.

Let us examine some examples of proposed complexity measures:

1. $C_i(x) = \text{the number of steps in a computation by program number } i \text{ on input } x$. The first axiom is clearly satisfied; the second follows from the computability of the step-counter predicate $\text{STP}^{(1)}$.

2. $M_i(x) = \text{the largest value assumed by any variable in program number } i \text{ when this program is given input } x$, if $\Phi_i(x) \downarrow$; $M_i(x) \uparrow$ otherwise. The definition forces the first axiom to be true. The truth of the second axiom is a more subtle matter. The key observation is that for a given program, there are only finitely many *different snapshots*¹ in which all variables have values less than or equal to a given number y . Hence, given numbers i, x, y we can test the condition $M_i(x) \leq y$ by “running” program number i on the input x until one of the following occurs:

I. A snapshot is reached in which some variable has a value $> y$. Then we return the value FALSE.

II. The computation halts with all variables having values $\leq y$. Then we return the value TRUE.

III. The same snapshot is reached twice. (By the pigeon-hole principle this must happen eventually if neither I nor II occurs.) Then, recognizing that the computation is in an “infinite” loop and so will never terminate, we return to value FALSE. (The reader should note that this algorithm in no way contradicts the unsolvability of the halting problem. Case I can include both halting and nonhalting computations.)

We will make important use of this “maximum-space” complexity measure, and we reserve the notation $M_i(x)$ for it.

3. $C_i(x) = \Phi_i(x)$. Although the first Blum axiom is satisfied, the second is *certainly not*, namely: choose i so that

$$\Phi_i(x) = \begin{cases} 0 & \text{for } x \in S \\ \uparrow & \text{otherwise,} \end{cases}$$

where S is any given r.e. nonrecursive set. Then the condition $\Phi_i(x) \leq 0$ is equivalent to $x \in S$ and hence is not recursive.

As in Chapter 13, if $P(x)$ is any predicate on N , we write

$$P(x) \quad \text{a.e.,}$$

and say that $P(x)$ is true *almost everywhere*, to mean that there exists $m_0 \in N$ such that $P(x)$ is true for all $x > m_0$. Equivalently, $P(x)$ is true for all but a finite set of numbers. We may think of a partial function on N as a total function with values in the set $N \cup \{\infty\}$. That is, we write $g(x) = \infty$ to

¹ The definition of “snapshot” is in Chapter 2, Section 3.

mean that $g(x) \uparrow$. We extend the meaning of $<$ so that $n < \infty$ for all $n \in N$. $x \leq y$ continues to mean $x < y$ or $x = y$, so that $n \leq \infty$ for $n \in N$ but also $\infty \leq \infty$.

The second Blum axiom can be written in the equivalent forms:

- 2'. The predicate $C_i(x) = y$ is recursive.
- 2''. The predicate $C_i(x) < y$ is recursive.

To see that 2, 2' and 2'' are all equivalent we note that

$$C_i(x) = y \Leftrightarrow (C_i(x) \leq y \& \sim(C_i(x) \leq y - 1)) \vee (y = 0 \& C_i(x) \leq y),$$

so that 2 implies 2'. 2' implies 2'' because

$$C_i(x) < y \Leftrightarrow (\exists z)_{\leq y} (C_i(x) = z).$$

Finally, 2'' implies 2 because

$$C_i(x) \leq y \Leftrightarrow C_i(x) < y + 1.$$

Let us call a *recursive* function $r(x)$ a *scaling factor* if

- (1) r is increasing, i.e., $r(x + 1) \geq r(x)$, and
- (2) $\lim_{x \rightarrow \infty} r(x) = \infty$, i.e. r assumes arbitrarily large values.

(1) is obviously equivalent to the statement: $x \leq y$ implies $r(x) \leq r(y)$. Then we have

Theorem 1.1. Let $C_i(x)$ be a complexity measure and let $r(x)$ be a scaling factor. Let $D_i(x) = r(C_i(x))$. Then $D_i(x)$ is a complexity measure.

Proof. It is clear that D satisfies the first Blum axiom. To test $D_i(x) \leq y$, find the number t for which

$$r(0) \leq r(1) \leq r(2) \leq \cdots \leq r(t) \leq y < r(t + 1).$$

We claim that $D_i(x) \leq y$ if and only if $C_i(x) \leq t$. It remains only to verify this claim. If $C_i(x) \leq t$, then

$$D_i(x) = r(C_i(x)) \leq r(t) \leq y.$$

Otherwise, if $t + 1 \leq C_i(x)$, then

$$y < r(t + 1) \leq r(C_i(x)) = D_i(x). \quad \blacksquare$$

This theorem is hardly surprising. Naturally, if $C_i(x)$ is a plausible complexity measure, we would expect $2^{C_i(x)}$ to be one as well. What is surprising is that any pair of complexity measures are related to each other in a manner not so different from C and D in Theorem 1.1.

Theorem 1.2. (Recursive Relatedness Theorem). Let C and D be arbitrary complexity measures. Then there is a recursive function $r(x, y)$ such that $r(x, y) < r(x, y + 1)$, and for all i

$$C_i(x) \leq r(x, D_i(x)) \quad \text{a.e.}$$

and

(1.1)

$$D_i(x) \leq r(x, C_i(x)) \quad \text{a.e.}$$

[where we let $r(x, \infty) = \infty$ for all x].

Proof. Note that by the first Blum axiom

$$\begin{aligned} C_i(x) \downarrow &\quad \text{if and only if} \\ \Phi_i(x) \downarrow &\quad \text{if and only if } D_i(x) \downarrow. \end{aligned}$$

By the second Blum axiom (in the form 2'), the predicate

$$C_i(x) = y \quad \vee \quad D_i(x) = y$$

is recursive. Hence the function h defined below is recursive:

$$h(i, x, y) = \begin{cases} \max(C_i(x), D_i(x)) & \text{if } C_i(x) = y \vee D_i(x) = y \\ 0 & \text{otherwise} \end{cases}$$

Let

$$r(x, y) = y + \max_{j \leq x} \max_{z \leq y} h(j, x, z),$$

so that $r(x, y)$ is recursive. Then

$$\begin{aligned} r(x, y + 1) &= (y + 1) + \max_{j \leq x} \max_{z \leq y+1} h(j, x, z) \\ &> y + \max_{j \leq x} \max_{z \leq y} h(j, x, z) \\ &= r(x, y) \end{aligned}$$

since maximizing over a larger set of numbers cannot result in a smaller outcome. Moreover, using this same principle, and assuming that $x \geq i$

$$\begin{aligned} r(x, D_i(x)) &\geq \max_{j \leq x} \max_{z \leq D_i(x)} h(j, x, z) \\ &\geq \max_{j \leq x} h(j, x, D_i(x)) \\ &\geq h(i, x, D_i(x)) \quad (\text{since } x \geq i) \\ &= \max(C_i(x), D_i(x)) \\ &\geq C_i(x) \end{aligned}$$

Thus, the inequality

$$r(x, D_i(x)) \geq C_i(x)$$

holds for all $x \geq i$ and hence *almost everywhere*. Since the definition of h is symmetric in C and D , the same argument shows that

$$r(x, C_i(x)) \geq D_i(x) \quad \text{a.e.} \quad \blacksquare$$

As we shall see, one use of the recursive relatedness theorem is in enabling us to proceed, in some cases, from the knowledge that a theorem is true for one particular complexity measure to the truth of that theorem for all complexity measures.

2. The Gap Theorem

In this section C is *some given fixed complexity measure*. Suppose that $t(x)$ is a complexity bound. That is, assume that we are restricted to computations for which $C_i(x) \leq t(x)$ whenever $\Phi_i(x) \downarrow$. Then, in response to our complaints, the bound is increased enormously to $g(t(x))$, where g is some recursive, rapidly increasing function, e.g., $g(x) = 2^x$ [or $g(x)$ can even be the function $A(x, x)$ from Chapter 13 which grows more rapidly than any primitive recursive function]. Then, we can carry out far more computations. Right? Wrong! If the original function $t(x)$ is sufficiently tricky, it is possible that for every i , there are only finitely many values of x for which

$$C_i(x) \leq g(t(x)), \quad \text{but not } C_i(x) \leq t(x).$$

This surprising assertion is a consequence of the gap theorem:

Theorem 2.1 (Gap Theorem). Let $g(x, y)$ be any recursive function such that $g(x, y) > y$. Then, there is a recursive function $t(x)$ such that if $x > i$ and $C_i(x) < g(x, t(x))$, then $C_i(x) \leq t(x)$.

Proof. Consider the predicate

$$P(x, y) \leftrightarrow (\forall i)_{\prec_x} (C_i(x) \leq y \vee g(x, y) \leq C_i(x))$$

By the second Blum axiom, the predicate $C_i(x) \leq y$ is computable. So is the predicate

$$g(x, y) \leq C_i(x) \Leftrightarrow \sim(\exists z)_{\prec_{g(x, y)}} (z = C_i(x)).$$

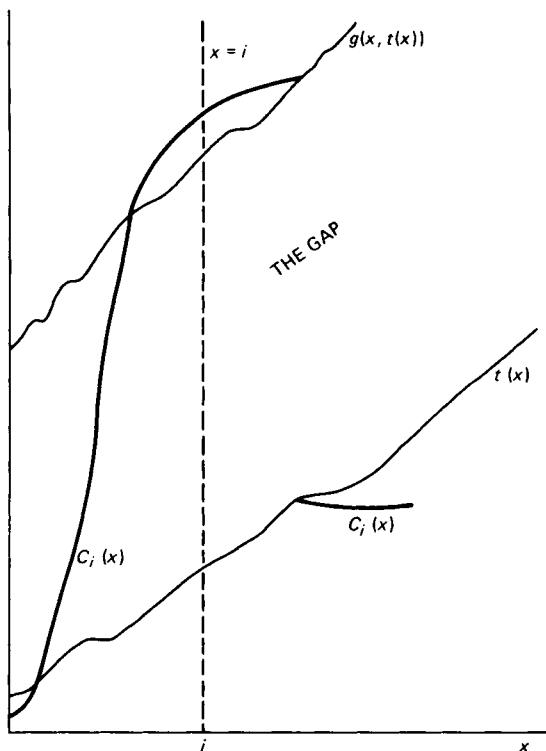


Fig. 2.1. For $x > i$, $C_i(x)$ cannot enter the “gap.”

Hence, $P(x, y)$ is also recursive. We define

$$t(x) = \min_y P(x, y), \quad (2.1)$$

so that t is a partially computable function. We will show that t is total.

Let x be a given number. Consider the set $Q = \{C_i(x) | i < x \text{ & } \Phi_i(x) \downarrow\}$. Let $y_0 = 0$ if $Q = \emptyset$ and let y_0 be the largest element of Q otherwise. We claim that $P(x, y_0)$ is true. To see this, choose $i < x$. Then if $\Phi_i(x) \downarrow$, then $C_i(x) \downarrow$ and therefore $C_i(x) \leq y_0$. If, on the other hand, $\Phi_i(x) \uparrow$ then $C_i(x) \uparrow$. Since $g(x, y_0) \downarrow$, the predicate $g(x, y_0) \leq C_i(x)$ is true. Thus, we have $P(x, y_0)$. We have shown that for every $x \in N$ there is a number y such that $P(x, y)$. Thus, $t(x)$ defined by (2.1) is total and therefore recursive.

Now let $x > i$ and $C_i(x) < g(x, t(x))$. Since $P(x, t(x))$ is true, and $i < x$, we have $C_i(x) \leq t(x) \vee g(x, t(x)) \leq C_i(x)$. But $C_i(x) < g(x, t(x))$. Hence $C_i(x) \leq t(x)$. ■

In their fine book, Machtey and Young (see *Suggestions For Further Reading*) give an amusing interpretation of the gap theorem. Let us imagine two computers, one of which is very much faster than the other. We think of each computer equipped with a reasonably efficient interpreter for our programming language \mathcal{S} so that we can speak of running a program of \mathcal{S} on one or another of the computers. Let $C_i(x)$ be the computation time of the slow computer running program number i on input x . Similarly for $D_i(x)$ and the fast computer. Clearly, C and D satisfy the Blum axioms. By the recursive relatedness theorem, there is a recursive function r satisfying (1.1). If we let $g(x, y) = r(x, y) + y + 1$, then we have $g(x, y) > y$, $g(x, y + 1) > g(x, y)$ and

$$C_i(x) \leq r(x, D_i(x)) \leq g(x, D_i(x)) \quad \text{a.e.}$$

Now let $t(x)$ satisfy the gap theorem for the complexity measure C with respect to this function g . And consider a program \mathcal{P} with number i such that $D_i(x) \leq t(x)$ a.e. That is, for sufficiently large inputs x , \mathcal{P} runs on the fast machine in time bounded by $t(x)$. Then on the slow computer, \mathcal{P} will run in time

$$C_i(x) \leq g(x, D_i(x)) \leq g(x, t(x)) \quad \text{a.e.}$$

But now the gap theorem comes into play to assure us that

$$C_i(x) \leq t(x) \quad \text{a.e.}$$

Conclusion: *Any program that runs in time $t(x)$ on the fast computer also runs in time $t(x)$ (for sufficiently large x) on the slow computer!*

3. Preliminary Form of the Speedup Theorem

Computer scientists often seek programs which will obtain a desired result using minimum resources. The speedup theorem, which is the deepest theorem in this chapter, tells us that it is possible for there to be no best program for this purpose. Roughly speaking, the theorem states that there exists a recursive function which is so badly behaved that for every program to compute it, there is another program which computes the same function but which uses much less resources. The proof of the speedup theorem is quite intricate. In this section we will prove a preliminary version. Then in the next section we will use this preliminary version to obtain the full speedup theorem. The proof of the speedup theorem will use the parameter theorem and the recursion theorem from Chapter 4 (Theorems 5.1 and 6.1).

We define a particular complexity measure $M_i(x)$ as follows: If $\Phi_i(x) \uparrow$, then $M_i(x) \uparrow$. If $\Phi_i(x) \downarrow$, then $M_i(x)$ is the largest value assumed by any variable in program number i when computing with input x . Thus $M_i(x)$ is just the complexity measure in Example 2 of Section 1. We will also work with $M_i^{(2)}(x_1, x_2)$, which is defined exactly like $M_i(x)$ except that program number i is given the pair of inputs x_1, x_2 . $M_i(x)$ and $M_i^{(2)}(x_1, x_2)$ are related by

Theorem 3.1. $M_i^{(2)}(x, y) = M_{S_1^1(y, i)}(x)$, where S_1^1 is the function defined in the parameter theorem.

Proof. Let $i = \#(\mathcal{P}_0)$. Then, examining the proof of Theorem 5.1 in Chapter 4, we see that $S_1^1(y, i) = \#(\mathcal{P})$, where \mathcal{P} is a program consisting of y copies of the instruction $X_2 \leftarrow X_2 + 1$ followed by the program \mathcal{P}_0 . The result is now obvious. ■

Our preliminary form of the speedup theorem is as follows:

Theorem 3.2. Let $g(x, y)$ be any given recursive function. Then there is a recursive function $f(x)$ such that $f(x) \leq x$ and, whenever $\Phi_i = f$, there is a j such that

$$\Phi_j(x) = f(x) \quad \text{a.e.} \quad (3.1)$$

and

$$g(x, M_j(x)) \leq M_i(x) \quad \text{a.e.} \quad (3.2)$$

To see the force of the theorem take $g(x, y) = 2^y$. Then, given $\Phi_i = f$, there is a j satisfying (3.1) such that

$$2^{M_j(x)} \leq M_i(x) \quad \text{a.e.,}$$

i.e.,

$$M_j(x) \leq \log_2 M_i(x) \quad \text{a.e.}$$

Thus program number j computes f a.e. and uses far less resources than program number i . In Section 4 we shall improve this preliminary version of the speedup theorem by eliminating the “a.e.” condition in (3.1) and by obtaining (3.2) for an arbitrary complexity measure, not merely for M .

The proof of Theorem 3.2 will use a diagonal argument, but one far more complex than we have encountered so far. Let us recall how a simple diagonal argument works. When we write

$$\bar{K} = \{n \in N \mid n \notin W_n\}$$

we know that \bar{K} is not r.e. because it differs from each r.e. set W_i with respect to the number i , namely: $i \in W_i$ if and only if $i \notin \bar{K}$. More generally, a diagonal argument constructs an object which is guaranteed not to belong to a given class by systematically ensuring that the object differs in some way from each member of the class. More intricate diagonal arguments often are carried out in an infinite sequence of stages; at each stage one seeks to ensure that the object being constructed is different from some particular member of the class. The proof of the speedup theorem is of this character. Additional examples of diagonal constructions which proceed in stages will be found in Part 5 of this book.

Proof of Theorem 3.2. We will proceed through “stages” $x = 0, 1, 2, 3, \dots$. At each stage x and for certain $n, w \in N$, we will define a set $C(n, w, x) \subseteq N$. We think of the members of $C(n, w, x)$ as numbers of programs which are *cancelled* at stage x with respect to n and w . $C(n, w, x)$ is defined recursively by the equation

$$C(n, w, x) = \left\{ i \in N \mid \begin{array}{l} w \leq i < x \text{ & } i \notin \bigcup_{y < x} C(n, w, y) \\ \& M_i(x) < g(x, M_n^{(2)}(x, i+1)) \end{array} \right\}. \quad (3.3)$$

We think of C as a 3-ary partial function on N . (The fact that the values of C are finite subsets of N instead of numbers is of no importance. Naturally, if we wished, we could use some coding device to represent each finite subset of N by a particular number.) The three conditions in (3.3) connected by “&” are to be tested in order with the understanding that if the first or second condition is false, the succeeding conditions are simply not tested. Thus we have

$$w \geq x \quad \text{implies} \quad C(n, w, x) = \emptyset \quad \text{for all } n. \quad (3.4)$$

Moreover, we have obviously

Lemma 1. If $C(n, w, y) \downarrow$ for all $y < x$ and $M_n^{(2)}(x, i+1) \downarrow$ for all i such that $w \leq i < x$, then $C(n, w, x) \downarrow$.

Indeed, when the conditions of Lemma 1 are satisfied, we can explicitly compute $C(n, w, x)$ given knowledge of $C(n, w, y)$ for $y < x$. Now clearly, when the conditions of Lemma 1 are not satisfied, $C(n, w, x) \uparrow$. Thus (3.3) can be used to give an algorithm for computing C and we may conclude that C is a partially computable function.

Lemma 2. If $i \in C(n, w, x)$, then $M_i(x) \downarrow$ and $\Phi_i(x) \downarrow$.

Proof. The truth of the condition

$$M_i(x) < g(x, M_n^{(2)}(x, i + 1))$$

implies that $M_i(x) \downarrow$, and by the Blum axioms, this implies $\Phi_i(x) \downarrow$. ■

We shall now define a 3-ary partially computable function k on N such that if $C(n, w, x) \downarrow$, then for each $i \in C(n, w, x)$, we will have $k(x, w, n) \neq \Phi_i(x)$. k is computed by using the following procedure:

Compute $C(n, w, x)$. If this computation terminates, compute $\Phi_i(x)$ for each $i \in C(n, w, x)$. [By Lemma 2, each such $\Phi_i(x) \downarrow$.] Finally, set $k(x, w, n)$ equal to the least number which is not a member of the finite set

$$\{\Phi_i(x) \mid i \in C(n, w, x)\}.$$

It is to this function k that we apply the recursion theorem. Thus, we obtain a number e such that

$$\Phi_e^{(2)}(x, w) = k(x, w, e). \quad (3.5)$$

Lemma 3. If $x \leq w$, then $k(x, w, e) = 0$.

Proof. Let $x \leq w$. By (3.4), $C(e, w, x) = \emptyset$. Hence, by definition, $k(x, w, e)$ is the least number which does not belong to \emptyset , namely, 0. ■

Lemma 4. If $k(x, w, e) \downarrow$, then $k(x, w, e) \leq x$.

Proof. The largest possible value for $k(x, w, e)$ would be obtained if the values $\Phi_i(x)$ for $i \in C(e, w, x)$ were all different and were consecutive numbers beginning with 0. In this “worst” case, there would be as many values of $\Phi_i(x)$ as in the set $C(e, w, x)$. But,

$$\begin{aligned} C(e, w, x) &\subseteq \{i \in N \mid w \leq i < x\} \\ &\subseteq \{0, 1, 2, \dots, x - 1\}. \end{aligned}$$

Thus, all the values of $\Phi_i(x)$ would be $< x$ and hence $k(x, w, e) \leq x$. ■

Lemma 5. Let $x > w$. Suppose that

$$\Phi_e^{(2)}(x, w + 1) \downarrow, \Phi_e^{(2)}(x, w + 2) \downarrow, \dots, \Phi_e^{(2)}(x, x) \downarrow \quad (3.6)$$

and

$$\Phi_e^{(2)}(0, w) \downarrow, \Phi_e^{(2)}(1, w) \downarrow, \dots, \Phi_e^{(2)}(x - 1, w) \downarrow. \quad (3.7)$$

Then, $\Phi_e^{(2)}(x, w) \downarrow$, i.e., $k(x, w, e) \downarrow$.

The reader is referred to Fig. 3.1 in connection with this lemma. In effect, Lemma 5 states that if $\Phi_e^{(2)}$ is defined along both the horizontal and vertical “pincers” shown pointing at (x, w) , then it must also be defined at (x, w) .

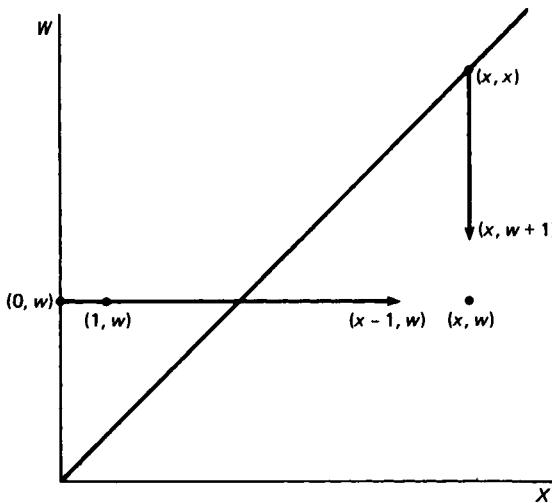


Fig. 3.1. Horizontal and vertical “pincers” pointing at (x, w) . (See Lemma 5.)

Proof of Lemma 5. By (3.7), $\Phi_e^{(2)}(y, w) \downarrow$ for all $y < x$. By definition of $k(y, w, e) = \Phi_e^{(2)}(y, w)$, we have that $C(e, w, y) \downarrow$ for all $y < x$. By (3.6), $\Phi_e^{(2)}(x, i + 1) \downarrow$ for all i such that $w \leq i < x$. Hence, likewise, $M_e^{(2)}(x, i + 1) \downarrow$ for these i . By Lemma 1, $C(e, w, x) \downarrow$. But now, by definition of k , $k(x, w, e) \downarrow$. ■

Lemma 6. $\Phi_e^{(2)}$ is total.

Proof. We shall prove by induction on x the assertion:

$$\text{For all } w, \quad \Phi_e^{(2)}(x, w) \downarrow. \quad (3.8)$$

By Lemma 3, we have

$$\Phi_e^{(2)}(0, w) = k(0, w, e) = 0,$$

which gives the result for $x = 0$. Suppose that $x > 0$, and it is known that

$$\Phi_e^{(2)}(y, w) \downarrow$$

for all $y < x$ and all w . We shall show that (3.8) then follows.

By Lemma 3, (3.8) holds for all $w \geq x$. Thus, we need only show that (3.8) holds for $w < x$. That is, it suffices to show that

$$\Phi_e^{(2)}(x, x - 1) \downarrow, \dots, \Phi_e^{(2)}(x, 0) \downarrow.$$

We will prove each of these in succession by using Lemma 5. That is, in Lemma 5, we successively set $w = x - 1, x - 2, \dots, 0$. In each case (3.7) (the horizontal “pincer”) is satisfied by the induction hypothesis. For $w = x - 1$, (3.6) only requires that $\Phi_e^{(2)}(x, x) \downarrow$, and this last follows at once from Lemma 3.

Thus by Lemma 5, $\Phi_e^{(2)}(x, x - 1) \downarrow$. But this means that (3.6) is now satisfied with $w = x - 2$. Hence once again Lemma 5 shows that $\Phi_e^{(2)}(x, x - 2) \downarrow$. Continuing, we eventually obtain $\Phi_e^{(2)}(x, 0) \downarrow$. ■

For the remainder of the proof of Theorem 3.1, we will use the notation

$$I_w = \{i \in N \mid i < w\} = \{0, 1, \dots, w - 1\}.$$

Lemma 7. $C(n, w, x) = C(n, 0, x) - I_w$.

Proof. The proof is by induction on x . $C(n, w, 0) = \emptyset$ for all n, w . Hence the result for $x = 0$ is trivially true. Suppose the result known for all $y < x$. We obtain the result for x as follows (noting $\{i \in N \mid w \leq i < x\} \cap I_w = \emptyset$):

$$\begin{aligned} C(n, w, x) &= \left\{ i \in N \mid w \leq i < x \text{ \& } i \notin \bigcup_{y < x} C(n, w, y) \right. \\ &\quad \left. \text{ \& } M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid w \leq i < x \text{ \& } i \notin \bigcup_{y < x} (C(n, 0, y) - I_w) \right. \\ &\quad \left. \text{ \& } M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid w \leq i < x \text{ \& } i \notin \bigcup_{y < x} C(n, 0, y) \right. \\ &\quad \left. \text{ \& } M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid 0 \leq i < x \text{ \& } i \notin \bigcup_{y < x} C(n, 0, y) \right. \\ &\quad \left. \text{ \& } M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} - I_w \\ &= C(n, 0, x) - I_w. \end{aligned}$$

Lemma 8. For each $w \in N$, there is a number m_w such that for all $x > m_w$, we have

$$\Phi_e^{(2)}(x, w) = \Phi_e^{(2)}(x, 0).$$

Proof. By (3.3) [the definition of $C(n, w, x)$], we have $C(e, 0, x) \cap C(e, 0, y) = \emptyset$ for $x \neq y$. [Numbers in $C(e, 0, y)$ for $y < x$ are automatically excluded

from $C(e, 0, x)$.] Hence each number in I_w belongs to at most one of the sets $C(e, 0, x)$. If we let m_w be the largest such value of x , then for $x > m_w$,

$$C(e, 0, x) \cap I_w = \emptyset.$$

Hence, using Lemma 7, for $x > m_w$,

$$C(e, w, x) = C(e, 0, x) - I_w = C(e, 0, x).$$

Hence, by the definition of the function k we have for $x > m_w$,

$$\Phi_e^{(2)}(x, w) = k(x, w, e) = k(x, 0, e) = \Phi_e^{(2)}(x, 0). \quad \blacksquare$$

Note that there is no claim being made that m_w is a computable function of w , and indeed it is not!

We are now ready to define the function $f(x)$ whose existence is asserted in Theorem 3.2. We set

$$f(x) = \Phi_e^{(2)}(x, 0).$$

Lemma 9. If $\Phi_i = f$ and $x > i$, then

$$g(x, M_e^{(2)}(x, i + 1)) \leq M_i(x).$$

Proof. Suppose otherwise. Choose the least value of $x > i$ with

$$g(x, M_e^{(2)}(x, i + 1)) > M_i(x). \quad (3.9)$$

Then we claim that for $y < x$, $i \notin C(e, 0, y)$. This is because

$$C(e, 0, y) = \left\{ j \in N \mid j < y \text{ & } j \notin \bigcup_{z < y} C(e, 0, z) \right. \\ \left. \text{ & } M_j(y) < g(y, M_e^{(2)}(y, j + 1)) \right\},$$

so that, if $i \in C(e, 0, y)$, we would have $i < y < x$, and

$$g(y, M_e^{(2)}(y, i + 1)) > M_i(y),$$

contradicting the choice of x as the least number $> i$ satisfying (3.9). Thus, we have

$$i \notin \bigcup_{y < x} C(e, 0, y).$$

Hence,

$$i \in C(e, 0, x) = \left\{ j \in N \mid j < x \text{ & } j \notin \bigcup_{y < x} C(e, 0, y) \right. \\ \left. \text{ & } M_j(x) < g(x, M_e^{(2)}(x, j + 1)) \right\}.$$

Now $k(x, 0, e)$ was defined to be different from all $\Phi_j(x)$ for which $j \in C(e, 0, x)$. Hence, $k(x, 0, e) \neq \Phi_i(x)$. But

$$k(x, 0, e) = \Phi_e^{(2)}(x, 0) = f(x) = \Phi_i(x).$$

This contradiction completes the proof. ■

Now we can complete the proof of Theorem 3.2. Let $\Phi_i = f$, and set $j = S_1^1(i + 1, e)$. Then, by Theorem 3.1 and Lemma 9, we have for $x > i$,

$$g(x, M_f(x)) = g(x, M_e^{(2)}(x, i + 1)) \leq M_i(x),$$

which proves (3.2). Finally, using the parameter theorem (Theorem 5.1 in Chapter 4) and Lemma 8 we have for $x > m_{i+1}$,

$$\Phi_j(x) = \Phi_e^{(2)}(x, i + 1) = \Phi_e^{(2)}(x, 0) = f(x),$$

which proves (3.1). ■

4. The Speedup Theorem Concluded

We will begin by showing how to eliminate the “a.e.” from Eq. (3.1) in Theorem 3.2. The technique we will use is a general one; to change a condition

$$\Phi_j(x) = f(x) \quad \text{a.e.}$$

into an equation valid everywhere, we need only modify program number j to agree with $f(x)$ at a finite number of values. We can do this by patching in a “table look-up” program. More precisely, we have

Theorem 4.1. There is a recursive function $t(u, w)$ such that

$$\begin{aligned} \Phi_{t(u, w)}(x) &= \begin{cases} \Phi_u(x) & \text{if } x > l(w) \\ (r(w))_{x+1} & \text{if } x \leq l(w), \end{cases} \\ M_{t(u, w)}(x) &= M_u(x) \quad \text{if } x > l(w). \end{aligned}$$

Here, once again we are using the pairing functions and Gödel numbers as coding devices (Chapter 3, Section 8).

Proof. Let the numbers u, w be given. Let P_u be program number u of the language \mathcal{S} , if this program begins with a labeled statement. Otherwise let P_u be program number u modified by having its initial statement labeled by a label not otherwise occurring in the program. In either case let L be the label with which P_u begins.

Let $Q_{u, w}$ be a program of \mathcal{S} which computes the primitive recursive function $(r(w))_{x+1}$, which always terminates using a branch instruction, and

which has no labels in common with P_u . Let V be a local variable which occurs neither in P_u nor in $Q_{u,w}$. Let $t(u, w)$ be the number of the program indicated in Fig. 4.1. Note that $V \leftarrow X$ is to be replaced by a suitable macro expansion as in Chapter 2 and that there are $l(w)$ statements $V \leftarrow V - 1$. Clearly this can all be done with t a recursive (even primitive recursive) function.

```


$$\begin{array}{l}
V \leftarrow X \\
V \leftarrow V - 1 \\
V \leftarrow V - 1 \\
\vdots \\
V \leftarrow V - 1
\end{array} \left. \right\} l(w)$$

```

IF $V \neq 0$ GOTO L

$Q_{u,w}$

P_u

Fig. 4.1

Now, let $x > l(w)$. Then after the $l(w)$ decrement instructions $V \leftarrow V - 1$ have been executed, V will have the value $x - l(w) > 0$. Hence, the branch shown will be taken and program P_u will be executed. Hence, $\Phi_{t(u,w)}(x) = \Phi_u(x)$. To compare the value of $M_{t(u,w)}(x)$ and $M_u(x)$ we need to be concerned about the maximum value assumed by variables in the macro expansion of $V \leftarrow X$. Examining this macro expansion as given in (c) in Chapter 2, Section 2, we see that the only possibility for a number $> x$ to arise is in the case $x = 0$. This is because local variables need to be incremented to 1 in this macro expansion in order to force a branch to be taken.² However, we are assuming $x > l(w) \geq 0$, so that $x \neq 0$. Hence, $M_{t(u,w)}(x) = M_u(x)$.

Finally, let $x \leq l(w)$. Then after $l(w)$ executions of $V \leftarrow V - 1$, V has the value 0. Thus $Q_{u,w}$ is executed. Hence, $\Phi_{t(u,w)}(x) = (r(w))_x$. ■

Now we can easily prove

Theorem 4.2. Let $g(x, y)$ be any given recursive function. Then there is a recursive $f(x)$ such that $f(x) \leq x$ and, whenever $\Phi_i = f$, there is a j such that

$$\Phi_j(x) = f(x)$$

and

$$g(x, M_j(x)) \leq M_i(x) \quad \text{a.e.}$$

² Actually, if each unconditional branch statement in program (c), Chapter 2, Section 2, is directly expanded, some of the local variables used in this expansion will reach values > 1 . The simplest way to get around this is to place the single statement $Z_2 \leftarrow Z_2 + 1$ at the beginning of this program and then to replace each of the four unconditional branch statements GOTO L by the corresponding conditional branch statement IF $Z_2 \neq 0$ GOTO L .

Proof. Let f be as in Theorem 3.2, and suppose $\Phi_i = f$. Then there is $j \in N$ such that (3.1) and (3.2) hold. Let $\Phi_j(x) = f(x)$ for $x > x_0$. Let

$$w = \langle x_0, [f(0), \dots, f(x_0)] \rangle.$$

Finally, let $\bar{j} = t(j, w)$. Then using Theorem 4.1,

$$\Phi_{\bar{j}}(x) = \Phi_{t(j, w)}(x) = \begin{cases} \Phi_j(x) & \text{if } x > x_0 \\ f(x) & \text{if } x \leq x_0, \end{cases}$$

i.e. $\Phi_{\bar{j}} = f$. Theorem 4.1 also implies that $M_{\bar{j}}(x) = M_j(x)$ a.e. Hence, using (3.2) we have almost everywhere

$$g(x, M_{\bar{j}}(x)) = g(x, M_j(x)) \leq M_i(x)$$

■

Finally, we are ready to give the speedup theorem for arbitrary complexity measures:

Theorem 4.3 (Blum Speedup Theorem). Let $g(x, y)$ be any given recursive function and let C be any complexity measure. Then there is a recursive function $f(x)$ such that $f(x) \leq x$ and whenever $\Phi_i = f$, there is a j such that

$$\Phi_j(x) = f(x)$$

and

$$g(x, C_j(x)) \leq C_i(x) \quad \text{a.e.}$$

Proof. Using the recursive relatedness theorem (Theorem 1.2), there is a recursive function $r(x, y)$ such that

$$\begin{aligned} r(x, y) &< r(x, y + 1), \\ C_i(x) &\leq r(x, M_i(x)) \quad \text{a.e.} \\ M_i(x) &\leq r(x, C_i(x)) \quad \text{a.e.} \end{aligned}$$

Let

$$h(x, y) = \sum_{z \leq y} g(x, z),$$

so that h is recursive,

$$h(x, y) \geq g(x, y),$$

and

$$h(x, y + 1) \geq h(x, y).$$

Finally, let

$$\bar{g}(x, y) = r(x, h(x, r(x, y))).$$

Now, we apply Theorem 4.2 using \bar{g} as the given function g . Let $f(x)$ be the recursive function obtained, so that $f(x) \leq x$. Let $\Phi_i = f$. Then there is a j such that $\Phi_j = f$ and

$$\bar{g}(x, M_j(x)) \leq M_i(x) \quad \text{a.e.}$$

Hence, we have, almost everywhere,

$$\begin{aligned} r(x, g(x, C_j(x))) &\leq r(x, h(x, C_j(x))) \\ &\leq r(x, h(x, r(x, M_j(x)))) \\ &= \bar{g}(x, M_j(x)) \\ &\leq M_i(x) \leq r(x, C_i(x)). \end{aligned}$$

Now, if $C_i(x) < g(x, C_j(x))$ for any value of x , we would have, for that value of x ,

$$r(x, C_i(x)) < r(x, g(x, C_j(x))).$$

Hence, we must have, almost everywhere,

$$g(x, C_j(x)) \leq C_i(x). \quad \blacksquare$$

Exercises

1. Which of the following are complexity measures?

(a) $C_i(x) = 0$ for all i, x . (That is, all computation is “free.”)

$$(b) \quad C_i(x) = \begin{cases} M_i(x) & \text{for } i \notin A \\ 0 & \text{for } i \in A, \end{cases}$$

where A is some given finite set such that Φ_i is total for all $i \in A$. (That is, the programs whose numbers belong to A can be run “free.”)

$$(c) \quad C_i(x) = 2^{\Phi_i(x)}.$$

$$(d) \quad C_i(x) = \begin{cases} M_i(x) & \text{if } i \text{ is even} \\ \text{the number of steps in computing } \Phi_i(x) & \text{if } i \text{ is odd.} \end{cases}$$

2. Prove that if C is a complexity measure and

$$D_i(x) = \begin{cases} C_i(x) & \text{for } i \notin A \\ 0 & \text{for } i \in A, \end{cases}$$

where A is as in Exercise 1(b), then D is a complexity measure.

3. Let C be an arbitrary complexity measure. Show that there is a recursive function t such that

$$\Phi_i(x) \leq t(x, C_i(x)) \quad \text{a.e.}$$

[Hint: Use the complexity measure $M_i(x)$ and the recursive relatedness theorem.]

4. Can the result of the previous problem be improved so that t is a unary recursive function such that

$$\Phi_i(x) \leq t(C_i(x)) \quad \text{a.e. ?}$$

Prove that your answer is correct.

Polynomial-Time Computability

1. Rates of Growth

In this chapter we will be working with functions f such that $f(n) \in N$ for all sufficiently large $n \in N$, but which may be undefined or have negative values for some finite number of values of n . We refer to such functions briefly, and slightly inaccurately, as functions from N to N . These functions f will typically have the additional property

$$\lim_{n \rightarrow \infty} f(n) = \infty. \quad (1.1)$$

Examples of such functions are n^2 , 2^n , and $\lfloor \log_2 n \rfloor$. It will be important for us to understand in what sense we can say that 2^n grows faster than n^2 and that n^2 grows faster than $\lfloor \log_2 n \rfloor$. Although in practice, the definitions we are about to give are only of interest for functions which satisfy (1.1), our definitions will not assume that this is the case.

Definition. Let f, g be functions from N to N . Then, we say that $f(n) = O(g(n))$ if there are numbers c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$. If these conditions do not hold we say that $f(n) \neq O(g(n))$.

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ we say that f and g have *the same rate of growth*. On the other hand, if $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$, we say that $g(n)$ grows faster than $f(n)$.

An example should help clarify these notions. We have

$$n^2 = O(3n^2 - 6n + 5)$$

since

$$\frac{n^2}{3n^2 - 6n + 5} = \frac{1}{3 - 6/n + 5/n^2} \rightarrow \frac{1}{3}$$

as $n \rightarrow \infty$, and therefore there is a number n_0 such that for all $n \geq n_0$,

$$\frac{n^2}{3n^2 - 6n + 5} \leq 1.$$

Likewise $3n^2 - 6n + 5 = O(n^2)$, so that these two functions have the same rate of growth.

Clearly, it is also true that $3n^2 - 6n + 5 = O(n^3)$; however,

$$n^3 \neq O(3n^2 - 6n + 5)$$

because

$$\frac{n^3}{3n^2 - 6n + 5} = n \cdot \frac{1}{3 - 6/n + 5/n^2} \rightarrow \infty$$

as $n \rightarrow \infty$. Thus, we can say that n^3 grows faster than $3n^2 - 6n + 5$.

More generally, we can prove

Theorem 1.1. Let f, g be functions from N to N , and let

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \beta, \quad (1.2)$$

where β is a positive real number. Then $f(n) = O(g(n))$ and $g(n) = O(f(n))$, so that f and g have the same rate of growth.

If, on the other hand,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty, \quad (1.3)$$

then $g(n) = O(f(n))$ but $f(n) \neq O(g(n))$, so that $f(n)$ grows faster than $g(n)$.

Proof. If (1.2) holds, then there is a number n_0 such that for all $n \geq n_0$,

$$\frac{f(n)}{g(n)} \leq \beta + 1.$$

Hence, $f(n) = O(g(n))$. Since (1.2) implies that

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{\beta},$$

the same reasoning can be used to show that $g(n) = O(f(n))$.

Next, (1.3) implies that

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0.$$

Therefore, there is a number n_0 such that $n \geq n_0$ implies

$$\frac{g(n)}{f(n)} \leq 1.$$

Hence, $g(n) = O(f(n))$. If we had also $f(n) = O(g(n))$, then for numbers c, n_0 we should have for $n \geq n_0$,

$$\frac{f(n)}{g(n)} \leq c;$$

on the other hand, (1.3) implies that there is a number n_1 such that $n \geq n_1$ implies

$$\frac{f(n)}{g(n)} > c,$$

which is a contradiction. ■

A *polynomial* is a function p from N to N which is defined by a formula of the form

$$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_r n^r, \quad (1.4)$$

where a_0, a_1, \dots, a_{r-1} are integers, positive, negative, or zero, while a_r is a positive integer. In this case the number r is called the *degree* of the polynomial p . The degree of a polynomial determines its rate of growth in the following precise sense:

Theorem 1.2. Let p be a polynomial of degree r . Then p and n^r have the same rate of growth. Moreover, p grows faster than n^m if $m < r$, and n^m grows faster than p if $m > r$.

Proof. Letting p be as in (1.4), we have

$$\frac{p(n)}{n^r} = \frac{a_0}{n^r} + \frac{a_1}{n^{r-1}} + \cdots + a_r \rightarrow a_r$$

as $n \rightarrow \infty$. Also,

$$\frac{p(n)}{n^m} = \frac{p(n)}{n^r} \cdot n^{r-m},$$

so that

$$\frac{p(n)}{n^m} \rightarrow \infty \quad \text{if } r > m, \quad \text{and} \quad \frac{p(n)}{n^m} \rightarrow 0 \quad \text{if } r < m.$$

The result then follows from Theorem 1.1. ■

Next we shall see that exponential functions grow faster than any fixed power.

Theorem 1.3. The function k^n , with $k > 1$, grows faster than any polynomial.

Proof. It clearly suffices to prove that for any $r \in N$,

$$\lim_{n \rightarrow \infty} \frac{k^n}{n^r} = \infty.$$

One way to obtain this result is to use L'Hospital's rule from calculus; on differentiating the numerator and denominator of this fraction r times, a fraction is obtained whose numerator approaches infinity and whose denominator is a constant (in fact, $r!$). To obtain the result directly, we first prove the lemma

Lemma. Let g be a function from N to N such that

$$\lim_{n \rightarrow \infty} \frac{g(n+1)}{g(n)} = \beta > 1.$$

Then $g(n) \rightarrow \infty$ as $n \rightarrow \infty$.

Proof of Lemma. Let γ be a number strictly between 1 and β , for example, $\gamma = (1 + \beta)/2$. Then there is a number n_0 such that $n \geq n_0$ implies

$$\frac{g(n+1)}{g(n)} \geq \gamma.$$

Thus, for each m ,

$$g(n_0 + m) \geq \gamma g(n_0 + m - 1) \geq \cdots \geq \gamma^m g(n_0).$$

Since $\gamma^m \rightarrow \infty$ as $m \rightarrow \infty$, the result follows. ■

Proof of Theorem—Concluded. Setting

$$g(n) = k^n/n^r,$$

we have

$$\frac{g(n+1)}{g(n)} = \frac{k}{\left(1 + \frac{1}{n}\right)^r} \rightarrow k \quad \text{as } n \rightarrow \infty,$$

which, by the lemma, gives the result. ■

Exercises

1. (a) Show that if $p(n)$ is defined by (1.4), then $p(n)$ is positive for n sufficiently large, so that p is a function from N to N in the sense defined at the beginning of this chapter.
(b) Show that if $p(n)$ is as in (a), then $p(n) \rightarrow \infty$ as $n \rightarrow \infty$.
2. Show that n grows faster than $\lfloor \log_2 n \rfloor$.

2. P versus NP

Computability theory has enabled us to distinguish clearly and precisely between problems for which there are algorithms and those for which there are none. However, there is a great deal of difference between solvability “in principle,” with which computability theory deals, and solvability “in practice,” which is a matter of obtaining an algorithm which can be implemented to run using space and time resources likely to be available. It has become customary to speak of problems which are solvable, not only in principle but also in practice, as *tractable*; problems which may be solvable in principle but are not solvable in practice are then called *intractable*.

The satisfiability problem, discussed in Chapter 11, is an example which is illuminating in this connection, and will, in fact, play a central role in this chapter. The satisfiability problem is certainly *solvable*; in Chapter 11, we discussed algorithms for testing a given formula in CNF for satisfiability based on truth tables, on converting to DNF, on resolution, and on the Davis–Putnam rules. However, we cannot claim that the satisfiability problem is tractable on the basis of any of these algorithms or, for that matter, on the basis of any known algorithm. As we have seen, procedures based on truth tables or DNF require a number of steps which is an *exponential* function of the length of the expression representing a given formula in CNF. It is because of the rapid growth of the exponential function that these procedures can quickly exhaust available resources. Procedures based on resolution or on the Davis–Putnam rules can be designed which work well on “typical” formulas. However, no one has succeeded in designing such a procedure for which it can be proved that exponential behavior never arises, and it is widely believed (for reasons that will be indicated later) that every possible procedure for the satisfiability problem behaves exponentially in some cases. Thus the satisfiability problem is regarded as a prime candidate for intractability, although the matter remains far from being settled.

This association of intractability with the exponential function, coupled with the fact (Theorem 1.3) that an exponential function grows faster than

any polynomial function, suggests that a problem be regarded as tractable if there is an algorithm that solves it which requires a number of steps bounded by some polynomial in the length of the input.

To make these ideas precise, we have recourse to the Turing machine model of computation as developed in Chapter 6. In particular, we shall use the terms *configuration* and *computation* as in Chapter 6.

Definition. A language L on an alphabet A is said to be *polynomial-time decidable* if there is a Turing machine \mathcal{M} which accepts L , and a polynomial $p(n)$, such that the number of steps in an accepting computation by \mathcal{M} with input x is $\leq p(|x|)$. When the alphabet is understood, we write \mathbf{P} for the class of polynomial-time decidable languages.

Definition. A function f on A^* , where A is an alphabet, is said to be *polynomial-time computable* if there is a Turing machine \mathcal{M} which computes f , and a polynomial $p(n)$, such that the number of steps in the computation by \mathcal{M} with input x is $\leq p(|x|)$.

With respect to both of these definitions, we note

1. It suffices that there exist a polynomial $p(n)$ such that the number of steps in the computation by \mathcal{M} with input x is $\leq p(|x|)$ for all but a finite number of input strings x . For, in such a case, to include the finite number of omitted cases as well, we let c be the largest number of steps used by \mathcal{M} in these cases, and replace $p(n)$ by the polynomial $p(n) + c$.
2. Using 1 and Theorem 1.2, it suffices that the number of steps be $O(|x|^r)$ for some $r \in N$.

The discussion leading to these definitions suggests that in analogy with Church's thesis, we consider the

Cook–Karp Thesis. The problem of determining membership of strings in a given language L is tractable if and only if $L \in \mathbf{P}$.

The evidence supporting the Cook–Karp thesis is much weaker than that supporting Church's thesis. Nevertheless, it has gained wide acceptance. Later, we shall discuss some of the reasons for this.

The following simple result is quite important:

Theorem 2.1. Let $L \in \mathbf{P}$, let f be a polynomial-time computable function on A^* , and let $Q = \{x \in A^* \mid f(x) \in L\}$. Then $Q \in \mathbf{P}$.

Proof. Let \mathcal{M} accept L using a number of steps which is $O(|x|^r)$, and let \mathcal{N} compute $f(x)$ in a number of steps which is $O(|x|^s)$. A Turing machine \mathcal{R} that accepts Q is easily constructed that, in effect, first runs \mathcal{N} on x to compute $f(x)$ and then runs \mathcal{M} on $f(x)$ to determine whether $f(x) \in L$. Since a

Turing machine cannot print more symbols in the course of a computation then there are steps in that computation, we have

$$|f(x)| \leq |x| + p(|x|), \quad \text{where } p(n) = O(n^s).$$

By Theorem 1.2, it follows that $|f(x)| = O(|x|^s)$. Hence, the number of steps required by \mathcal{R} on input x is $O(|x|^s)$. ■

Theorem 2.2. Let f, g be polynomial-time computable functions such that $h(x) = f(g(x))$ is total. Then h is polynomial-time computable.

Proof. The proof is similar to that of the previous theorem. ■

It has turned out to be extremely difficult to prove that specific languages do not belong to **P**, although there are many likely candidates. An important example is the satisfiability problem discussed in Chapter 11. To make matters definite, we assume a set of atoms $\mathcal{A} = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$, where each atom is a single symbol. We use the symbols

$$\alpha_1, \alpha_2, \dots, \alpha_n, \bar{\alpha}_1, \bar{\alpha}_2, \dots, \bar{\alpha}_n,$$

for the atoms and their negations, simply using concatenation for disjunction. Finally, we use the symbol / to begin a clause. Then, any string on the alphabet

$$C = \{\alpha_1, \alpha_2, \dots, \alpha_n, \bar{\alpha}_1, \bar{\alpha}_2, \dots, \bar{\alpha}_n, /\}$$

which begins “/” stands for a CNF formula (where in the interest of simplicity we are permitting empty and tautologous clauses, and repetitions of literals in a clause). Thus the CNF formula

$$(p \vee q \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r})$$

from Chapter 11 could be written as

$$/\alpha_1 \alpha_2 \bar{\alpha}_3 \alpha_4 / \bar{\alpha}_2 \bar{\alpha}_1 \bar{\alpha}_3 \alpha_4 / \bar{\alpha}_2 \bar{\alpha}_1 \bar{\alpha}_3.$$

Any string in C^* which ends “/” or in which “/” is repeated represents a CNF formula which contains the empty clause, and hence is unsatisfiable.

Now, we write SAT for the language consisting of all elements of C^* which represent satisfiable CNF formulas. In spite of a great deal of attention to the question, it is still not known whether $SAT \in P$. The starting point of the work on computational complexity that we discuss in this chapter is the observation that the situation changes entirely when we shift our attention from deterministic to nondeterministic computation. Nondeterministically one can discover very rapidly that a formula is satisfiable; it is only necessary that the satisfying assignment be “guessed.” That is, instead of constructing an entire truth table, it suffices to construct a single row. To make these ideas

precise, we have recourse to nondeterministic Turing machines as discussed in Chapter 6, Section 5.

Definition. A language L is said to belong to the class **NP** if there is a nondeterministic Turing machine \mathcal{M} which accepts L , and a polynomial $p(n)$, such that for each $x \in L$, there is an accepting computation $\gamma_1, \gamma_2, \dots, \gamma_m$ by \mathcal{M} for x with $m \leq p(|x|)$.

We then have readily

Theorem 2.3. $P \subseteq NP$. If $L \in NP$, then L is recursive.

Proof. The first inclusion is obvious, since an ordinary Turing machine is a nondeterministic Turing machine.

For the rest, let $L \in NP$, let \mathcal{M} be a nondeterministic Turing machine which accepts L , with corresponding polynomial $p(n)$. We set γ_1 to be the configuration

$$\begin{array}{c} s_0 x \\ \uparrow \\ q_1 \end{array}$$

Next, by examining the quadruples of \mathcal{M} , we find all configurations γ_2 such that $\gamma_1 \vdash \gamma_2$. Continuing in this manner, we determine all possible sequences $\gamma_1, \gamma_2, \dots, \gamma_m$ with $m \leq p(|x|)$ such that

$$\gamma_1 \vdash \gamma_2 \vdash \cdots \vdash \gamma_m.$$

Then, $x \in L$ if and only if at least one of these sequences is an accepting computation by \mathcal{M} for x . This gives an algorithm for determining whether $x \in L$, and so, invoking Church's thesis, we conclude that L is recursive. (Methods like those used in Chapter 7 could be used to prove that L is recursive without using Church's thesis.) ■

In line with our discussion of the satisfiability problem viewed nondeterministically, we can prove

Theorem 2.4. $SAT \in NP$.

Proof. We will construct a nondeterministic Turing machine \mathcal{M} that accepts SAT. \mathcal{M} will begin by verifying that the input string begins “/”; if not \mathcal{M} will enter an infinite loop. Then, successively for $i = 1, 2, \dots, n$, it nondeterministically selects a value TRUE or FALSE for the atom α_i . (This is the only element of nondeterminism involved.) Then it moves from left to right over the input string looking for occurrences of α_i if the value TRUE was selected and for occurrences of $\bar{\alpha}_i$ if the value FALSE was selected. If the

symbol being sought is found so that the clause in which it is located has been satisfied, the marker at the beginning of that clause is changed from “/” to “|.” Finally, \mathcal{M} looks for any remaining “/.” If there are none, all the clauses have been satisfied and \mathcal{M} enters a terminal configuration; otherwise it enters an infinite loop.

We write $\tilde{C} = C \cup \{|, B\}$, where as usual, $s_0 = B$ is the blank. The quadruples of \mathcal{M} which serve to check that the input string begins “/” are

$$\begin{aligned} q_1 & B R q_1, \\ q_1 & / L m_1, \\ q_1 & s L q_1, \quad s \in C - \{/ \}. \end{aligned}$$

Next we list the quadruples which select a value for $\alpha_i, i = 1, 2, \dots, n$, and check which clauses are thereby satisfied. We shall use “+” as a superscript on state symbols to indicate a TRUE value, and “−” to indicate a FALSE value. Where no confusion will result, we write quadruples in pairs using the superscript “±” to exhibit both cases.

$m_i \ B R r_i^+$	Select value TRUE or FALSE for α_i .
$m_i \ B R r_i^- \}$	
$r_i^\pm \mid R v_i^\pm$	Clause already satisfied; look for next clause.
$r_i^\pm \ / R r_i^\pm$	Clause not yet satisfied; look for selected literal.
$r_i^+ \ s R r_i^+, \ s \in \tilde{C} - \{\alpha_i, , /, B\}$	
$r_i^- \ s R r_i^-, \ s \in \tilde{C} - \{\bar{\alpha}_i, , /, B\}$	
$r_i^+ \ \alpha_i L w_i^+ \}$	Satisfying literal found.
$r_i^- \ \bar{\alpha}_i L w_i^- \}$	
$w_i^\pm \ s L w_i^\pm, \ s \in C - \{/ \} \}$	Mark satisfied clause.
$w_i^\pm \ / \mid v_i^\pm \}$	
$v_i^\pm \ s R v_i^\pm, \ s \in \tilde{C} - \{/, B\} \}$	Find next unsatisfied clause.
$v_i^\pm \ / R r_i^\pm \}$	
$r_i^\pm \ B L m_{i+1} \}$	Prepare to select value for next atom.
$v_i^\pm \ B L m_{i+1} \}$	
$m_{i+1} \ s L m_{i+1}, \ s \in \tilde{C} - \{B\}$	Find left end of input string.

After selecting a value for each of the atoms $\alpha_1, \dots, \alpha_n$, and marking the clauses which are thereby satisfied, \mathcal{M} finds itself in state m_{n+1} , scanning the B just to the left of the string which represents the CNF formula, with all satisfied clauses marked “|.” The selected assignment satisfies the given

CNF formula, just in case no occurrences of the symbol “/” remain. Hence, it suffices to place in \mathcal{M} the quadruples:

$$\begin{aligned} m_{n+1} & B \ R \ p, \\ p \ s & \ R \ p, \quad s \in \tilde{C} - \{/, B\}, \\ p & / \ L \ \bar{p}, \\ p \ s & \ R \ p, \quad s \in \tilde{C}. \end{aligned}$$

If the input string has been satisfied, \mathcal{M} will arrive at the blank on the right in state p , and hence will be in a terminal configuration; i.e., the input string will be accepted.

It remains to count the number of configurations \mathcal{M} will pass through in accepting an input string x . The check that x begins with the symbol “/” requires 3 configurations including the initial one. By counting the total number of passes over the input string required, we then get an upper bound of $(2n + 2)(|x| + 2) + 3$, a polynomial in $|x|$ of degree 1. Thus \mathcal{M} accepts SAT, not only in polynomial time, but actually in “linear” time. ■

It is natural to ask whether the inclusion $\mathbf{P} \subseteq \mathbf{NP}$ is proper, i.e., whether there is a language L such that $L \in \mathbf{NP} - \mathbf{P}$. As we shall see, using the notion of **NP-completeness** to be defined below, it can be shown that if there were such a language, then it would follow that $SAT \in \mathbf{NP} - \mathbf{P}$. Unfortunately, this remains an open question.

Definition. Let L, Q be languages, then we write

$$Q \leq_p L,$$

and say that Q is *polynomial-time reducible* to L , if there is a polynomial-time computable function f such that

$$x \in Q \Leftrightarrow f(x) \in L.$$

Theorem 2.5. Let $R \leq_p Q$ and $Q \leq_p L$. Then $R \leq_p L$.

Proof. This follows at once from Theorem 2.2. ■

Definition. A language L is called **NP-hard** if for every $Q \in \mathbf{NP}$, we have $Q \leq_p L$. L is called **NP-complete** if $L \in \mathbf{NP}$ and L is NP-hard.

The significance of NP-completeness can be appreciated from the following result:

Theorem 2.6. If there is an NP-complete language L such that $L \in \mathbf{P}$, then $\mathbf{NP} = \mathbf{P}$.

Proof. We need to show that if $Q \in \text{NP}$, then $Q \in \text{P}$. Let $Q \subseteq A^*$. Since L is **NP-hard**, there is a polynomial-time computable function f such that

$$Q = \{x \in A^* \mid f(x) \in L\}.$$

The result now follows from Theorem 2.1. ■

Intuitively, one can thus think of the **NP-complete** languages as the “hardest” languages in **NP**. As we shall see in the next section, SAT is **NP-complete**. Thus, if it should turn out that $SAT \in \text{P}$, then every **NP-complete** problem would also be in **P**. It is considerations like these that have led to the tentative conclusion that **NP-complete** problems should be regarded as being intractable. To date, however, although very many problems are known to be **NP-complete**, there is no language known to be in $\text{NP} - \text{P}$, and it thus remains possible that $\text{NP} = \text{P}$.

Exercises

1. Show that $L \leq_p M$ does not necessarily imply that $M \leq_p L$.
2. Let A be an alphabet and set

$$\text{co-}\text{NP} = \{L \subseteq A^* \mid A^* - L \in \text{NP}\}.$$

Show that if there is a language L such that L is **NP-complete** and $L \in \text{co-}\text{NP}$, then $\text{NP} = \text{co-}\text{NP}$.

3. Cook's Theorem

We now prove the main theorem of this chapter:

Theorem 3.1 (Cook). SAT is **NP-complete**.

Proof. Since we know, by Theorem 2.4, that $SAT \in \text{NP}$, it remains to show that SAT is **NP-hard**. That is, we need to show that if $L \in \text{NP}$, then $L \leq_p \text{SAT}$. Thus, let $L \in \text{NP}$, and let \mathcal{M} be a nondeterministic Turing machine that accepts L , with $p(n)$ the polynomial which furnishes a bound on the number of steps \mathcal{M} requires to accept an input string. Without loss of generality, we assume that $p(n) \geq n$ for all n . We must show that there is a polynomial-time computable function which translates any input string u for \mathcal{M} into a CNF formula δ_u such that u is accepted by \mathcal{M} if and only if δ_u is satisfiable. For a given input u , let $t = p(|u|)$.

We know that if \mathcal{M} accepts input u , it does so in $\leq t$ steps. Therefore, in order to determine whether \mathcal{M} accepts u , we need only run it on u for at most

t steps and check to see whether the final configuration is terminal. Since at each step of the computation, \mathcal{M} can move at most one square to the left or right of the square currently being scanned, it follows that after t steps, the scanned square can be at most t squares to the left or t squares to the right of its original position. Since we have chosen the polynomial $p(n)$ so that $t \geq |u|$, for our present purposes it suffices to consider $2t + 1$ squares of tape. Thus, since we are considering only t steps of the computation, we can completely exhibit all of the information on \mathcal{M} 's tape, using a t by $(2t + 1)$ array (see Fig. 3.1).

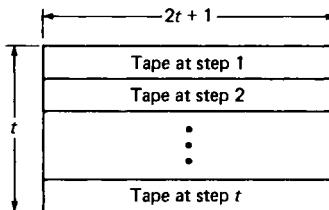


Fig. 3.1

The first line of this array, corresponding to the initial tape contents, will then have the form

$$s_0^{[t+1]} u s_0^{[t-|u|]}$$

where \mathcal{M} begins in state q_1 scanning the $(t + 1)$ th symbol in this string, the s_0 immediately preceding u .

We will find it convenient, in this proof, to use the Turing machine model used in Theorem 4.2 in Chapter 6, in which acceptance of an input is by arrival in a unique accepting state q_m . We assume, therefore, that \mathcal{M} is a Turing machine of this type. Let the set of states of \mathcal{M} be $Q = \{q_1, q_2, \dots, q_m\}$ and let the set of tape symbols be $S = \{s_0, s_1, \dots, s_r\}$. It will simplify matters if we only need to check configuration number t to determine acceptance. Thus, we alter our definition of accepting computation to permit any number of repetitions of consecutive configurations; hence we may assume that our accepting computation consists of *exactly* t steps.

Our goal is to define a CNF formula δ_u which is satisfiable if and only if u is accepted by \mathcal{M} . Our set of atoms will be

$$\mathcal{A} = \{\rho_{h,j,k}, \sigma_{i,j,k} \mid 1 \leq h \leq m, 0 \leq i \leq r, 1 \leq j \leq 2t + 1, 1 \leq k \leq t\}.$$

We first assume that u is accepted by \mathcal{M} , so that we have an accepting computation by \mathcal{M} for u . We assume that the above t by $2t + 1$ array has been constructed correspondingly. We will construct the CNF formula δ_u so that $\delta_u^v = 1$, where v is the assignment on \mathcal{A} defined by

$$v(\rho_{h,j,k}) = \begin{cases} 1 & \text{if } \mathcal{M} \text{ is in state } q_h \text{ scanning the } j\text{th position at the } k\text{th} \\ & \text{step of the computation} \\ 0 & \text{otherwise,} \end{cases} \quad (3.1)$$

$$v(\sigma_{i,j,k}) = \begin{cases} 1 & \text{if tape symbol } s_i \text{ is in the } j\text{th position of the } k\text{th row of} \\ & \text{the array} \\ 0 & \text{otherwise.} \end{cases}$$

In constructing δ_u , we will find the following abbreviation useful:

$$\nabla\{x_e \mid 1 \leq e \leq l\} = \bigwedge_{1 \leq e < f \leq l} (\neg x_e \vee \neg x_f) \wedge \bigvee_{1 \leq e \leq l} x_e,$$

where $\{x_e \mid 1 \leq e \leq l\}$ is a set of formulas. Thus,

$$\nabla\{x_e \mid 1 \leq e \leq l\} \quad (3.2)$$

is a formula whose value is TRUE (i.e., 1) under a given assignment if and only if *exactly* one of the formulas x_1, x_2, \dots, x_l has the value TRUE under that assignment. In the particular case that x_1, x_2, \dots, x_l are atoms, (3.2) is a CNF formula. We will need to calculate $|\nabla\{x_e \mid 1 \leq e \leq l\}|$ in this case. Formula (3.2) contains a clause consisting of two literals for each pair (e, f) with $1 \leq e < f \leq l$, followed by a single clause of l literals. Since there are $l(l - 1)/2$ such pairs (e, f) , and since in our notation, with “/” being used to separate clauses, each clause is of length 1 plus the number of its literals, we have

$$|\nabla\{x_e \mid 1 \leq e \leq l\}| = \frac{l(l - 1)}{2} \cdot 3 + (l + 1) = O(l^2).$$

Let us write

$$u = s_{u_1}s_{u_2} \cdots s_{u_z}, \quad \text{where } |u| = z.$$

We present a sequence of CNF formulas whose conjunction δ_u (which is then also a CNF formula) may be thought of as simulating the behavior of \mathcal{M} in accepting u . Each of these formulas has the value TRUE under the assignment v . We precede each formula with an English sentence in quotes which may be thought of as expressing a corresponding property of the accepting computation by \mathcal{M} for u ; each such sentence is intended to make it clear that the corresponding formula is indeed true under the assignment v . In some cases the formula as written will not be in CNF; in these cases the formula written is intended to stand for a formula in CNF obtained from it by using the methods of Chapter 11, Section 3.

- (1) “The initial configuration has tape contents corresponding to the first row of the array, with \mathcal{M} in state q_1 scanning the symbol s_0 immediately to the left of the first symbol of u .”

$$\bigwedge_{0 < j \leq t+1} \sigma_{0,j,1} \wedge \bigwedge_{0 < j \leq z} \sigma_{u_j,t+j+1,1} \wedge \bigwedge_{0 < j \leq t-z} \sigma_{0,i+z+j+1,1} \wedge \rho_{1,t+1,1}.$$

Taking into account the $/$ s needed to separate these unit clauses, we see that this expression is of length $4t + 4 = O(t)$.

- (2) “At each step of the computation there is a unique state and a unique scanned square.”

$$\bigwedge_{1 \leq k \leq t} \nabla\{\rho_{h,j,k} \mid 1 \leq h \leq m, 1 \leq j \leq 2t+1\}.$$

By the above remarks, the length of this expression is $O(t^3)$.

- (3) “Each entry of the array contains exactly one symbol.”

$$\bigwedge_{1 \leq k \leq t} \bigwedge_{1 \leq j \leq 2t+1} \nabla\{\sigma_{i,j,k} \mid 0 \leq i \leq r\}.$$

r is a constant, so that this expression is of length $O(t^2)$.

- (4) “Each configuration in the computation, after the first, is identical to the preceding configuration, or is obtained from it by applying one of the quadruples of \mathcal{M} .”

This formula will be the most complicated. Let the quadruples of \mathcal{M} be as follows:

$$\{q_{i_a} s_{j_a} s_{k_a} q_{l_a} \mid a = 1, 2, \dots, \bar{a}\}, \quad (3.3a)$$

$$\{q_{i_b} s_{j_b} R q_{l_b} \mid b = 1, 2, \dots, \bar{b}\}, \quad (3.3b)$$

$$\{q_{i_c} s_{j_c} L q_{l_c} \mid c = 1, 2, \dots, \bar{c}\}. \quad (3.3c)$$

To make the formula easier to understand, we write it in the form

$$\begin{aligned} & \bigwedge_{1 \leq k < t} \bigwedge_{1 \leq j \leq 2t+1} (\text{NOTHEAD}(j, k) \vee \text{IDENT}(j, k) \\ & \quad \vee A(j, k) \vee B(j, k) \vee C(j, k)), \end{aligned}$$

where each of these five disjuncts will be explained below. It will turn out that each disjunct has constant length; hence we may conclude that the length of the entire formula will be $O(t^2)$.

We define

$$\text{NOTHEAD}(j, k) = \bigvee_{0 \leq i \leq r} (\sigma_{i,j,k} \wedge \sigma_{i,j,k+1}) \wedge \bigwedge_{1 \leq h \leq m} \neg \rho_{h,j,k},$$

so that $\text{NOTHEAD}(j, k)^v = 1$ for given j, k if and only if \mathcal{M} is not scanning the j th position at the k th step of the computation.

Next we set

$$\text{IDENT}(j, k) = \bigvee_{1 \leq h \leq m} \bigvee_{0 \leq i \leq r} (\rho_{h,j,k} \wedge \sigma_{i,j,k} \wedge \rho_{h,j,k+1} \wedge \sigma_{i,j,k+1}),$$

so that $\text{IDENT}(j, k)^v = 1$ for given j, k if and only if \mathcal{M} is scanning the j th position at both the k th and the $(k + 1)$ th steps of the computation, and both the state and the symbol are the same in both of these configurations.

Next,

$$A(j, k) = \bigvee_{1 \leq a \leq \bar{a}} (\rho_{i_a,j,k} \wedge \sigma_{j_a,j,k} \wedge \sigma_{k_a,j,k+1} \wedge \rho_{l_a,j,k+1}),$$

where $A(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3a).

Similarly, we will define $B(j, k)$ so that $B(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3b). For $j \neq 2t + 1$, we can define

$$B(j, k) = \bigvee_{1 \leq b \leq \bar{b}} (\rho_{i_b,j,k} \wedge \sigma_{j_b,j,k} \wedge \sigma_{j_b,j,k+1} \wedge \rho_{l_b,j+1,k+1}).$$

This definition will not work for $j = 2t + 1$ because there are no atoms $\rho_{h,2t+2,k}$. But since the computation can not proceed beyond the boundaries of our array, it suffices to take $B(2t + 1, k)$ to be any unsatisfiable formula, e.g., the empty clause.

Finally, we will define $C(j, k)$ so that $C(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3c). For $j \neq 1$, we can define

$$C(j, k) = \bigvee_{1 \leq c \leq \bar{c}} (\rho_{i_c,j,k} \wedge \sigma_{j_c,j,k} \wedge \sigma_{j_c,j,k+1} \wedge \rho_{l_c,j-1,k+1}).$$

This definition will not work for $j = 1$ because there are no atoms $\rho_{k,0,k}$. But since the computation can not proceed beyond the boundaries of our array, it suffices to let $C(1, k)$ be any unsatisfiable formula, e.g., the empty clause.

(5) “The t th configuration is a terminal configuration.” Equivalently, “At the t th step, \mathcal{M} is in state q_m .”

$$\bigvee_{1 \leq j \leq 2t+1} \rho_{m,j,t}.$$

This expression is clearly of length $O(t)$.

Now, we take δ_u to be simply the conjunction of the CNF formulas (1)–(5) above. It is clear from what has already been said that if \mathcal{M} accepts u , then δ_u is satisfiable; in fact, $\delta_u^v = 1$.

Conversely, let v be an assignment such that $\delta_u^v = 1$. We will show that \mathcal{M} accepts u . By (3), we see that for each $1 \leq j \leq 2t + 1, 1 \leq k \leq t$, there is a

unique i such that $v(\sigma_{i,j,k}) = 1$. Hence we can uniquely reconstruct our t by $2t + 1$ array. By (2), for each row of the array there is a unique state q_h and position j in the row such that $v(\rho_{h,j,k}) = 1$. Thus, each row can be made into a configuration of \mathcal{M} so that (3.1) is satisfied. By (1), the configuration corresponding to the first row of the array is an initial configuration for \mathcal{M} with input u . By (4), for each row of the array after the first, the corresponding configuration is identical to it or results from it using one of the quadruples of \mathcal{M} . Finally, by (5) the entire sequence of configurations constitutes an accepting computation by \mathcal{M} for u . Thus, u is accepted by \mathcal{M} .

It remains to be shown that there is a polynomial-time computable function which maps each string u onto the corresponding CNF formula δ_u . Now, the CNF formulas of (2)–(5) do not depend on u and a Turing machine can easily be constructed to write these on a tape in a number of steps proportional to the length of the expression, which, as we have seen, is $O(t^3)$, and hence polynomial in $|u|$. It remains to consider (1), which is a conjunction of atoms. Some of these atoms do not depend on u ; producing this part of (1) simply involves writing $O(t)$ symbols. The remaining atoms of (1) correspond in a one-one manner to the symbols making up u ; they can obviously be produced by a Turing machine in a number of steps proportional to $|u|$. This completes the proof. ■

Using Theorem 2.6, we have at once

Corollary 3.2. $\mathbf{P} = \mathbf{NP}$ if and only if $\text{SAT} \in \mathbf{P}$.

4. Other NP-Complete Problems

The principal technique for proving a problem to be **NP**-complete is given by the following result:

Theorem 4.1. Let Q be an **NP**-complete problem, and let $Q \leq_p L$. Then L is **NP**-hard.

Proof. Let R be any language such that $R \in \mathbf{NP}$. Since Q is **NP**-complete, we have $R \leq_p Q$. By Theorem 2.5, $R \leq_p L$. Thus, L is **NP**-hard. ■

Corollary 4.2. Let Q be an **NP**-complete problem, let $L \in \mathbf{NP}$, and let $Q \leq_p L$. Then L is **NP**-complete.

Thus, once it has been shown that a problem is **NP**-complete, it can be used to show that other problems are **NP**-complete. In this way many problems have been shown to be **NP**-complete. It is this fact that constitutes

the main evidence for regarding NP-complete problems as being intractable. Since the existence of a polynomial-time algorithm for even a single one of these problems would imply that there is a polynomial-time algorithm for every one of them, and, since it is argued that it is most unlikely that this could be the case without even one of these algorithms having been discovered, it is concluded that in all likelihood none of these problems have polynomial-time algorithms, and so they should all be regarded as intractable.

We will present a very small sample of this work, showing that a few problems are NP-complete. We begin with a restricted form of the satisfiability problem.

The 3-SAT problem is to determine whether a formula in CNF in which no clause contains more than three literals, is satisfiable. We show that 3-SAT is NP-complete by showing that any CNF formula ζ can be transformed in polynomial time to a CNF formula ζ' containing at most three literals per clause such that ζ is satisfiable if and only if ζ' is satisfiable.

Theorem 4.3. 3-SAT is NP-complete.

Proof. Since 3-SAT is a special case of SAT, and SAT is in NP, it follows that 3-SAT is in NP. Let

$$/\alpha_1\alpha_2 \cdots \alpha_k, \quad k \geq 4, \quad (4.1)$$

be any one of the clauses of ζ containing more than three literals. Let $\beta_1, \beta_2, \dots, \beta_{k-3}$ be atoms which do not appear in ζ . We construct ζ' by replacing (4.1) by the conjunction

$$/\alpha_1\alpha_2\beta_1/\alpha_3\beta_1\beta_2/\alpha_4\beta_2\beta_3/\cdots/\alpha_{k-2}\beta_{k-4}\beta_{k-3}/\alpha_{k-1}\alpha_k\beta_{k-3}.$$

It is easy to see that ζ is satisfiable if and only if ζ' is satisfiable. Moreover, since the length of ζ' is bounded by a constant times the length of ζ , the transformation can be performed in linear time. ■

It is interesting that there are problems which superficially appear to be unrelated, but between which we can readily find a polynomial time transformation. Our next example is known as the COMPLETE-SUBGRAPH problem. A *graph* G consists of a finite nonempty set of *vertices* $V = \{v_1, \dots, v_n\}$ and a finite set of *edges* E . Each edge is a pair of vertices. The size of the graph is simply the number of vertices it contains. A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$, and $E' \subseteq E$. A graph $G = (V, E)$ is *complete* if there is an edge in E between every pair of distinct vertices in V .

The COMPLETE-SUBGRAPH problem is: given a graph and a number k , does the graph have a complete subgraph of size k ?

Theorem 4.4. COMPLETE-SUBGRAPH is NP-complete.

Proof. We show informally that COMPLETE-SUBGRAPH is in NP. Let the number k and a list of the vertices and of the edges of the given graph be written on the tape of a Turing machine in any reasonable notation. The procedure begins by nondeterministically selecting a vertex and then decrementing k . By continuing this process until k has been decremented to 0, a list of k vertices is obtained. The procedure then tests [in time $O(k^2)$] whether the graph has a complete subgraph in those vertices. Since

$$k \leq n \leq \text{length of the string representing } G \text{ on the tape},$$

where G is the given graph, this shows that COMPLETE-SUBGRAPH \in NP.

To show that COMPLETE-SUBGRAPH is NP-hard, we show that $SAT \leq_p$ COMPLETE-SUBGRAPH. Thus, we must show how to map each CNF formula γ into a pair consisting of a number k and a graph G so that γ is satisfiable if and only if G has a complete subgraph of size k . If $\gamma = /y_1/y_2 \cdots /y_k$ is a CNF formula, where y_1, y_2, \dots, y_k are clauses, then we take the number k to be simply the number of clauses in γ and construct the graph $G = (V, E)$, where

$$V = \{(\alpha, i) | \alpha \text{ is a literal in } y_i\},$$

$$E = \{((\alpha, i), (\beta, j)) | \alpha \neq \neg\beta \text{ and } i \neq j\}.$$

Thus we have a vertex for each occurrence of each literal in γ . Edges join pairs of vertices which represent literals in different clauses provided one is not the negation of the other. This means that these literals can both be assigned the value “TRUE” at the same time. If γ is satisfiable, there is some way to assign truth values to the atoms so that γ evaluates to “TRUE.” Thus at least one literal of each clause of γ must be assigned the value “TRUE,” and in G there will be an edge connecting each pair of “true literals.” This means that the nodes of G corresponding to the “true literals” of γ form a complete subgraph of size k . Conversely, if γ contains a complete subgraph of size k , then since edges join pairs of literals in different clauses which can be true at the same time, there is a way to make each clause of γ true at the same time. Thus γ is satisfiable. Furthermore, G can clearly be obtained from γ by a polynomial-time computable function. ■

A *clique* in a given graph is a maximal complete subgraph of that graph; that is a clique is a complete subgraph of a given graph which is *not* a subgraph of any other complete subgraph of that graph. The MAX-CLIQUE problem is to find the size of the largest clique in a given graph. Of course, in this form, MAX-CLIQUE is not a language but rather a function, and so it does not make sense in terms of our definitions to ask whether it is in NP.

However, since removing a vertex and all edges containing it from a complete subgraph yields another complete subgraph, we see that any algorithm for the MAX-CLIQUE problem which could actually be implemented using reasonable resources could easily be transformed into an equally usable algorithm for the COMPLETE-SUBGRAPH problem. Hence, to the extent that NP-completeness can be regarded as implying intractability, we are entitled to conclude that MAX-CLIQUE is likewise intractable.

We next consider a closely related graph-theoretic problem, known as VERTEX-COVER. A set S is a *vertex cover* for a graph $G = (V, E)$ if $S \subseteq V$ and for every $(x, y) \in E$, either $x \in S$ or $y \in S$. The VERTEX-COVER problem is to determine for a given graph G and integer k whether G has a vertex cover of size k .

Theorem 4.5. Let $G = (V, E)$ be a graph and let

$$E' = \{(x, y) | x, y \in V, x \neq y, \text{ and } (x, y) \notin E\}.$$

Let us consider the graph $G' = (V, E')$ (sometimes called the *complement graph* of G). Then $S \subseteq V$ is the set of vertices of a complete subgraph of G if and only if $V - S$ is a vertex cover in G' .

Proof. Let S be the set of vertices of a complete subgraph of G . Then, by definition, for any $(x, y) \in E'$, either $x \in V - S$ or $y \in V - S$. Thus, $V - S$ is a vertex cover of G' . Conversely, if $V - S$ is a vertex cover of G' , then for any $(x, y) \in E'$, either $x \in V - S$ or $y \in V - S$. Thus no edge of G' connects two vertices in S . Thus for every $u, v \in S, u \neq v$, we have $(u, v) \in E$, and so S is the set of vertices of a complete subgraph of G . ■

Corollary 4.6. VERTEX-COVER is NP-complete.

The SET-COVER problem is to determine for a family of sets $\Delta = \{S_1, S_2, \dots, S_n\}$, and number k , whether there exists a subfamily Γ of Δ of size k , $\Gamma = \{S_{m_1}, S_{m_2}, \dots, S_{m_k}\}$, such that

$$\bigcup_{i \leq n} S_i = \bigcup_{j \leq k} S_{m_j}.$$

Corollary 4.7. SET-COVER is NP-complete.

Proof. Let $G = (V, E)$ be a graph with $V = \{v_1, v_2, \dots, v_n\}$. For $i = 1, 2, \dots, n$, let

$$S_i = \{(v_i, v_j) | (v_i, v_j) \in E\} \cup \{(v_j, v_i) | (v_i, v_j) \in E\}.$$

Clearly $\Gamma = \{S_{i_1}, S_{i_2}, \dots, S_{i_k}\}$ is a set cover for $\Delta = \{S_1, S_2, \dots, S_n\}$ if and only if $\{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$ is a vertex cover for G . ■

Exercises

1. The CHROMATIC-NUMBER problem is to determine for a given graph $G = (V, E)$ and integer k whether there is a function f from V to $\{1, 2, \dots, k\}$ such that if $(x, y) \in E$, then $f(x) \neq f(y)$. (Intuitively, this problem amounts to determining whether or not it is possible to “color” the vertices of G using k colors in such a way that no two adjacent vertices are colored the same.) Show that CHROMATIC-NUMBER is NP-complete. (*Hint:* Show $3\text{-SAT} \leq_p \text{CHROMATIC-NUMBER}$.) (*Further hint:* Assume $\gamma = / \gamma_1 / \gamma_2 \cdots / \gamma_m$ is a CNF formula such that no γ_i contains more than three literals. Assume there are n atoms $\alpha_1, \alpha_2, \dots, \alpha_n$ which appear either negated or unnegated in γ . Construct a graph G with $3n + m$ vertices such that G is $n + 1$ colorable if and only if γ is satisfiable.)

2. The 2-COLORABILITY problem is to determine whether a given graph can be colored using only two colors. Show that 2-COLORABILITY is in P.

3. The 2-SAT problem is to determine whether a CNF formula in which no clause contains more than two literals is satisfiable. It is known that 2-SAT \in P. Show why a technique like the one used to show 3-SAT is NP-complete does not work for 2-SAT. Show that 2-SAT is in P.

4. The EXACT-COVER problem is to determine for a finite family of sets $\Delta = \{S_1, S_2, \dots, S_n\}$ whether there exists a set cover Γ of Δ such that the elements of Γ are pairwise disjoint. Show that EXACT-COVER is NP-complete. (*Hint:* Show that

$$\text{CHROMATIC-NUMBER} \leq_p \text{EXACT-COVER}.$$

Classifying Unsolvable Problems

1. Using Oracles

Once one gets used to the fact that there are explicit problems, such as the halting problem, that have no algorithmic solution, one is led to consider questions such as the following:

Suppose we were given a “black box” or, as one says, an *oracle*, which somehow can tell us whether a given Turing machine with given input eventually halts. (Of course, by Church’s thesis, the behavior of such an “oracle” cannot be characterized by an algorithm.) Then it is natural to consider a kind of program that is allowed to ask questions of our oracle and to use the answers in its further computation. Which noncomputable functions will now become computable?

In this chapter we will see how to give a precise answer to such questions. To begin with, we shall have to modify the programming language \mathcal{S} introduced in Chapter 2, to permit the use of “oracles.” Specifically, we change the definition of “statement” (in Chapter 2, Section 3) to allow statements of the form $V \leftarrow O(V)$ instead of $V \leftarrow V$. The modified version of \mathcal{S} thus contains four kinds of statement: increment, decrement, conditional branch, and this new kind of statement which we call an *oracle statement*. The definitions of *instruction*, *program*, *state*, *snapshot*, and *terminal snapshot* remain exactly as in Chapter 2.

We now let G be some partial function on N with values in N , and we shall think of G as an oracle. Let \mathcal{P} be a program of length n and let (i, σ) be a nonterminal snapshot of \mathcal{P} , i.e., $i \leq n$. We define the snapshot (j, τ) to be the G -successor of (i, σ) exactly as in the definition of *successor* in Chapter 2, Section 3, except that Case 3 is now replaced by

Case 3. The i th instruction of \mathcal{P} is $V \leftarrow O(V)$ and σ contains the equation $V = m$. If $G(m) \downarrow$, then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = G(m)$. If $G(m) \uparrow$, then (i, σ) has no successor.

Thus, when $G(m) \downarrow$, execution of this oracle statement has the intuitive effect of answering the computer's question " $G(m) = ?$ ". When $G(m) \uparrow$, an "out-of-bounds" condition is recognized and the computer halts without reaching a terminal snapshot. Of course, when G is total, every nonterminal snapshot has a successor.

A G -computation is defined just like computation except that the word *successor* is replaced by *G -successor*. A number m which is replaced by $G(m)$ in the course of a G -computation (under Case 3 above) is called an *oracle query of the G -computation*. We define $\psi_{\mathcal{P},G}^{(m)}(r_1, r_2, \dots, r_m)$ exactly as we defined $\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \dots, r_m)$ in Chapter 2, Section 4, except that the word *computation* is replaced by *G -computation*.

Now, let G be a total function. Then, the partial function $\psi_{\mathcal{P},G}^{(m)}(x_1, \dots, x_m)$ is said to be G -computed by \mathcal{P} . A partial function f is said to be *partially G -computable* or *G -partial recursive* if there is a program which G -computes it. A partially G -computable function which is total is called *G -computable* or *G -recursive*. Note that we have not defined *partially G -computable* unless G is a total function.

We have a few almost obvious theorems:

Theorem 1.1. If f is partially computable, then f is partially G -computable for all total functions G .

Proof. Clearly, we can assume that f is computed by a program \mathcal{P} containing no statements of the¹ form $V \leftarrow V$. Now this program \mathcal{P} is also a program in the new revised sense; moreover, a computation of \mathcal{P} is the same thing as a G -computation of \mathcal{P} since \mathcal{P} contains no oracle statements. Hence $\psi_{\mathcal{P},G}^{(m)} = \psi_{\mathcal{P}}^{(m)}$ for all G . ■

We write I for the identity function $I(x) = x$. (Thus, $I = u_1^1$.)

Theorem 1.2. f is partially computable if and only if f is partially I -computable.

¹ Unlabeled statements $V \leftarrow V$ can just be deleted, and

$$[L] \quad V \leftarrow V$$

can be replaced by

$$\begin{aligned} [L] &\quad V \leftarrow V + 1 \\ &\quad V \leftarrow V - 1 \end{aligned}$$

Proof. If f is partially computable, then by Theorem 1.1 it is certainly partially I -computable. Conversely, let \mathcal{P} I -compute f . Let \mathcal{P}' be obtained from \mathcal{P} by replacing each oracle statement $V \leftarrow O(V)$ by $V \leftarrow V$. Then, \mathcal{P}' is a program in the original sense and \mathcal{P}' computes f . ■

Theorem 1.3. Let G be a total function. Then G is G -computable.

Proof. The following program² clearly G -computes G :

$$\begin{aligned} X &\leftarrow O(X) \\ Y &\leftarrow X \end{aligned}$$

Theorem 1.4. The class of G -computable functions is a PRC class.

Proof. Exactly like the proof of Theorem 3.1 in Chapter 3. ■

This last proof illustrates a situation, which turns out to be quite typical, in which the proof of an earlier theorem can be used virtually intact to prove a theorem relative to an “oracle” G . One speaks of a *relativized* theorem and of *relativizing* a proof. It is a matter of taste how much detail to provide in such a case.

Theorem 1.5. Let F be partially G -computable and let G be H -computable. Then F is partially H -computable.

Proof. Let \mathcal{P} be a program which G -computes F . Let \mathcal{P}' be obtained from \mathcal{P} by replacing each oracle statement $V \leftarrow O(V)$ by a macro expansion obtained from some program which H -computes G . Then clearly, \mathcal{P}' H -computes F . ■

Theorem 1.6. Let G be any computable function. Then a function F is partially computable if and only if it is partially G -computable.

Proof. Theorem 1.1 gives the result in one direction. For the converse, let F be partially G -computable. By Theorem 1.2, G is I -computable. Hence, by Theorem 1.5, F is partially I -computable and so, by Theorem 1.2 again, F is partially computable. ■

It is useful to be able to work with “oracles” which are functions of more than one variable. We introduce this notion by using a familiar coding device from Chapter 3, Section 8.

Definition. Let f be a total n -ary function on N , $n > 1$. Then we say that g is (*partially*) f -computable to mean that g is (*partially*) G -computable, where

$$G(x) = f((x)_1, \dots, (x)_n). \tag{1.1}$$

² Of course, we can freely use macro expansions, as explained in Chapter 2.

Theorem 1.7. Let f be a total n -ary function on N . Then f is f -computable.

Proof. Let G be defined by (1.1). Then

$$f(x_1, \dots, x_n) = G([x_1, \dots, x_n]).$$

Hence the following program G -computes f :

$$Z \leftarrow [X_1, \dots, X_n]$$

$$Z \leftarrow O(Z)$$

$$Y \leftarrow Z$$

■

Since predicates are also total functions we can speak of a function being (partially) P -computable, where P is a predicate. Also, we speak of a function being (partially) A -computable when $A \subseteq N$; as usual, we simply identify A with the predicate which is its characteristic function.

Exercises

1. Provide a suitable definition of computability by a Post–Turing program relative to an oracle and prove an appropriate equivalence theorem.
2. For a given total function G from N to N , define the class $\text{Rec}(G)$ to be the class of functions obtained from G and the initial functions of Chapter 3 using composition, recursion, and minimalization. Prove that every function in $\text{Rec}(G)$ is partially G -computable.

2. Relativization of Universality

We now proceed to relativize the development in Chapter 4. As in Chapter 4, Section 1, we define an instruction number $\#(I) = \langle a, \langle b, c \rangle \rangle$ for all instructions I . The only difference is that $b = 0$ now indicates an oracle statement instead of one of the form $V \leftarrow V$. For \mathcal{P} a program, we now define $\#(\mathcal{P})$ as before. As indicated in Chapter 4, in order to avoid ambiguity we must not permit a program ending in the instruction whose number is 0. This instruction is now the unlabeled statement $Y \leftarrow O(Y)$. Hence, for complete rigor, if we wish to end a program with $Y \leftarrow O(Y)$, we will have to provide the statement with a spurious label.

We define $\Phi_G^{(n)}(x_1, \dots, x_n, y)$ to be $\psi_{\mathcal{P}, G}^{(n)}(x_1, \dots, x_n)$ where \mathcal{P} is the *unique* program such that $\#(\mathcal{P}) = y$. We also write $\Phi_G(x, y)$ for $\Phi_G^{(1)}(x, y)$. We have

Theorem 2.1. (Relativized Universality Theorem). Let G be total. Then the function $\Phi_G^{(n)}(x_1, \dots, x_n, y)$ is partially G -computable.

Proof. The proof of this theorem is essentially contained in the program of Fig. 2.1. The daggers (\ddagger) indicate the changes from the unrelativized universal program in Fig. 3.1 in Chapter 4. As in that case, what we have is essentially an interpretative program. The new element is of course the interpretation of oracle statements. This occurs in the following program segment which, not surprisingly, itself contains an oracle statement:

$$\begin{aligned} [O] \quad & W \leftarrow (S)_{r(U)+1} \\ & B \leftarrow W \\ & B \leftarrow O(B) \\ & S \leftarrow \lfloor S/P^W \rfloor \cdot P^B \end{aligned}$$

The program segment works as follows: First, W and B are both set to the current value of the variable in the oracle statement being interpreted. Then

$$\begin{aligned} Z &\leftarrow X_{n+1} + 1 \\ S &\leftarrow \prod_{i=1}^n (p_{2i})^{x_i} \\ K &\leftarrow 1 \\ [C] \quad & \text{IF } K = \text{Lt}(Z) + 1 \vee K = 0 \text{ GOTO } F \\ & U \leftarrow r((Z)_K) \\ & P \leftarrow p_{r(U)+1} \\ & \text{IF } l(U) = 0 \text{ GOTO } O \quad (\ddagger) \\ & \text{IF } l(U) = 1 \text{ GOTO } A \\ & \text{IF } \sim(P|S) \text{ GOTO } N \\ & \text{IF } l(U) = 2 \text{ GOTO } M \\ & K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)] \\ & \text{GOTO } C \\ [O] \quad & W \leftarrow (S)_{r(U)+1} \quad (\ddagger) \\ & B \leftarrow W \quad (\ddagger) \\ & B \leftarrow O(B) \quad (\ddagger) \\ & S \leftarrow \lfloor S/P^W \rfloor \cdot P^B \quad (\ddagger) \\ & \text{GOTO } N \quad (\ddagger) \\ [M] \quad & S \leftarrow \lfloor S/P \rfloor \\ & \text{GOTO } N \\ [A] \quad & S \leftarrow S \cdot P \\ [N] \quad & K \leftarrow K + 1 \\ & \text{GOTO } C \\ [F] \quad & Y \leftarrow (S)_1 \end{aligned}$$

Fig. 2.1. Program which G -computes $\Phi_G^{(n)}(X_1, \dots, X_n, X_{n+1})$.

an oracle statement gives B a new value which is G of the old value. Finally, this new value is stored as an exponent on the appropriate prime in the number S . The remainder of the program works exactly as in the unrelativized case. ■

```

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

    If  $Q > X_{n+2} + 1$  GOTO E (*)  

    If  $K = \text{Lt}(Z) + 1 \vee K = 0$  GOTO F  

 $U \leftarrow r((Z)_K)$   

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

    IF  $l(U) = 0$  GOTO O (*)  

    IF  $l(U) = 1$  GOTO A  

    IF  $\sim(P|S)$  GOTO N  

    IF  $l(U) = 2$  GOTO M  

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

    GOTO C  

[O]  $W \leftarrow (S)_{r(U)+1}$  (#)  

 $B \leftarrow W$  (#)  

 $B \leftarrow O(B)$  (#)  

 $S \leftarrow \lfloor S/P^W \rfloor \cdot P^B$  (#)  

    GOTO N (#)  

[M]  $S \leftarrow \lfloor S/P \rfloor$   

    GOTO N  

[A]  $S \leftarrow S \cdot P$   

[N]  $K \leftarrow K + 1$   

    GOTO C  

[F]  $Y \leftarrow 1$  (*)

```

Fig. 2.2. Program which G -computes $\text{STP}_G^{(n)}(X_1, \dots, X_n, X_{n+1}, X_{n+2})$.

Let G be any *partial* function on N with values in N . Then we define the relativized step-counter predicate by

$\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t) \Leftrightarrow$ there is a G -computation of program number y of length $\leq t + 1$ beginning with inputs x_1, \dots, x_n .

As in the unrelativized case, we have

Theorem 2.2 (Relativized Step-Counter Theorem). For any total function G , the predicates $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ are G -computable.

A program to compute $\text{STP}_G^{(n)}$ can be obtained from our relativized universal program exactly as the program for computing $\text{STP}^{(n)}$ was obtained from the unrelativized universal program. The program is given in Fig. 2.2. The asterisks (*) indicate changes from the relativized universal program and the daggers (\dagger) as before indicate the changes made in relativizing. For an explanation of how the program (relativized or unrelativized) works, see the discussion of the unrelativized step-counter theorem (Theorem 3.2 in Chapter 4).

We shall now consider certain partial functions with finite domains, and use numbers as codes for them. For every $u \in N$ we define

$$\{u\}(i) = \begin{cases} (r(u))_{i+1} & \text{for } i < l(u) \\ \uparrow & \text{for } i \geq l(u). \end{cases} \quad (2.1)$$

Thus, if $l(u) = 0$, then $\{u\} = \emptyset$, the nowhere defined function. Also, if

$$u = \langle k, [a_0, a_1, \dots, a_{k-1}] \rangle,$$

then $\{u\}(x) = a_x$ for $x = 0, 1, \dots, k-1$ and $\{u\}(x)\uparrow$ for $x \geq k$.

Theorem 2.3. The predicate

$$P(x_1, \dots, x_n, y, t, u) \Leftrightarrow \text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$$

is computable.

Proof. We will transform the program in Fig. 2.2 into one which computes $P(x_1, \dots, x_n, x_{n+1}, x_{n+2}, x_{n+3})$. We need only replace the single oracle statement $B \leftarrow O(B)$ by instructions that operate on x_{n+3} to obtain the required information about $\{x_{n+3}\}$. This involves first testing for $\{x_{n+3}\}(b)\downarrow$, where b is the value of the variable B . If $\{x_{n+3}\}(b)\uparrow$, computation should halt with output 0, because there is no computation in this case. Otherwise B should be given the value $\{x_{n+3}\}(b)$. Thus, by (2.1), it suffices to replace the oracle statement $B \leftarrow O(B)$ in the program in Fig. 2.2 by the following pair of instructions:

IF $l(X_{n+3}) \leq B$ GOTO E
 $B \leftarrow (r(X_{n+3}))_{B+1}$

■

Now, let G be a total function. Then, we define

$$u \prec G$$

to mean that $\{u\}(i) = G(i)$ for $0 \leq i < l(u)$. [Of course, by (2.1), $\{u\}(i)\uparrow$ for $i \geq l(u)$.] For a number u such that $u \prec G$, values of G can be retrieved by using the equations

$$\dot{G}(i) = (r(u))_{i+1}, \quad i = 0, 1, \dots, l(u) - 1.$$

We can use the predicate $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ to obtain an important result which isolates the noncomputability of the relativized step-counter predicate in a way which will prove helpful. The simple observation on which this result capitalizes is that any G -computation can contain only finitely many oracle queries.

Theorem 2.4 (Finiteness Theorem). Let G be a total function. Then, we have

$$\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t) \Leftrightarrow (\exists u)[u \prec G \ \& \ \text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)].$$

Proof. First suppose that $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ is true for some given values of x_1, \dots, x_n, y, t , and let \mathcal{P} be the program with $\#(\mathcal{P}) = y$. Let s_1, s_2, \dots, s_k be a G -computation of \mathcal{P} where s_1 is the initial snapshot corresponding to the input values x_1, x_2, \dots, x_n and where $k \leq t + 1$. Let M be the largest value of an oracle query of this G -computation, and let $u = \langle M + 1, [G(0), G(1), \dots, G(M)] \rangle$. Thus, $u \prec G$ and $\{u\}(m) = G(m)$ for all $m \leq M$. Hence, s_1, s_2, \dots, s_k is likewise a $\{u\}$ -computation of \mathcal{P} . Since $k \leq t + 1$, $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ is true.

Conversely, let us be given $u \prec G$ such that $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ is true, and let $\#(\mathcal{P}) = y$. Let s_1, s_2, \dots, s_k be a $\{u\}$ -computation of \mathcal{P} where s_1 is the initial snapshot corresponding to the input values x_1, x_2, \dots, x_n and where $k \leq t + 1$. For each m which is an oracle query of this $\{u\}$ -computation, we must have $\{u\}(m)\downarrow$, since otherwise one of the snapshots in this $\{u\}$ -computation would be nonterminal and yet not have a successor. Since $u \prec G$, we must have $\{u\}(m) = G(m)$ for all such m . Hence s_1, s_2, \dots, s_k is likewise a G -computation of \mathcal{P} . Since $k \leq t + 1$, $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ is true. ■

To conclude this section we turn to the parameter theorem (Theorem 5.1 in Chapter 4).

Theorem 2.5 (Relativized and Strengthened Parameter Theorem). For each $n, m > 0$, there is a primitive recursive function $S_m^n(u_1, \dots, u_n, y)$ such that for every total function G :

$$\Phi_G^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi_G^{(m)}(x_1, \dots, x_m, S_m^n(u_1, \dots, u_n, y)). \quad (2.2)$$

Moreover, the functions S_m^n have the property:

$$S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, y) \quad \text{implies} \quad u_1 = \bar{u}_1, \dots, u_n = \bar{u}_n.$$

Proof. The functions S_m^n are defined exactly as in the proof of Theorem 5.1 in Chapter 4. We briefly give the proof again in a slightly different way. Thus, let $\#(\mathcal{P}) = y$; then the function $S_m^1(u, y)$ is defined to be the number of the program $\bar{\mathcal{P}}$ obtained from \mathcal{P} by preceding it by the statement

$$X_{m+1} \leftarrow X_{m+1} + 1$$

repeated u times. Since $\bar{\mathcal{P}}$ on inputs x_1, \dots, x_m will do exactly what \mathcal{P} would have done on inputs x_1, \dots, x_m, u , we have

$$\Phi_G^{(m+1)}(x_1, \dots, x_m, u, y) = \Phi_G^{(m)}(x_1, \dots, x_m, S_m^1(u, y)),$$

the desired result for $n = 1$. To complete the proof, we define S_m^n for $n > 1$ by the recursion

$$S_m^{k+1}(u_1, \dots, u_k, u_{k+1}, y) = S_m^k(u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y)).$$

It is now easy to prove by induction on n that if $\#(\mathcal{P}) = y$, then $S_m^n(u_1, \dots, u_n, y) = \#(\bar{\mathcal{P}})$, where $\bar{\mathcal{P}}$ is obtained from \mathcal{P} by preceding it by the following program consisting of $u_n + \dots + u_1$ statements.

$$\begin{array}{c} X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ \vdots \\ X_{m+n} \leftarrow X_{m+n} + 1 \\ \vdots \\ X_{m+n} \leftarrow X_{m+n} + 1 \end{array} \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} u_1 \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} u_n$$

Hence, $\bar{\mathcal{P}}$ on inputs x_1, \dots, x_m will do exactly what \mathcal{P} would have done on inputs $x_1, \dots, x_m, u_1, \dots, u_n$. Thus, we obtain (2.2).

Finally, let

$$S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, y) = \#(\bar{\mathcal{P}}),$$

and let $y = \#(\mathcal{P})$. Then, $\bar{\mathcal{P}}$ consists of a list of increment statements followed by \mathcal{P} , and for $1 \leq i \leq n$, u_i and \bar{u}_i are both simply the number of times the statement

$$X_{m+i} \leftarrow X_{m+i} + 1$$

occurs in $\bar{\mathcal{P}}$ preceding \mathcal{P} . Thus, $u_i = \bar{u}_i$. ■

Exercises

1. (a) Show that the functions S_m^n do *not* have the property:
 $S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, \bar{y})$ implies $u_1 = \bar{u}_1, \dots, u_n = \bar{u}_n, y = \bar{y}$.
(b) Can the definition of S_m^n be modified so the parameter theorem continues to hold, but so the condition of (a) holds as well? How?
2. Prove the converse of Exercise 1.2.

3. Reducibility

If A and B are sets such that A is B -recursive, we also say that A is *Turing-reducible to B* and we write $A \leq_t B$. We have

Theorem 3.1. $A \leq_t A$. If $A \leq_t B$ and $B \leq_t C$, then $A \leq_t C$.

Proof. The first statement follows at once from Theorem 1.3 and the second from Theorem 1.5. ■

Any relation on the subsets of N for which Theorem 3.1 is true is called a *reducibility*. Many reducibilities have been studied. We mention one in particular:

Definition. We write $A \leq_m B$ and say that A is *many-one reducible to B* if there is a recursive function f such that

$$A = \{x \in N \mid f(x) \in B\}.$$

If the function f is, in addition, one-one, i.e., if $f(x) = f(y)$ implies $x = y$, then we write $A \leq_1 B$ and say that A is *one-one reducible to B* .

Theorem 3.2. $A \leq_1 B$ implies $A \leq_m B$ implies $A \leq_t B$.

Proof. The first implication is immediate. For the second, let $A = \{x \in N \mid f(x) \in B\}$, where f is recursive. Then the following program B -computes A :

$$\begin{aligned} X &\leftarrow f(X) \\ X &\leftarrow O(X) \\ Y &\leftarrow X \end{aligned}$$

Theorem 3.3. \leq_1 and \leq_m are both reducibilities.

Proof. Clearly $A = \{x \in N \mid I(x) \in A\}$, where I is the identity function. Hence $A \leq_1 A$ and therefore $A \leq_m A$.

Let $A \leq_m B$ and $B \leq_m C$, and let

$$A = \{x \in N \mid f(x) \in B\}, \quad B = \{x \in N \mid g(x) \in C\},$$

where f, g are recursive. Then

$$A = \{x \in N \mid g(f(x)) \in C\},$$

so that $A \leq_m C$. If, moreover, f and g are one-one and $h(x) = g(f(x))$, then h is also one-one, because

$$\begin{aligned} h(x) = h(y) &\quad \text{implies } g(f(x)) = g(f(y)) \\ &\quad \text{implies } f(x) = f(y) \\ &\quad \text{implies } x = y. \end{aligned}$$

■

Thus we have three examples, \leq_1 , \leq_m , and \leq_t , of reducibilities. Polynomial-time reducibility, \leq_p , studied in Chapter 15, is another example. (In fact, historically, polynomial-time reducibility was suggested by many-one reducibility.) There are a number of simple properties that all reducibilities share. To work some of these out, let us write \leq_Q to represent an arbitrary reducibility. By replacing Q by 1, m, t (or even p) we specialize to the particular reducibilities we have been studying. We write $A \not\leq_Q B$ to indicate that it is not the case that $A \leq_Q B$.

Definition. $A \equiv_Q B$ means that $A \leq_Q B$ and $B \leq_Q A$.

Theorem 3.4. For any reducibility \leq_Q :

$$\begin{aligned} A \equiv_Q A, \\ A \equiv_Q B &\quad \text{implies } B \equiv_Q A, \\ A \equiv_Q B &\quad \text{and } B \equiv_Q C \quad \text{implies } A \equiv_Q C. \end{aligned}$$

Proof. Immediate from the definition. ■

Definition. Let \mathbf{W} be a collection of subsets of N and let \leq_Q be a reducibility. \mathbf{W} is called *Q-closed* if it has the property

$$A \in \mathbf{W} \text{ and } B \leq_Q A \quad \text{implies } B \in \mathbf{W}.$$

Also, a set $A \in \mathbf{W}$ is called *Q-complete for \mathbf{W}* if for every $B \in \mathbf{W}$ we have $B \leq_Q A$.

NP-completeness, studied in Chapter 15, is, in the present terminology, *polynomial-time completeness for NP*. Completeness of a set A is often proved

by showing that a set already known to be complete can be reduced to A ; this method was used in Chapter 15 in the case of NP-completeness.

Theorem 3.5. Let A be Q -complete for \mathbf{W} , let $B \in \mathbf{W}$, and let $A \leq_Q B$. Then B is Q -complete for \mathbf{W} .

Proof. Let $C \in \mathbf{W}$. Then $C \leq_Q A$. Hence $C \leq_Q B$. ■

If \mathbf{W} is a collection of subsets of N , we write

$$\text{co-}\mathbf{W} = \{A \subseteq N \mid \bar{A} \in \mathbf{W}\}.$$

Theorem 3.6. Let $\text{co-}\mathbf{W}$ be Q -closed, let A be Q -complete for \mathbf{W} , and let $A \in \text{co-}\mathbf{W}$. Then we have $\mathbf{W} = \text{co-}\mathbf{W}$.

Proof. Let $B \in \mathbf{W}$. Then, since A is Q -complete for \mathbf{W} , $B \leq_Q A$. Since $A \in \text{co-}\mathbf{W}$ and $\text{co-}\mathbf{W}$ is Q -closed, $B \in \text{co-}\mathbf{W}$. This proves that $\mathbf{W} \subseteq \text{co-}\mathbf{W}$.

Next let $B \in \text{co-}\mathbf{W}$. Then $\bar{B} \in \mathbf{W}$. By what has already been shown, $\bar{B} \in \text{co-}\mathbf{W}$. Hence $B \in \mathbf{W}$. This proves that $\text{co-}\mathbf{W} \subseteq \mathbf{W}$. ■

As we shall see, Theorem 3.6 is quite useful. Our applications will be to the case of one-one and many-one reducibility. For this purpose, it is useful to note

Theorem 3.7. If $A \leq_m B$, then $\bar{A} \leq_m \bar{B}$. Likewise if $A \leq_1 \bar{B}$, then $\bar{A} \leq_1 \bar{B}$.

Proof. If $A = \{x \in N \mid f(x) \in B\}$, then clearly $\bar{A} = \{x \in N \mid f(x) \in \bar{B}\}$. ■

Corollary 3.8. If \mathbf{W} is m -closed or 1 -closed, then so is $\text{co-}\mathbf{W}$.

Proof. Let $B \in \text{co-}\mathbf{W}$, $A \leq_m B$. By the theorem, $\bar{A} \leq_m \bar{B}$. Since $\bar{B} \in \mathbf{W}$ and \mathbf{W} is m -closed, $\bar{A} \in \mathbf{W}$. Hence $A \in \text{co-}\mathbf{W}$. Similarly for \leq_1 . ■

For a concrete example, we may take \mathbf{W} to be the collection of r.e. subsets of N . (For notation, the reader should review Chapter 4, Section 4.) We have

Theorem 3.9. K is 1-complete for the class of r.e. sets.

Proof. Let A be any r.e. set. We must show that $A \leq_1 K$. Since A is r.e., we have

$$A = \{x \in N \mid f(x) \downarrow\},$$

where f is a partially computable function. Let $g(t, x) = f(x)$ for all t, x . Thus, g is also partially computable. Using the (unrelativized) universality and parameter theorems, we have for a suitable number e :

$$g(t, x) = \Phi^{(2)}(t, x, e) = \Phi(t, S_1^1(x, e)).$$

Hence,

$$\begin{aligned} A &= \{x \in N \mid f(x) \downarrow\} \\ &= \{x \in N \mid g(S_1^1(x, e), x) \downarrow\} \\ &= \{x \in N \mid \Phi(S_1^1(x, e), S_1^1(x, e)) \downarrow\} \\ &= \{x \in N \mid S_1^1(x, e) \in K\}. \end{aligned}$$

Thus, $A \leq_m K$. But, by the strengthened version of the parameter theorem (Theorem 2.5), $S_1^1(x, e)$ is actually one-one. Hence, $A \leq_1 K$. ■

The class of r.e. sets is easily seen to be m-closed. Thus, let f be partially computable, let $A = \{x \in N \mid f(x) \downarrow\}$, and let $B = \{x \in N \mid g(x) \in A\}$, where g is computable. Then

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

so that B is r.e. Applying Theorems 3.2, 3.6, and 3.9 and Corollary 3.8 we obtain the not very interesting conclusion:

If \bar{K} is r.e., then the complement of every r.e. set is r.e.

Since we know that \bar{K} is in fact not r.e., this does us no good. However, Corollary 3.8 and Theorem 3.6 together with the fact that there is an r.e. set (e.g., K) whose complement is not r.e. permits us to conclude

Theorem 3.10. If A is m-complete for the class of r.e. sets, then \bar{A} is not r.e., so that A is not recursive.

We conclude this section with a simple but important construction. For $A, B \subseteq N$ we write

$$A \oplus B = \{2x \mid x \in A\} \cup \{2x + 1 \mid x \in B\}.$$

Intuitively, $A \oplus B$ contains the information in both A and B and nothing else. This suggests the truth of the following simple result:

Theorem 3.11. $A \leq_1 A \oplus B$, $B \leq_1 A \oplus B$. If $A \leq_1 C$ and $B \leq_1 C$, then $A \oplus B \leq_1 C$.

Proof. The following program $(A \oplus B)$ -computes A :

$$\begin{aligned} X &\leftarrow 2X \\ X &\leftarrow O(X) \\ Y &\leftarrow X \end{aligned}$$

If the first instruction is replaced by $X \leftarrow 2X + 1$, the program $(A \oplus B)$ computes B .

Finally, let C_A, C_B be the characteristic functions of A and B , respectively. Assuming that A and B are both C -computable, there must be programs which C -compute the functions C_A and C_B , respectively. Hence, we may use macros

$$Y \leftarrow C_A(X) \quad \text{and} \quad Y \leftarrow C_B(X)$$

in programs having C available as oracle. Thus, the following program C -computes $A \oplus B$:

```

IF  $2|X$  GOTO D
     $X \leftarrow \lfloor (X - 1)/2 \rfloor$ 
     $Y \leftarrow C_B(X)$ 
    GOTO E
[D]    $X \leftarrow \lfloor X/2 \rfloor$ 
     $Y \leftarrow C_A(X)$ 

```

■

Exercises

1. Let $U = \{x \in N \mid l(x) \in W_{r(x)}\}$. Show that U is 1-complete for the class of r.e. sets.

2. Let $K \leq_1 A$ and let

$$C = \{x \in K \mid \Phi_x(x) \notin A \oplus \bar{A}\}.$$

Prove that $A \leq_1 C, C \leq_1 A$, but $C \not\leq_m A$.

3. Prove that Theorem 3.11 holds with \leq_1 replaced by \leq_m .

4. Let $\text{FIN} = \{x \in N \mid W_x \text{ is finite}\}$. Prove that $K \leq_1 \text{FIN}$.

5. Prove that if $B, \bar{B} \neq \emptyset$, then for every recursive set A , $A \leq_m B$.

4. Sets R.E. Relative to an Oracle

If G is a total function (of one or more arguments) we say that a set $B \subseteq N$ is G -recursively enumerable (abbreviated G -r.e.) if there is a partially G -computable function g such that

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

By Theorem 1.6, r.e. sets are then simply sets that are G -r.e. for some computable function G .

It is easy to relativize the proofs in Chapter 4, Section 4, using, in particular, the relativized step-counter theorem. We give some of the results and leave the details to the reader.

Theorem 4.1. If B is a G -recursive set, then B is G -r.e.

Theorem 4.2. The set B is G -recursive if and only if B and \bar{B} are both G -r.e.

Theorem 4.3. If B and C are G -r.e. sets, so are $B \cup C$ and $B \cap C$.

Next, we obtain

Theorem 4.4. The set A is G -r.e. if and only if there is a G -computable predicate $Q(x, t)$ such that

$$A = \{x \in N \mid (\exists t)Q(x, t)\}. \quad (4.1)$$

Proof. First let A be G -r.e. Then, there is a partially G -computable function h such that

$$A = \{x \in N \mid h(x)\downarrow\}.$$

Writing $h(x) = \Phi_G(x, z_0)$, we have

$$A = \{x \in N \mid (\exists t)\text{STP}_G^{(1)}(x, z_0, t)\},$$

which gives the result in one direction.

Conversely, let (4.1) hold, where Q is a G -computable predicate. Let $h(x)$ be the partial function which is G -computed by the following program:

```
[B]      Z ← Q(X, Y)
          Y ← Y + 1
          IF Z = 0 GOTO B
```

Then clearly,

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

so that A is G -r.e. ■

Corollary 4.5. The set A is G -r.e. if and only if there is a G -recursive set B such that

$$A = \{x \in N \mid (\exists y)(\langle x, y \rangle \in B)\}.$$

Proof. If B is G -recursive, then the predicate $\langle x, y \rangle \in B$ is G -computable (by Theorem 1.4) and hence, by the theorem, A is G -r.e.

Conversely, if A is G -r.e., we have a G -computable predicate Q such that (4.1) holds. Letting $B = \{z \in N \mid Q(l(z), r(z))\}$, B is (again by Theorem 1.4) G -recursive and

$$A = \{x \in N \mid (\exists y)(\langle x, y \rangle \in B)\}. \quad \blacksquare$$

For any unary function G , we write

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

(Thus $W_n = W_n^I$.) For the remainder of this section, G will be a unary total function. We have at once

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

We define

$$G' = \{n \in N \mid n \in W_n^G\}.$$

(Thus, $K = I'$.) G' is called the *jump* of G . We have

Theorem 4.7. G' is G -r.e. but not G -recursive.

This is just the relativization of Theorem 4.6, in Chapter 4, and the proof of that theorem relativizes easily. However, we include the details because of the importance of the result.

Proof of Theorem 4.7. Since

$$G' = \{n \in N \mid \Phi_G(n, n) \downarrow\},$$

the relativized universality theorem shows that G' is G -r.e. If \bar{G}' were also G -r.e., we would have $\bar{G}' = W_i^G$ for some $i \in N$. Then

$$i \in G' \Leftrightarrow i \in W_i^G \Leftrightarrow i \in \bar{G}',$$

a contradiction. ■

Our next result is essentially a relativization of Theorem 3.9.

Theorem 4.8. The following assertions are all equivalent:

- (a) $A \leq_1 G'$;
- (b) $A \leq_m G'$;
- (c) A is G -r.e.

Proof. It is obvious that (a) implies (b). To see that (b) implies (c), let h be a recursive function such that

$$x \in A \quad \text{if and only if} \quad h(x) \in G'.$$

Then

$$x \in A \quad \text{if and only if} \quad \Phi_G(h(x), h(x)) \downarrow,$$

so that A is G -r.e.

Finally, to see that (c) implies (a), let A be G -r.e., so that we can write

$$A = \{x \in N \mid f(x) \downarrow\},$$

where f is partially G -computable. Let $g(t, x) = f(x)$ for all t, x . By the relativized universality and parameter theorems, we have, for some number e ,

$$g(t, x) = \Phi_G^{(2)}(t, x, e) = \Phi_G(t, S_1^1(x, e)).$$

Hence,

$$\begin{aligned} A &= \{x \in N \mid f(x) \downarrow\} \\ &= \{x \in N \mid g(S_1^1(x, e), x) \downarrow\} \\ &= \{x \in N \mid \Phi_G(S_1^1(x, e), S_1^1(x, e)) \downarrow\} \\ &= \{x \in N \mid S_1^1(x, e) \in G'\}. \end{aligned}$$

Since, by Theorem 2.5, $S_1^1(x, e)$ is one-one, we have $A \leq_1 G'$. ■

Theorem 4.9. If F and G are total unary functions and F is G -recursive, then $F' \leq_1 G'$.

Proof. By Theorem 4.7, F' is F -r.e. That is, we can write

$$F' = \{x \in N \mid f(x) \downarrow\},$$

where f is partially F -computable. By Theorem 1.5, f is also partially G -computable. Hence F' is G -r.e. By Theorem 4.8, $F' \leq_1 G'$. ■

By iterating the jump operation, we can obtain a hierarchy of problems each of which is “more unsolvable” than the preceding one.

We write $G^{(n)}$ for the jump iterated n times. That is, we define:

$$G^{(0)} = G,$$

$$G^{(n+1)} = (G^{(n)})'.$$

We have

Theorem 4.10. $\emptyset^{(n+1)}$ is $\emptyset^{(n)}$ -r.e. but not $\emptyset^{(n)}$ -recursive.

Proof. Immediate from Theorem 4.7. ■

It should be noted that, by Theorem 4.9, $K \equiv_1 \emptyset'$, since I and \emptyset are both recursive and $K = I'$. Later we shall see that much more can be said along these lines.

Exercise

1. Show that there are sets A, B, C such that A is B -r.e. and B is C -r.e., but A is not C -r.e.

5. The Arithmetic Hierarchy

The arithmetic hierarchy, which we will study in this section, is one of the principle tools used in classifying unsolvable problems.

Definition. Σ_0 is the class of recursive sets. For each $n \in N$, Σ_{n+1} is the class of sets which are A -r.e. for some set A which belongs to Σ_n . For all n , $\Pi_n = \text{co-}\Sigma_n$, $\Delta_n = \Sigma_n \cap \Pi_n$.

Note that Σ_1 is the class of r.e. sets and that $\Sigma_0 = \Pi_0 = \Delta_0 = \Delta_1$ is the class of recursive sets.

Theorem 5.1. $\Sigma_n \subseteq \Sigma_{n+1}$, $\Pi_n \subseteq \Pi_{n+1}$.

Proof. For any set $A \in \Sigma_n$, A is A -r.e. and hence $A \in \Sigma_{n+1}$. The rest follows by taking complements. ■

Theorem 5.2. $\emptyset^{(n)} \in \Sigma_n$.

Proof. By induction. For $n = 0$ the result is obvious. The inductive step follows at once from Theorem 4.10. ■

Theorem 5.3. $A \in \Sigma_{n+1}$ if and only if A is $\emptyset^{(n)}$ -r.e.

Proof. If A is $\emptyset^{(n)}$ -r.e., it follows at once from Theorem 5.2 that $A \in \Sigma_{n+1}$.

We prove the converse by induction. If $A \in \Sigma_1$, then A is r.e., so, of course, A is \emptyset -r.e. Assume the result known for $n = k$ and let $A \in \Sigma_{k+2}$. Then A is B -r.e. for some $B \in \Sigma_{k+1}$. By the induction hypothesis, B is $\emptyset^{(k)}$ -r.e. By Theorem 4.8, $A \leq_1 B'$ and $B \leq_1 \emptyset^{(k+1)}$. By Theorem 4.9, $B' \leq_1 \emptyset^{(k+2)}$. Hence $A \leq_1 \emptyset^{(k+2)}$, and by Theorem 4.8 again, A is $\emptyset^{(k+1)}$ -r.e. ■

Corollary 5.4. For $n \geq 1$ the following are all equivalent:

$$A \leq_1 \emptyset^{(n)};$$

$$A \leq_m \emptyset^{(n)};$$

$$A \in \Sigma_n.$$

Proof. This follows at once from Theorems 4.8 and 5.3. ■

Corollary 5.5. For $n \geq 1$, $\emptyset^{(n)}$ is 1-complete for Σ_n .

Proof. Immediate from Theorem 5.2 and Corollary 5.4. ■

Corollary 5.6. For $n \geq 1$, Σ_n and Π_n are both m-closed and hence 1-closed.

Proof. Let $A \in \Sigma_n$, $B \leq_m A$. Then using Corollary 5.4 twice, $B \leq_m \emptyset^{(n)}$, and hence $B \in \Sigma_n$. This proves that Σ_n is m-closed. The result for Π_n is now immediate from Corollary 3.8. ■

Theorem 5.7. $A \in \Delta_{n+1}$ if and only if $A \leq_t \emptyset^{(n)}$.

Proof. Immediate from Theorems 4.2 and 5.3. ■

In particular, since $K \equiv_1 \emptyset'$ (actually $K \equiv_1 \emptyset$), Δ_2 consists of all sets which are K -recursive, that is, sets for which there are algorithms which can decide membership by making use of an oracle for the halting problem.

Theorem 5.8. $\Sigma_n \cup \Pi_n \subseteq \Delta_{n+1}$.

Proof. For $n = 0$, the inclusion becomes an equality, so we assume $n \geq 1$. If $A \in \Sigma_n$, then by Corollary 5.4, $A \leq_1 \emptyset^{(n)}$, so by Theorem 5.7, $A \in \Delta_{n+1}$. If $A \in \Pi_n$, then $\bar{A} \leq_1 \emptyset^{(n)}$. But clearly $A \leq_1 \bar{A}$ (for example, by Theorem 1.4). Hence $A \leq_1 \emptyset^{(n)}$ and by Theorem 5.7, $A \in \Delta_{n+1}$. ■

Theorem 5.9. For $n \geq 1$, $\emptyset^{(n)} \in \Sigma_n - \Delta_n$.

Proof. By Theorem 4.10, $\emptyset^{(n)}$ is not $\emptyset^{(n-1)}$ -recursive. ■

Theorem 5.10. (Kleene's Hierarchy Theorem). We have for $n \geq 1$

- (1) $\Delta_n \subset \Sigma_n$, $\Delta_n \subset \Pi_n$;
- (2) $\Sigma_n \subset \Sigma_{n+1}$, $\Pi_n \subset \Pi_{n+1}$;
- (3) $\Sigma_n \cup \Pi_n \subset \Delta_{n+1}$.

Proof. (1) By definition $\Delta_n \subseteq \Sigma_n$, $\Delta_n \subseteq \Pi_n$. By Theorem 5.9, $\emptyset^{(n)} \in \Sigma_n - \Delta_n$, and so $\emptyset^{(n)} \in \Pi_n - \Delta_n$. Thus the inclusions are proper.

(2) By Theorem 5.1 we need only show that the inclusions are proper. But $\emptyset^{(n+1)} \in \Sigma_{n+1}$. If $\emptyset^{(n+1)} \in \Sigma_n$, by Theorem 5.8, $\emptyset^{(n+1)} \in \Delta_{n+1}$, contradicting Theorem 5.9. Likewise $\emptyset^{(n+1)} \in \Pi_{n+1} - \Pi_n$.

(3) By Theorem 5.8, we need only show that the inclusion is proper. Let $A_n = \emptyset^{(n)} \oplus \emptyset^{(n)}$. We shall show that $A_n \in \Delta_{n+1} - (\Sigma_n \cup \Pi_n)$. By Theorem 3.11 (with $C = \emptyset^{(n)}$), we have $A_n \leq_t \emptyset^{(n)}$. Hence $A_n \in \Delta_{n+1}$. Also,

$$\emptyset^{(n)} = \{x \in N \mid 2x \in A_n\},$$

$$\overline{\emptyset^{(n)}} = \{x \in N \mid 2x + 1 \in A_n\}.$$

Hence, $\emptyset^{(n)} \leq_1 A_n$, $\overline{\emptyset^{(n)}} \leq_1 A_n$. Suppose that $A_n \in \Sigma_n$. Then, by Corollary 5.6, $\emptyset^{(n)} \in \Sigma_n$, so that $\emptyset^{(n)} \in \Delta_n$, contradicting Theorem 5.9. Likewise if $A_n \in \Pi_n$, then $\emptyset^{(n)} \in \Pi_n$ and hence $\emptyset^{(n)} \in \Delta_n$. ■

Since we have now seen that for all $n \geq 1$, $\Sigma_n \neq \text{co-}\Sigma_n$, and since we know that for $n \geq 1$, Σ_n and Π_n are each m-closed, we may apply Theorem 3.6 to obtain the following extremely useful result:

Theorem 5.11. If A is m-complete for Σ_n , then $A \notin \Pi_n$. Likewise, if A is m-complete for Π_n , then $A \notin \Sigma_n$.

6. Post's Theorem

In order to make use of the arithmetic hierarchy, we will employ an alternative characterization of the classes Σ_n , Π_n involving strings of quantifiers. This alternative formulation is most naturally expressed in terms of predicates rather than sets. Hence we will use the following terminology:

We first associate with each predicate $P(x_1, \dots, x_s)$ the set

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then we say that P is Σ_n or that P is a Σ_n predicate to mean that $A \in \Sigma_n$. Likewise, we say that P is Π_n or Δ_n if $A \in \Pi_n$ or $A \in \Delta_n$, respectively. Notice that we continue to regard Σ_n and Π_n as consisting of subsets of N , and we will not speak of a predicate as being a member of Σ_n or Π_n .

Our terminology involves a slight anomaly for unary predicates. We have just defined $P(x)$ to be Σ_n (or Π_n) if the set $A = \{x \in N \mid P((x)_1)\}$ belongs to Σ_n (or Π_n), whereas it would be more natural to speak of $P(x)$ as being Σ_n (or Π_n) depending on whether $B = \{x \in N \mid P(x)\}$ belongs to Σ_n (or Π_n). Fortunately, there is really no conflict, for we have

Theorem 6.1. Let $B = \{x \in N \mid P(x)\}$. Then $P(x)$ is Σ_n if and only if $B \in \Sigma_n$. Likewise for Π_n , Δ_n .

Proof. For $n = 0$, the result is obvious, so assume that $n \geq 1$. $P(x)$ is Σ_n (or Π_n , or Δ_n) if and only if the set $A = \{x \in N \mid P((x)_1)\}$ belongs to Σ_n (or Π_n or Δ_n). Now,

$$A = \{x \in N \mid (x)_1 \in B\},$$

and

$$B = \{x \in N \mid 2^x \in A\}.$$

Thus $A \equiv_m B$. By Corollary 5.6, this gives the result. ■

Theorem 6.2. Let $P(x_1, \dots, x_s)$ be a Σ_n predicate and let

$$Q(t_1, \dots, t_k) \Leftrightarrow P(f_1(t_1, \dots, t_k), \dots, f_s(t_1, \dots, t_k)),$$

where f_1, \dots, f_s are computable functions. Then Q is also Σ_n . Likewise for Π_n .

Proof. Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

$$B = \{t \in N \mid Q((t)_1, \dots, (t)_k)\}.$$

We shall prove that $B \leq_m A$. It will thus follow that if $A \in \Sigma_n$ (or Π_n), then $B \in \Sigma_n$ (or Π_n), giving the desired result.

We have:

$$\begin{aligned} t \in B &\Leftrightarrow Q((t)_1, \dots, (t)_k) \\ &\Leftrightarrow P(f_1((t)_1, \dots, (t)_k), \dots, f_s((t)_1, \dots, (t)_k)) \\ &\Leftrightarrow [f_1((t)_1, \dots, (t)_k), \dots, f_s((t)_1, \dots, (t)_k)] \in A, \end{aligned}$$

so that $B \leq_m A$. ■

Theorem 6.3. A predicate P is Σ_n (or Π_n) if and only if $\sim P$ is Π_n (or Σ_n).

Proof. $A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}$ implies

$$\bar{A} = \{x \in N \mid \sim P((x)_1, \dots, (x)_s)\}. \quad \blacksquare$$

Theorem 6.4. Let $P(x_1, \dots, x_s)$, $Q(x_1, \dots, x_s)$ be Σ_n (or Π_n). Then the predicates $P \& Q$ and $P \vee Q$ are likewise Σ_n (or Π_n).

Proof. For $n = 0$, the result is obvious. Assume that $n \geq 1$ and let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

$$B = \{x \in N \mid Q((x)_1, \dots, (x)_s)\},$$

$$C = \{x \in N \mid P((x)_1, \dots, (x)_s) \& Q((x)_1, \dots, (x)_s)\},$$

$$D = \{x \in N \mid P((x)_1, \dots, (x)_s) \vee Q((x)_1, \dots, (x)_s)\}.$$

Thus, $C = A \cap B$ and $D = A \cup B$. If P and Q are Σ_n , then $A, B \in \Sigma_n$. Thus, by Theorem 5.3, A and B are both $\emptyset^{(n-1)}$ -r.e. By Theorem 4.3, C and D are likewise $\emptyset^{(n-1)}$ -r.e., and so $P \& Q$ and $P \vee Q$ are Σ_n .

If P and Q are Π_n , then $A, B \in \Pi_n$ so that $\bar{A}, \bar{B} \in \Sigma_n$. By Theorems 4.3 and 5.3, $\bar{A} \cap \bar{B} = (\overline{A \cup B}) \in \Sigma_n$ and $\bar{A} \cup \bar{B} = (\overline{A \cap B}) \in \Sigma_n$. Hence $D, C \in \Pi_n$, so that both $P \vee Q$ and $P \& Q$ are Π_n . ■

Theorem 6.5. Let $Q(x_1, \dots, x_s, y)$ be Σ_n , $n \geq 1$, and let

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y).$$

Then P is also Σ_n .

Proof. Let

$$A = \{x \in N \mid Q((x)_1, \dots, (x)_s, (x)_{s+1})\},$$

$$B = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

We are given that $A \in \Sigma_n$, i.e., that A is $\emptyset^{(n-1)}$ -r.e., and we must show that B is likewise $\emptyset^{(n-1)}$ -r.e.

By Theorem 4.4, we may write

$$A = \{x \in N \mid (\exists t)R(x, t)\},$$

where R is $\emptyset^{(n-1)}$ -recursive. Hence,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow [x_1, \dots, x_s, y] \in A \\ &\Leftrightarrow (\exists t)R([x_1, \dots, x_s, y], t). \end{aligned}$$

Thus,

$$\begin{aligned} x \in B &\Leftrightarrow P((x)_1, \dots, (x)_s) \\ &\Leftrightarrow (\exists y)Q((x)_1, \dots, (x)_s, y) \\ &\Leftrightarrow (\exists y)(\exists t)R([(x)_1, \dots, (x)_s, y], t) \\ &\Leftrightarrow (\exists z)R([(x)_1, \dots, (x)_s, l(z)], r(z)). \end{aligned}$$

By Theorems 1.4 and 4.4, B is $\emptyset^{(n-1)}$ -r.e. ■

Theorem 6.6. Let $Q(x_1, \dots, x_s, y)$ be Π_n , $n \geq 1$, and let

$$P(x_1, \dots, x_s) \Leftrightarrow (\forall y)Q(x_1, \dots, x_s, y).$$

Then P is also Π_n .

Proof. $\sim P(x_1, \dots, x_s) \Leftrightarrow (\exists y) \sim Q(x_1, \dots, x_s, y)$. The result follows from Theorems 6.3 and 6.5. ■

The main result of this section is

Theorem 6.7 (Post's Theorem). A predicate $P(x_1, \dots, x_s)$ is Σ_{n+1} if and only if there is a Π_n predicate $Q(x_1, \dots, x_s, y)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y). \quad (6.1)$$

Proof. If (6.1) holds, with Q a Π_n predicate, it is easy to see that P must be Σ_{n+1} . By Theorem 5.8, Q is certainly itself Σ_{n+1} , and therefore, by Theorem 6.5, P is Σ_{n+1} .

The converse is somewhat more difficult. Let us temporarily introduce the following terminology: we will say that a predicate $P(x_1, \dots, x_s)$ is \exists^{n+1} if it can be expressed in the form (6.1), where Q is Π_n . Then Post's theorem just says that the Σ_{n+1} and the \exists^{n+1} predicates are the same. We have already seen that all \exists^{n+1} predicates are Σ_{n+1} .

Lemma 1. If a predicate is Σ_n , then it is \exists^{n+1} .

Proof. For $n = 0$, the result is obvious. Let $n \geq 1$, and let $P(x_1, \dots, x_s)$ be Σ_n . Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then A is $\emptyset^{(n-1)}$ -r.e., so by Theorem 4.4,

$$A = \{x \in N \mid (\exists t)R(x, t)\},$$

where R is $\emptyset^{(n-1)}$ -recursive. Thus

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists t)R([x_1, \dots, x_s], t).$$

It remains to show that $R([x_1, \dots, x_s], t)$ is Π_n . But in fact, by Theorem 1.4, $R([x_1, \dots, x_s], t)$ is \emptyset^{n-1} -recursive, so that it is actually Δ_n and hence certainly Π_n . ■

Lemma 2. If a predicate is Π_n , then it is \exists^{n+1} .

Proof. If $P(x_1, \dots, x_s)$ is Π_n , we need only set

$$Q(x_1, \dots, x_s, y) \Leftrightarrow P(x_1, \dots, x_s),$$

so that, of course,

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y).$$

Since

$$\{x \in N \mid Q((x)_1, \dots, (x)_s, (x)_{s+1})\} = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

the predicate Q is also Π_n , which gives the result. ■

Lemma 3. If $P(x_1, \dots, x_s, z)$ is \exists^{n+1} and

$$Q(x_1, \dots, x_s) \Leftrightarrow (\exists z)P(x_1, \dots, x_s, z),$$

then Q is \exists^{n+1} .

Proof. We may write

$$P(x_1, \dots, x_s, z) \Leftrightarrow (\exists y)R(x_1, \dots, x_s, z, y),$$

where R is Π_n . Then

$$\begin{aligned} Q(x_1, \dots, x_s) &\Leftrightarrow (\exists z)(\exists y)R(x_1, \dots, x_s, z, y) \\ &\Leftrightarrow (\exists t)R(x_1, \dots, x_s, l(t), r(t)), \end{aligned}$$

which is \exists^{n+1} by Theorem 6.2. ■

Lemma 4. If P and Q are \exists^{n+1} , then so are $P \& Q$ and $P \vee Q$.

Proof. Let us write

$$\begin{aligned} P(x_1, \dots, x_s) &\Leftrightarrow (\exists y)R(x_1, \dots, x_s, y), \\ Q(x_1, \dots, x_s) &\Leftrightarrow (\exists z)S(x_1, \dots, x_s, z), \end{aligned}$$

where R and S are Π_n . Then

$$P(x_1, \dots, x_s) \& Q(x_1, \dots, x_s) \Leftrightarrow (\exists y)(\exists z)[R(x_1, \dots, x_s, y) \& S(x_1, \dots, x_s, z)]$$

and

$$P(x_1, \dots, x_s) \vee Q(x_1, \dots, x_s) \Leftrightarrow (\exists y)(\exists z)[R(x_1, \dots, x_s, y) \vee S(x_1, \dots, x_s, z)].$$

The result follows from Theorem 6.4 and Lemmas 2 and 3. ■

Lemma 5. If $P(x_1, \dots, x_s, t)$ is \exists^{n+1} and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\forall t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is \exists^{n+1} .

Proof. Let

$$P(x_1, \dots, x_s, t) \Leftrightarrow (\exists z)R(x_1, \dots, x_s, t, z),$$

where R is Π_n . Thus,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow (\forall t)_{\leq y}(\exists z)R(x_1, \dots, x_s, t, z) \\ &\Leftrightarrow (\exists u)(\forall t)_{\leq y} R(x_1, \dots, x_s, t, (u)_{t+1}), \end{aligned}$$

where we are using the Gödel number $u = [z_0, z_1, \dots, z_y]$ to encode the sequence of values of z corresponding to $t = 0, 1, \dots, y$. Thus,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow (\exists u)(\forall t)[t > y \vee R(x_1, \dots, x_s, t, (u)_{t+1})] \\ &\Leftrightarrow (\exists u)S(x_1, \dots, x_s, y, u), \end{aligned}$$

where S is Π_n . For $n = 0$, we have used Theorem 6.3 from Chapter 3, and for $n > 0$, we have used the fact that the predicate $t > y$ is recursive (and hence certainly Π_n), and Theorems 6.2, 6.4, and 6.6. ■

We now recall from Section 2 that $u \prec G$ means that

$$\{u\}(i) = G(i) \quad \text{for } 0 \leq i < l(u).$$

Lemma 6. Let $R(x)$ be Σ_n . Then the predicate $u \prec R$ is \exists^{n+1} .

Proof. We have

$$\begin{aligned} u \prec R &\Leftrightarrow (\forall i)_{< l(u)} \{[(r(u))_{i+1} = 1 \& R(i)] \vee [(r(u))_{i+1} = 0 \& \sim R(i)]\} \\ &\Leftrightarrow l(u) = 0 \vee (\exists z)(z + 1 = l(u) \& (\forall i)_{\leq z} \{[(r(u))_{i+1} = 1 \& R(i)] \\ &\quad \vee [(r(u))_{i+1} = 0 \& \sim R(i)]\}). \end{aligned}$$

Thus, using Lemmas 1–5 and the fact that the predicate $\sim R(i)$ is Π_n , we have the result. ■

Now we are ready to complete the proof of Post's theorem. Let $P(x_1, \dots, x_s)$ be any Σ_{n+1} predicate. Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then $A \in \Sigma_{n+1}$, which means that A is B -r.e. for some set $B \in \Sigma_n$. Let $R(x)$ be the characteristic function of B , so that by Theorem 6.1, R is Σ_n . Since A is B -r.e., we are able to write

$$A = \{x \in N \mid f(x) \downarrow\},$$

where f is partially B -computable. Let f be B -computed by a program with number y_0 . Then, using Theorem 2.4 (the finiteness theorem), we have

$$\begin{aligned} x \in A &\Leftrightarrow (\exists t) \text{STP}_R^{(1)}(x, y_0, t) \\ &\Leftrightarrow (\exists t)(\exists u)\{u \prec R \& \text{STP}_{\{u\}}^{(1)}(x, y_0, t)\}. \end{aligned}$$

Thus,

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists t)(\exists u)\{u \prec R \& \text{STP}_{\{u\}}^{(1)}([x_1, \dots, x_s], y_0, t)\}.$$

Therefore by Theorem 2.3 and Lemmas 3, 4, and 6, P is \exists^{n+1} . ■

Now that we know that being Σ_{n+1} and \exists^{n+1} are the same, we may rewrite Lemma 5 as

Corollary 6.8. If $P(x_1, \dots, x_s, t)$ is Σ_n and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\forall t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is also Σ_n .

Also, we can easily obtain the following results:

Corollary 6.9. A predicate $P(x_1, \dots, x_s)$ is Π_{n+1} if and only if there is a Σ_n predicate $Q(x_1, \dots, x_s, y)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (\forall y)Q(x_1, \dots, x_s, y).$$

Proof. Immediate from Post's theorem and Theorem 6.3. ■

Corollary 6.10. If $P(x_1, \dots, x_s, t)$ is Π_n , and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\exists t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is also Π_n .

Proof. Immediate from Corollary 6.8 and Theorem 6.3. ■

We are now in a position to survey the situation. We call a predicate $P(x_1, \dots, x_s)$ *arithmetic* if there is a recursive predicate $R(x_1, \dots, x_s, y_1, \dots, y_n)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (Q_1 y_1)(Q_2 y_2) \cdots (Q_n y_n) R(x_1, \dots, x_s, y_1, \dots, y_n), \quad (6.2)$$

where each of Q_1, \dots, Q_n is either the symbol \exists or the symbol \forall . We say that the Q_i are *alternating* if for $1 \leq i < n$ when Q_i is \exists , then Q_{i+1} is \forall and vice versa. Then we have

Theorem 6.11. (a) Every predicate which is Σ_n or Π_n for any n is arithmetic.

(b) Every arithmetic predicate is Σ_n for some n (and also Π_n for some n).

(c) A predicate is Σ_n (or Π_n) if and only if it can be represented in the form (6.2) with $Q_1 = \exists$ (or $Q_1 = \forall$) and the Q_i alternating.

Proof. Since Σ_0 and Π_0 predicates are just recursive, they are arithmetic. Proceeding by induction, if we know, for some particular n , that all Σ_n and Π_n predicates are arithmetic, then Theorem 6.7 and Corollary 6.9 show that the same is true for Σ_{n+1} and Π_{n+1} predicates. This proves (a).

For (b) we proceed by induction on n , the number of quantifiers. For $n = 0$, we have a Σ_0 (and a Π_0) predicate. If the result is known for $n = k$, then it follows for $n = k + 1$ using Theorems 6.5–6.7 and Corollary 6.9.

Finally, (c) is easily proved by mathematical induction using Theorem 6.7 and Corollary 6.9. ■

7. Classifying Some Unsolvable Problems

We will now see how to apply the arithmetic hierarchy. We begin with the set

$$\text{TOT} = \{z \in N \mid (\forall x)\Phi(x, z)\downarrow\},$$

which consists of all numbers of programs which compute total functions. This set was discussed in Chapter 7, Section 9, where it was shown that TOT is not r.e. Without relying on this previous discussion, we shall obtain much sharper information about TOT.

We begin by observing that

$$\text{TOT} = \{z \in N \mid (\forall x)(\exists t) \text{STP}^{(1)}(x, z, t)\},$$

so that $\text{TOT} \in \Pi_2$. We shall prove

Theorem 7.1. TOT is 1-complete for Π_2 . Therefore, $\text{TOT} \notin \Sigma_2$.

Proof. The second assertion follows from the first by Theorem 5.11.

Since we know that $\text{TOT} \in \Pi_2$, it remains to show that for any $A \in \Pi_2$, we have $A \leq_1 \text{TOT}$. For $A \in \Pi_2$, we can write

$$A = \{w \in N \mid (\forall x)(\exists y)R(x, y, w)\},$$

where R is recursive. Let

$$h(x, w) = \min_y R(x, y, w),$$

so that h is partially computable. Let h be computed by a program with number e . Thus,

$$(\exists y)R(x, y, w) \Leftrightarrow h(x, w) \downarrow \Leftrightarrow \Phi^{(2)}(x, w, e) \downarrow \Leftrightarrow \Phi(x, S_1^1(w, e)) \downarrow,$$

where we have used the parameter theorem. Hence,

$$\begin{aligned} w \in A &\Leftrightarrow (\forall x)(\exists y)R(x, y, w) \\ &\Leftrightarrow (\forall x)[\Phi(x, S_1^1(w, e)) \downarrow] \\ &\Leftrightarrow S_1^1(w, e) \in \text{TOT}. \end{aligned}$$

Since, by Theorem 2.5, $S_1^1(w, e)$ is one-one, we can conclude that

$$A \leq_1 \text{TOT}. \quad \blacksquare$$

As a second simple example we consider

$$\text{INF} = \{z \in N \mid W_z \text{ is infinite}\}.$$

We have

$$z \in \text{INF} \Leftrightarrow (\forall x)(\exists y)(y > x \ \& \ y \in W_z).$$

Now

$$y \in W_z \Leftrightarrow (\exists t) \text{STP}^{(1)}(y, z, t),$$

and hence the predicate $y \in W_z$ is Σ_1 . Using Theorems 6.4 and 6.5, $(\exists y)(y > x \ \& \ y \in W_z)$ is also Σ_1 , and finally $\text{INF} \in \Pi_2$. We shall show that

INF is also 1-complete for Π_2 . By Theorem 3.5, it suffices to show that $\text{TOT} \leq_1 \text{INF}$ since we already know that TOT is 1-complete for Π_2 .

To do this we shall obtain a recursive one-one function $f(x)$ such that

$$W_x = N \quad \text{implies} \quad W_{f(x)} = N$$

and

(7.1)

$$W_x \neq N \quad \text{implies} \quad W_{f(x)} \text{ is finite.}$$

Having done this we will be through since we will have

$$x \in \text{TOT} \Leftrightarrow f(x) \in \text{INF},$$

and therefore,

$$\text{TOT} \leq_1 \text{INF}.$$

The intuitive idea behind the construction of f is that program number $f(x)$ will “accept” a given input z if and only if program number x “accepts” successively inputs $0, 1, \dots, z$. We can write this intuitive idea in the form of an equation as follows:

$$W_{f(x)} = \{z \in N \mid (\forall k)_{\leq z} (k \in W_x)\}.$$

Now it is a routine matter to use the parameter theorem to obtain f . We first note that, by Corollary 6.8, the predicate $(\forall k)_{\leq z} (k \in W_x)$ is Σ_1 . Hence, as above, there is a number e such that

$$\begin{aligned} (\forall k)_{\leq z} (k \in W_x) &\Leftrightarrow \Phi^{(2)}(z, x, e) \downarrow \\ &\Leftrightarrow \Phi(z, S_1^1(x, e)) \downarrow \\ &\Leftrightarrow z \in W_{S_1^1(x, e)}. \end{aligned}$$

Thus the desired function $f(x)$ is simply $S_1^1(x, e)$, which is one-one, as we know from Theorem 2.5.

This completes the proof that INF is 1-complete for Π_2 . Hence also, $\text{INF} \notin \Sigma_2$.

The following notation will be useful:

Definition. Let $A, B, C \subseteq N$. Then we write $A \leq_m (B, C)$ to mean that there is a recursive function f such that

$$x \in A \quad \text{implies} \quad f(x) \in B$$

and

$$x \in \bar{A} \quad \text{implies} \quad f(x) \in C.$$

If f is one-one we write $A \leq_1 (B, C)$.

Thus $A \leq_1 B$ is simply the assertion: $A \leq_1 (B, \bar{B})$.

It will be useful to note that by (7.1), we have actually proved

$$\text{TOT} \leq_1 (\text{TOT}, \bar{\text{INF}}). \quad (7.2)$$

Now, we have

Theorem 7.2. If $A \leq_1 (B, C)$, $B \subseteq D$, and $C \cap D = \emptyset$, then $A \leq_1 D$.

Proof. We have a recursive one-one function f such that

$$x \in A \quad \text{implies} \quad f(x) \in B \quad \text{implies} \quad f(x) \in D$$

and

$$x \in \bar{A} \quad \text{implies} \quad f(x) \in C \quad \text{implies} \quad f(x) \in \bar{D}. \quad \blacksquare$$

Our final example will classify a Σ_3 set, and is considerably more difficult than either of those considered so far.

Theorem 7.3. Let

$$\text{COF} = \{x \in N \mid \bar{W}_x \text{ is finite}\}.$$

Then COF is 1-complete for Σ_3 .

Lemma 1. $\text{COF} \in \Sigma_3$.

Proof.

$$\text{COF} = \{x \in N \mid (\exists n)(\forall k)(k \leq n \vee k \in W_x)\}.$$

Since the predicate in parentheses is Σ_1 , the result follows from Theorem 6.11. \blacksquare

We introduce the notation

$${}_n W_x = \{m \in N \mid \text{STP}^{(1)}(m, x, n)\}.$$

Intuitively, ${}_n W_x$ is the set of numbers which program number x “accepts” in $\leq n$ steps. Clearly,

$$W_x = \bigcup_{n \in N} ({}^n W_x).$$

We also define

$${}^n W_x^r = \{m < r \mid m \in {}^n W_x\}.$$

We write $L(n, x)$ to mean that

$${}_{n+1} W_x^n = {}^n W_x^n.$$

Clearly $L(n, x)$ is a recursive predicate. We write

$$R(x, n) \Leftrightarrow (\forall r)_{\leq n}(r \in W_x) \vee [L(n, x) \& (\exists k)_{< n}(k \notin {}_n W_x)].$$

Since $R(x, n)$ is Σ_1 we can use the parameter theorem, as in the previous example, to find a recursive one-one function $g(x)$ such that

$$W_{g(x)} = \{n \mid R(x, n)\}.$$

Lemma 2. If $x \in \text{TOT}$, then $g(x) \in \text{TOT}$. If $x \notin \text{INF}$, then $g(x) \in \text{COF} - \text{TOT}$.

Proof. If $x \in \text{TOT}$, then $W_x = N$, so that $(\forall r)_{\leq n}(r \in W_x)$ is true for all n . Hence $R(x, n)$ is true for all n , i.e., $W_{g(x)} = N$ and $g(x) \in \text{TOT}$.

Now let $x \notin \text{INF}$, i.e., W_x is finite. Therefore, there is a number n_0 such that for all $n > n_0$, we have

$$W_x = {}_n W_x^n$$

and

$$(\exists k)_{< n}(k \notin W_x).$$

Thus, for $n > n_0$,

$${}_{n+1} W_x^n = {}_n W_x^n,$$

i.e., $L(n, x)$ is true. Thus, $n > n_0$ implies that $R(x, n)$ is true, i.e., that $n \in W_{g(x)}$. We have shown that all sufficiently large integers belong to $W_{g(x)}$. Hence $g(x) \in \text{COF}$. It remains to show that $g(x) \notin \text{TOT}$.

Let s be the least number not in W_x . We consider two cases:

Case 1. $s \notin W_{g(x)}$. Then surely $g(x) \notin \text{TOT}$.

Case 2. $s \in W_{g(x)}$. That is, $R(x, s)$ is true. But $(\forall r)_{\leq s}(r \in W_x)$ must be false because $s \notin W_x$. Hence $L(s, x)$ must be true and $(\exists k)_{< s}(k \notin {}_s W_x)$. Now this number k is less than s , which is the least number not in W_x . Hence $k \in W_x$. Since $k \notin {}_s W_x$,

$$(\exists n)_{n \geq s}[k \notin {}_n W_x \& k \in {}_{n+1} W_x]. \quad (7.3)$$

Now we claim that this number $n \notin W_{g(x)}$, which will show that in this case also $g(x) \notin \text{TOT}$. Thus, suppose that $n \in W_{g(x)}$, i.e., that $R(x, n)$ is true. Since $s \notin W_x$ and $n \geq s$, the condition $(\forall r)_{\leq n}(r \in W_x)$ must be false. Thus we would have to have $L(n, x)$, i.e., ${}_{n+1} W_x^n = {}_n W_x^n$. But by (7.3), $k < s \leq n$, $k \in {}_{n+1} W_x$, and $k \notin {}_n W_x$. This is a contradiction. ■

Lemma 3. $\text{TOT} \leq_1 (\text{TOT}, \text{COF}-\text{TOT})$.

Proof. Let f be the recursive one-one function satisfying (7.1) and let g be as above. Let $h(x) = g(f(x))$. Then using Lemma 2 and (7.1), we have

$$\begin{aligned} x \in \text{TOT} &\quad \text{implies } f(x) \in \text{TOT} \quad \text{implies } h(x) \in \text{TOT}, \\ x \notin \text{TOT} &\quad \text{implies } f(x) \notin \text{INF} \quad \text{implies } h(x) \in \text{COF-TOT}. \end{aligned} \quad \blacksquare$$

Now let $A \in \Sigma_3$. We wish to show that $A \leq_1 \text{COF}$. By Post's theorem, we can write

$$x \in A \Leftrightarrow (\exists n)B(x, n),$$

where B is Π_2 . Using the pairing functions, let

$$C = \{t \in N \mid (\exists n)_{\leq l(t)} B(r(t), n)\}.$$

Thus, $C \in \Pi_2$. By Theorem 7.1, $C \leq_1 \text{TOT}$. Hence, using Lemma 3, $C \leq_1 (\text{TOT}, \text{COF-TOT})$. Let θ be a recursive one-one function such that

$$\begin{aligned} t \in C &\quad \text{implies } \theta(t) \in \text{TOT}, \\ t \notin C &\quad \text{implies } \theta(t) \in \text{COF-TOT}. \end{aligned} \quad (7.4)$$

Consider the Σ_1 predicate $r(z) \in W_{\theta(\langle \iota(z), x \rangle)}$. Using the parameter theorem as usual, we can write this in the form $z \in W_{\psi(x)}$, where ψ is a one-one recursive function. Thus,

$$W_{\psi(x)} = \{\langle k, m \rangle \mid m \in W_{\theta(\langle k, x \rangle)}\}. \quad (7.5)$$

The theorem then follows at once from

Lemma 4. $x \in A$ if and only if $\psi(x) \in \text{COF}$.

Proof. Let $x \in A$. Then $B(x, n)$ is true for some least value of n . Hence, for all $k \geq n$, we have $\langle k, x \rangle \in C$. By (7.4), $\theta(\langle k, x \rangle) \in \text{TOT}$ for all $k \geq n$. Since n is the least value for which $B(x, n)$ is true, $B(x, k)$ is false for $k < n$. Hence, for $k < n$, $\langle k, x \rangle \notin C$. Thus, by (7.4), $\theta(\langle k, x \rangle) \in \text{COF-TOT}$. To recapitulate:

$$\begin{aligned} k \geq n &\quad \text{implies } \theta(\langle k, x \rangle) \in \text{TOT}, \\ \text{and} & \end{aligned} \quad (7.6)$$

$$k < n \quad \text{implies} \quad \theta(\langle k, x \rangle) \in \text{COF-TOT}.$$

Thus, by (7.5) we see that for $k \geq n$, $\langle k, m \rangle \in W_{\psi(x)}$ for all m . For each $k < n$, $W_{\theta(\langle k, x \rangle)}$ contains all but a finite set of m . Thus, altogether, $W_{\psi(x)}$ can omit at most finitely many integers, i.e., $\psi(x) \in \text{COF}$.

Now, let $x \notin A$. Then, $B(x, n)$ is false for all n . Therefore, $\langle k, x \rangle \notin C$ for all k . By (7.4),

$$\theta(\langle k, x \rangle) \in \text{COF-TOT} \quad \text{for all } k \in N,$$

and thus certainly,

$$\theta(\langle k, x \rangle) \notin \text{TOT} \quad \text{for all } k \in N.$$

That is, for every $k \in N$, there exists m such that $m \notin W_{\theta(\langle k, x \rangle)}$, i.e., by (7.5), such that $\langle k, m \rangle \notin W_{\psi(x)}$. Thus, $\overline{W}_{\psi(x)}$ is infinite, and hence $\psi(x) \notin \text{COF}$. ■

Exercises

1. Show that the following sets belong to Σ_3 :
 - (a) $\{x \in N \mid \Phi_x \subseteq f\}$, where f is a recursive function;
 - (b) $\{\langle x, y \rangle \mid x \in N \& y \in N \& W_x - W_y \text{ is finite}\}$.
2. (a) Prove that for each m, n there is a predicate $U(x_1, \dots, x_m, y)$ which is Σ_n , such that for every Σ_n predicate $P(x_1, \dots, x_m)$ there is a number y_0 with

$$P(x_1, \dots, x_m) \Leftrightarrow U(x_1, \dots, x_m, y_0).$$
- (b) State and prove a similar result for Π_n .
3. Use the previous exercise to prove that for each n , $\Pi_n - \Sigma_n \neq \emptyset$.

8. Rice's Theorem Revisited

In Chapter 4, we gave a proof of Rice's theorem (Theorem 7.1) using the recursion theorem. Now we give a different proof which furnishes additional information.

Definition. Let Γ be a set of partially computable functions of one variable. As in Chapter 4, Section 7, we write

$$R_\Gamma = \{t \in N \mid \Phi_t \in \Gamma\}.$$

We call Γ *nontrivial* if $\Gamma \neq \emptyset$ and there is at least one partially computable function $g(x)$ such that $g \notin \Gamma$.

Theorem 8.1 (Strengthened Form of Rice's Theorem). Let Γ be a nontrivial collection of partially computable functions of one variable. Then, $K \leq_1 R_\Gamma$ or $\overline{K} \leq_1 R_\Gamma$, so that R_Γ is not recursive.

Thus not only is R_Γ nonrecursive, but the halting problem can be "solved" using R_Γ as an oracle.

Proof. We recall (Chapter 1, Section 2) that \emptyset is a partially computable function, namely, the nowhere defined function.

Case 1. $\emptyset \notin \Gamma$. Since Γ is nontrivial, it contains at least one function, say f . Since $f \in \Gamma$ and $\emptyset \notin \Gamma$, $f \neq \emptyset$; thus f must be defined for at least one value. Let

$$\Omega(x, t) = \begin{cases} f(t) & \text{if } x \in K \\ \uparrow & \text{if } x \notin K. \end{cases}$$

Since

$$x \in K \Leftrightarrow \Phi(x, x) \downarrow,$$

it is clear that Ω is partially computable. Using the parameter theorem in its strengthened form, we can write

$$\Omega(x, t) = \Phi_{g(x)}(t),$$

where g is a one-one recursive function. Then we have

$$\begin{aligned} x \in K &\quad \text{implies} \quad \Phi_{g(x)} = f \quad \text{implies} \quad g \in R_\Gamma; \\ x \notin K &\quad \text{implies} \quad \Phi_{g(x)} = \emptyset \quad \text{implies} \quad g \notin R_\Gamma. \end{aligned}$$

Thus, $K \leq_1 R_\Gamma$.

Case 2. $\emptyset \in \Gamma$. Now let Δ be the class of all partially computable functions not in Γ . Thus, $R_\Gamma = \bar{R}_\Delta$ and $\emptyset \notin \Delta$. By Case 1, $K \leq_1 R_\Delta$, and hence by Theorem 3.7, $\bar{K} \leq_1 R_\Gamma$.

Exercises

1. State and prove a relativized version of Rice's theorem.
2. (a) Develop a code for partial functions from N to N with finite domains, writing f_n for the n th such function.
 (b) Prove the Rice-Shapiro theorem: R_Γ is r.e. if and only if $\Gamma = \emptyset$ or there is a recursive function $t(x)$ such that

$$\Gamma = \{g \mid (\exists x)(g \supseteq f_{t(x)})\}.$$

9. Recursive Permutations

Definition. A one-one recursive function f whose domain and range are both N is called a *recursive permutation*.

With each recursive permutation f we may associate its *inverse* f^{-1} :

$$f^{-1}(t) = \min_x(t = f(x)).$$

Then, f^{-1} is clearly likewise a recursive permutation.

Definition. Let $A, B \subseteq N$. Then A and B are said to be *recursively isomorphic*, written $A \equiv B$, if there is a recursive permutation f such that $x \in A$ if and only if $f(x) \in B$.

Since a recursive permutation provides what is essentially a mere change of notation, recursively isomorphic sets may be thought of as containing the same “information” presented in different notation.

It is obvious that $A \equiv B$ implies $A \equiv_1 B$. Remarkably, the converse statement is also true:

Theorem 9.1 (Myhill). If $A \equiv_1 B$, then $A \equiv B$.

In our proof of this theorem we shall need to code sequences of ordered pairs of numbers. We shall speak of the *code* of the sequence

$$(a_1, b_1), \dots, (a_n, b_n) \quad (9.1)$$

of pairs of elements of N meaning the number

$$u = \langle n, [\langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle] \rangle.$$

Thus, the numbers a_i, b_i can be retrieved from the code u by using the relations

$$\left. \begin{array}{l} a_i = l((r(u))_i) \\ b_i = r((r(u))_i) \end{array} \right\} \quad i = 1, 2, \dots, l(u).$$

Note that every natural number is the code of a unique finite (possibly empty) sequence of ordered pairs.

We say that the finite sequence (9.1) *associates* A and B , where $A, B \subseteq N$, if

1. $a_i \neq a_j$ for $1 \leq i < j \leq n$;
2. $b_i \neq b_j$ for $1 \leq i < j \leq n$;
3. for each i , $1 \leq i \leq n$, either $a_i \in A$ and $b_i \in B$ or $a_i \notin A$ and $b_i \notin B$.

We shall prove the

Lemma. Let $A \leq_1 B$. Then there is a computable function $k(u, v)$ such that if u codes the sequence (9.1) which associates A and B and $a \notin \{a_1, a_2, \dots, a_n\}$, then there is a b such that $k(u, a)$ codes the sequence

$$(a_1, b_1), \dots, (a_n, b_n), (a, b) \quad (9.2)$$

which also associates A and B .

Proof. Let f be a recursive one-one function such that

$$x \in A \quad \text{if and only if} \quad f(x) \in B. \quad (9.3)$$

We provide an algorithm for computing b from u and a . $k(u, a)$ can then be set equal to the code of (9.2), i.e.

$$k(u, a) = \langle l(u) + 1, r(u) \cdot p_{l(u)+1}^{(a, b)} \rangle.$$

The numbers $f(a_1), f(a_2), \dots, f(a_n), f(a)$ are all distinct, because f is one-one. Hence, at least one of these $n + 1$ numbers does not belong to the set $\{b_1, b_2, \dots, b_n\}$. Our algorithm for obtaining b begins by computing $f(a)$. If $f(a) \notin \{b_1, b_2, \dots, b_n\}$, we set $b = f(a)$. Otherwise, $f(a) = b_i$ for some i and we try $f(a_i)$, because

$$a \in A \Leftrightarrow f(a) = b_i \in B \Leftrightarrow a_i \in A \Leftrightarrow f(a_i) \in B.$$

If $f(a_i) \notin \{b_1, b_2, \dots, b_n\}$, we set $b = f(a_i)$. Otherwise, if $f(a_i) = b_j$, we continue the process, trying $f(a_j)$. By (1) and (2), none of the a_i and b_i obtained in this way duplicate previous ones. Thus, by our above remark the process must terminate in a value b . Using (9.3), we see that either $a \in A$ and $b \in B$ or $a \notin A$ and $b \notin B$. ■

Proof of Theorem 9.1. Since $A \leq_1 B$, by the Lemma there is a computable function $k(u, v)$ such that if u codes (9.1) which associates A and B and $a \notin \{a_1, a_2, \dots, a_n\}$, then for some b , $k(u, a)$ codes the sequence (9.2) which also associates A and B . But since $B \leq_1 A$, we can also apply the Lemma to obtain a computable function $\bar{k}(u, v)$ such that if u codes (9.1) which associates A and B and $b \notin \{b_1, b_2, \dots, b_n\}$, then for some a , $\bar{k}(u, b)$ codes the sequence (9.2) which likewise associates A and B .

We let $v(0) = 0$, which codes the empty sequence. (Note that the empty sequence does associate A and B .) We let

$$v(2x + 1) = \begin{cases} v(2x) & \text{if } x \text{ is one of the left components} \\ & \text{of the sequence coded by } v(2x) \\ k(v(2x), x) & \text{otherwise;} \end{cases}$$

$$v(2x + 2) = \begin{cases} v(2x + 1) & \text{if } x \text{ is one of the right components} \\ & \text{of the sequence coded by } v(2x + 1) \\ \bar{k}(v(2x + 1), x) & \text{otherwise.} \end{cases}$$

Thus, we have

- (1) v is a computable function.
- (2) For each x , $v(x)$ codes a sequence which associates A and B .
- (3) The sequence coded by $v(x + 1)$ is identical to, or is an extension of, the sequence coded by $v(x)$.
- (4) For each $a \in N$, there is an x such that a pair (a, b) occurs in the sequence coded by $v(x)$. (In fact, we can take $x = 2a + 1$.)

(5) For each $b \in N$, there is an x such that a pair (a, b) occurs in the sequence coded by $v(x)$. (In fact, we can take $x = 2b + 2$.)

We now define the function f by setting $f(a)$ to be the number b such that the pair (a, b) appears in the sequence coded by some $v(x)$. b is uniquely determined because all the $v(x)$ code sequences which associate A and B , f is clearly computable. In fact,

$$f(a) = \min_b (\exists i)_{\leq l(v(2a+1))} [(r(v(2a+1)))_i = \langle a, b \rangle].$$

By (5), the range of f is N ; thus f is a recursive permutation and hence, $A \equiv B$. ■

Exercises

1. Prove that $K \equiv U$, where U is defined in Exercise 3.1.
2. Prove that

$$A \oplus \bar{A} \equiv \overline{A \oplus \bar{A}}.$$

Degrees of Unsolvability and Post's Problem

1. Turing Degrees

Let \leq_Q be an arbitrary reducibility in the sense of Chapter 16, Section 3. Recall that we write $A \equiv_Q B$ to mean that $A \leq_Q B$ and $B \leq_Q A$. For any set A we write

$$\deg_Q(A) = \{B \subseteq N \mid A \equiv_Q B\}.$$

$\deg_Q(A)$ is called the *Q-degree* of A .

Theorem 1.1. We have

1. $A \in \deg_Q(A)$.
2. $A \equiv_Q B$ if and only if $\deg_Q(A) = \deg_Q(B)$.
3. If $\deg_Q(A) \cap \deg_Q(B) \neq \emptyset$, then $\deg_Q(A) = \deg_Q(B)$.

Proof. We are simply noting that by Theorem 3.4 in Chapter 16, \equiv_Q is an “equivalence relation” and the *Q-degrees* are the corresponding equivalence classes. For readers who are unfamiliar with these matters, we give the easy details.

1. Since $A \equiv_Q A$, $A \in \deg_Q(A)$.
2. Let $A \equiv_Q B$ and let $C \in \deg_Q(A)$. Then $A \equiv_Q C$. Therefore, $B \equiv_Q C$ and $C \in \deg_Q(B)$. Thus, $\deg_Q(A) \subseteq \deg_Q(B)$. By symmetry, $\deg_Q(B) \subseteq \deg_Q(A)$. Conversely, let $\deg_Q(A) = \deg_Q(B)$. By (1), $B \in \deg_Q(A)$, i.e., $A \equiv_Q B$.
3. Let $C \in \deg_Q(A) \cap \deg_Q(B)$. Then $A \equiv_Q C$ and $B \equiv_Q C$. Therefore $A \equiv_Q B$ and by (2), $\deg_Q(A) = \deg_Q(B)$. ■

In fact our main interest is in the Turing degrees, and from now on we write $\deg(A)$ for $\deg_t(A)$. It is conventional to use boldface letters to represent degrees.

Often, when we define operations and relations on degrees, we will do so in terms of particular sets which belong to the degrees in question. Then, in order to be certain that what we have defined is really an operation (or relation) on the degrees, we must check that our definition is *invariant*, which means that we would get the same result if we replaced the sets we have used with other “representatives” of the same degrees.

Here is a first example:

Definition. $\mathbf{a} \leq \mathbf{b}$ if there are sets A, B such that $\deg(A) = \mathbf{a}$, $\deg(B) = \mathbf{b}$, and $A \leq_t B$. We also write $\mathbf{a} \not\leq \mathbf{b}$ if it is not the case that $\mathbf{a} \leq \mathbf{b}$.

Invariance. Suppose that $\deg(A) = \deg(A_1)$ and $\deg(B) = \deg(B_1)$, i.e., $A \equiv_t A_1$, $B \equiv_t B_1$. Then we must check that

$$A \leq_t B \quad \text{if and only if} \quad A_1 \leq_t B_1.$$

This is easy:

$$A \leq_t B \quad \text{implies} \quad A_1 \leq_t A \leq_t B \leq_t B_1,$$

and

$$A_1 \leq_t B_1 \quad \text{implies} \quad A \leq_t A_1 \leq_t B_1 \leq_t B.$$

Definition. $\mathbf{0} = \deg(\emptyset)$.

By Theorem 1.6 in Chapter 16, $\deg(A) = \mathbf{0}$ if and only if A is a recursive set.

Definition. $\mathbf{a}' = \deg(A')$ where $\deg(A) = \mathbf{a}$.

Invariance. We must check that $A \equiv_t B$ implies $A' \equiv_t B'$. But by Theorem 4.9 in Chapter 16, $A \equiv_t B$ implies $A' \equiv_t B'$.

Definition. $\mathbf{a} \vee \mathbf{b} = \deg(A \oplus B)$, where $\deg(A) = \mathbf{a}$ and $\deg(B) = \mathbf{b}$.

$\mathbf{a} \vee \mathbf{b}$ is called the *join* of the degrees \mathbf{a} and \mathbf{b} . Here, as we recall,

$$A \oplus B = \{2x \mid x \in A\} \cup \{2x + 1 \mid x \in B\}.$$

Invariance. Let $A \equiv_t A_1$, $B \equiv_t B_1$. Then we have, using Theorem 3.11 in Chapter 16,

$$A_1 \leq_t A \leq_t A \oplus B,$$

$$B_1 \leq_t B \leq_t A \oplus B,$$

and so

$$A_1 \oplus B_1 \leq_t A \oplus B.$$

By symmetry,

$$A \oplus B \leq_t A_1 \oplus B_1.$$

Definition. $a < b$ means that $a \leq b$ but $a \neq b$.

Theorem 1.2. For all degrees a, b, c :

- (1) $a \leq a$.
- (2) $a \leq b$ and $b \leq c$ implies $a \leq c$.
- (3) $\mathbf{0} \leq a$.
- (4) $a \leq b$ and $b \leq a$ implies $a = b$.
- (5) $a < a'$.
- (6) $a \leq a \vee b$, $b \leq a \vee b$.
- (7) If $a \leq c$ and $b \leq c$, then $a \vee b \leq c$.
- (8) $a \vee \mathbf{0} = \mathbf{0} \vee a = a$.

Proof. (1) and (2) are obvious simply because \leq is a reducibility. (3) follows at once from Theorem 1.1 in Chapter 16. For (4), let $\deg(A) = a$, $\deg(B) = b$. Then $A \leq_t B$ and $B \leq_t A$, i.e., $A \equiv_t B$. Thus, $a = \deg(A) = \deg(B) = b$. For (5), let $\deg(A) = a$. Since A is A -r.e., Theorem 4.8 in Chapter 16 implies that $A \leq_1 A'$; so certainly $a \leq a'$. But by Theorem 4.7 in Chapter 16, $A' \not\leq_t A$, and hence $a' \neq a$. (6) and (7) are immediate consequences of Theorem 3.11 in Chapter 16. (8) follows at once from (1), (3), (6), and (7). ■

Note that

$$\mathbf{0} < \mathbf{0}' < \mathbf{0}'' < \dots$$

For any degree a and $n \in N$ we define $a^{(n)}$ as follows:

$$a^{(0)} = a$$

$$a^{(n+1)} = (a^n)'.$$

Thus, $\mathbf{0}^{(n)} = \deg(\emptyset^{(n)})$. We have

Theorem 1.3. Let A be m-complete for Σ_n or Π_n , $n \geq 1$. Then $\deg(A) = \mathbf{0}^{(n)}$.

Proof. Let A be m-complete for Σ_n . Then by Corollary 5.5 in Chapter 16, $A \equiv_m \emptyset^{(n)}$. Hence $\deg(A) = \deg(\emptyset^{(n)}) = \mathbf{0}^{(n)}$.

If A is m-complete for Π_n , then by Theorem 3.7 in Chapter 16, \bar{A} is m-complete for Σ_n . Hence, $\deg(A) = \deg(\bar{A}) = \mathbf{0}^{(n)}$. ■

Thus, the results of Chapter 16, Section 7 show that

$$\deg(\text{TOT}) = \deg(\text{INF}) = 0''$$

and

$$\deg(\text{COF}) = 0'''.$$

Examples like these suggest the conjecture:

If A is arithmetic, then $\deg(A) = 0^{(n)}$ for some n .

This, however, is *FALSE*. In fact, as we shall soon see, there are degrees a such that $0 < a < 0'$. Of course, if $\deg(A) = a$ for such a degree a , we must have $A \in \Delta_2$.

Definition. A degree a is called an *r.e. degree* if there is an r.e. set A such that $\deg(A) = a$.

Much early work on r.e. sets and on degrees was motivated by efforts to solve

Post's Problem. Is there an r.e. degree a such that $0 < a < 0'$?

Later we shall see that the answer to this question is *yes*; but the proof is not at all easy.

2. The Kleene–Post Theorem

We write $a|b$, and say that the degrees a and b are *incomparable*, to mean that $a \not\leq b$ and $b \not\leq a$. In this section we shall prove

Theorem 2.1 (Kleene–Post Theorem). There are degrees a, b such that $a|b$ and $a, b \leq 0'$.

Before beginning to work on the proof we note the following consequence of the Kleene–Post theorem:

Corollary 2.2. There is a degree a such that $0 < a < 0'$.

Proof. Let a be as in the theorem. Then we have $0 \leq a \leq 0'$. Thus, we need only show that $a \neq 0$ and $a \neq 0'$. But, if $a = 0$, then $a \leq b$, and if $a = 0'$, then $b \leq a$. ■

Note that this corollary is not yet a solution to Post's problem as posed in the previous section, because we are not claiming that the degree a is r.e.

In fact, we will prove

Theorem 2.3. There are total unary functions F, G such that F and G are both \emptyset' -recursive, but F is not G -recursive and G is not F -recursive.

Before proving Theorem 2.3, we show that it implies the Kleene–Post theorem:

Proof That Theorem 2.3 Implies Theorem 2.1. Let F, G be as in Theorem 2.3. Let

$$A = \{\langle x, y \rangle \mid y = F(x)\},$$

$$B = \{\langle x, y \rangle \mid y = G(x)\},$$

$$\mathbf{a} = \deg(A), \quad \mathbf{b} = \deg(B).$$

Obviously, F is A -recursive, A is F -recursive, G is B -recursive, and B is G -recursive. Thus, if $A \leq_t B$, then we would have that F is G -recursive, and if $B \leq_t A$, we would have that G is F -recursive. We conclude that $\mathbf{a} \mid \mathbf{b}$. Finally, we have $A, B \leq_t \emptyset'$, so that $\mathbf{a}, \mathbf{b} \leq \emptyset'$. ■

Proof of Theorem 2.3. This is a construction by stages (like the proof of the speedup theorem in Chapter 14). We seek total functions F, G which satisfy the infinite set of *conditions*

$$(1_y) \quad F(x) \neq \Phi_G(x, y),$$

$$(2_y) \quad G(x) \neq \Phi_F(x, y),$$

$y = 0, 1, 2, \dots$. That is, we seek to guarantee that for no number y will the program whose number is y either G -compute F or F -compute G . Thus, F and G must be constructed so as to be sure that for each y there is some x for which the relations $(1_y), (2_y)$ are satisfied. This involves a kind of diagonalization which is considerably more sophisticated than those we encountered in Chapter 4.

We will make explicit use of the definition of a function as a set of ordered pairs so that, for example, we will write $f \subseteq g$, where f and g are partial functions. Of course, this simply means that whenever $f(m) \downarrow$, then $g(m) = f(m)$. We will say that a pair f, g of partial unary functions are *compatible*, written $f \sim g$, to mean that there is a partial function h such that

$$f \subseteq h, \quad g \subseteq h.$$

We will use the notation $\{u\}$ defined in Eq. (2.1) of Chapter 16. Note that if $\{u\} \sim \{v\}$, then either $\{u\} \subseteq \{v\}$ or $\{v\} \subseteq \{u\}$.

Our construction will proceed through stages $s = 0, 1, 2, 3, \dots$. At each stage s , we will define numbers u_s, v_s . This will be done in such a way that

$$\{u_s\} \subseteq \{u_{s+1}\}, \quad \{v_s\} \subseteq \{v_{s+1}\}.$$

The functions F, G will then be given by

$$F = \bigcup_{s \in N} \{u_s\}, \quad G = \bigcup_{s \in N} \{v_s\}. \quad (2.1)$$

We begin at stage 0 with $u_0 = v_0 = 2 = \langle 0, 1 \rangle$, so that $\{u_0\} = \{v_0\} = \emptyset$. At stage $s = 2y + 1$ we will deal with condition (1_y) , and at stage $s = 2y + 2$ we will deal with condition (2_y) .

Now let $s = 2y + 1$, where we assume u_t, v_t are defined for $t < s$. Let $n = l(u_{s-1})$. Then n is the least integer such that $\{u_{s-1}\}(n) \uparrow$. We distinguish two cases:

Case 1. There is a number w such that $\{w\} \sim \{v_{s-1}\}$ and $n \in W_y^{(w)}$. Then we define

$$v_s = \begin{cases} v_{s-1} & \text{if } \{w\} \subseteq \{v_{s-1}\} \\ w & \text{if } \{v_{s-1}\} \subset \{w\}; \end{cases}$$

$$k = \Phi_{\{w\}}(n, y) + 1 \quad (\text{the diagonalization!});$$

$$u_s = \langle n + 1, r(u_{s-1}) \cdot p_{n+1}^k \rangle.$$

Thus,

$$\{u_s\}(x) = \begin{cases} \{u_{s-1}\}(x) & \text{for } 0 \leq x < n \\ k & \text{for } x = n \\ \uparrow & \text{for } x > n. \end{cases}$$

Case 2. Otherwise. Then we define

$$v_s = v_{s-1}.$$

u_s will be defined just as in Case 1 but with $k = 0$ (a “default” value).

Next let $s = 2y + 2$, and again let u_t, v_t be defined for $t < s$. Then we proceed exactly as for $s = 2y + 1$, except that we interchange the roles of u_{s-1}, u_s with v_{s-1}, v_s , respectively.

It is clear from the definition that $\{u_{s-1}\} \subseteq \{u_s\}$, $\{v_{s-1}\} \subseteq \{v_s\}$. Also, for odd s , the domain of $\{u_s\}$ contains exactly one element not in the domain of $\{u_{s-1}\}$, and for even $s > 0$, the domain of $\{v_s\}$ contains exactly one element not in the domain of $\{v_{s-1}\}$. Since these domains must be of the form $\{n \in N \mid n \leq z\}$, it follows that for $s \geq 2x + 1$, $\{u_s\}(x) \downarrow$, and for $s \geq 2x + 2$, $\{v_s\}(x) \downarrow$. We now define F, G by (2.1) above. We have

Lemma 1. The functions F, G are total and \emptyset' -recursive.

Proof. We have

$$F(x) = \{u_{2x+1}\}(x), \quad G(x) = \{v_{2x+2}\}(x). \quad (2.2)$$

Thus, F , G are clearly total and recursive in the functions mapping s to u_s , v_s , respectively. These functions in turn are recursive in the predicate

$$(\exists w)[\{w\} \sim \{v\} \& n \in W_y^{\{w\}}].$$

This predicate is equivalent to

$$(\exists w)[\{w\} \sim \{v\} \& (\exists t) \text{STP}_{\{w\}}^{(1)}(n, y, t)],$$

which by Theorem 2.3 in Chapter 16 is Σ_1 , and hence recursive in \emptyset' . ■

Lemma 2. F is not G -recursive.

Proof. Suppose that F were G -recursive. Then for some number y_0 ,

$$F(x) = \Phi_G(x, y_0). \quad (2.3)$$

Let $s = 2y_0 + 1$, and let n be the least integer such that $\{u_{s-1}\}(n) \uparrow$. We consider the same two cases that were distinguished at stage s .

Case 1. In that case we have $\{w\} \subseteq \{v_s\} \subseteq G$, and

$$F(n) = \{u_s\}(n) = \Phi_{\{w\}}(n, y_0) + 1 = \Phi_G(n, y_0) + 1,$$

which contradicts (2.3).

Case 2. In this case we will show that $\Phi_G(n, y_0) \uparrow$, which again will contradict (2.3) because F is total. Thus, suppose $\Phi_G(n, y_0) \downarrow$. Then for some t , $\Phi_{\{v_t\}}(n, y_0) \downarrow$. Now for this t , $\{v_t\} \sim \{v_{s-1}\}$ and $n \in W_{y_0}^{\{v_t\}}$. But then we would be in Case 1. ■

Lemma 3. G is not F -recursive.

Proof. By symmetry. ■

This completes the proof of Theorem 2.3. ■

In order to use a construction in stages like the one used here to solve Post's problem, a new idea is required. In order to obtain r.e. degrees, it will be necessary to give up an arrangement where each "requirement" is settled at a preassigned stage. Without the information available from a \emptyset' oracle, all that is possible is to *attempt* to satisfy requirements at each stage, leaving open the possibility that the arrangements made at the various stages generate "conflicts." The ingenious *priority method*, invented independently by Friedberg and Muchnik, makes it possible to resolve these conflicts. We will exhibit a "priority" construction at the end of this chapter.

3. Creative Sets—Myhill's Theorem

The first efforts toward the solution of Post's problem, by Post himself in 1944, initiated a program of classifying r.e. sets in terms of properties of their complements. It was hoped that information about the Turing degrees of the sets could be obtained in this manner. Although this hope turned out to be futile, the program remained of considerable interest. Post began by distinguishing two kinds of r.e. sets, *creative sets* and *simple sets* which, as Post noted, were “poles apart.” In this and the next section, we shall develop some of the properties of these sets—including some discovered by Post and some found only by later investigators.

As in Chapter 16, we will need to use the parameter theorem to write a Σ_1 -predicate $P(x, t)$ in the form $x \in W_{f(t)}$, where f is a one-one computable function. We recall that this can be done by writing

$$\begin{aligned} P(x, t) &\Leftrightarrow \Phi^{(2)}(x, t, e) \downarrow \\ &\Leftrightarrow \Phi_{S_1^1(t, e)}(x) \downarrow \\ &\Leftrightarrow x \in W_{S_1^1(t, e)}, \end{aligned}$$

using the strengthened form of the parameter theorem to conclude that $S_1^1(t, e)$ is one-one.

Definition. A set P is called *productive* if there is a partially computable function ϕ , called a *productive function for P* , such that for all $x \in N$, we have

$$W_x \subseteq P \quad \text{implies} \quad \phi(x) \downarrow \text{ and } \phi(x) \in P - W_x.$$

An r.e. set C is called *creative* if \bar{C} is productive.

Theorem 3.1. A productive set always has a total productive function.

Proof. Let P be productive with a partially computable productive function ϕ . We show how to convert ϕ into a total (hence computable) productive function. The Σ_1 -predicate $u \in W_x \& \phi(x) \downarrow$ can be written in the form $u \in W_{g(x)}$, where g is computable. Hence,

$$W_{g(x)} = \begin{cases} W_x & \text{if } \phi(x) \downarrow \\ \emptyset & \text{otherwise.} \end{cases}$$

Now, if $\phi(x) \uparrow$, then $W_{g(x)} = \emptyset$, so trivially, $W_{g(x)} \subseteq P$. Hence, by definition of productive function, $\phi(g(x)) \downarrow$. Hence, for all $x \in N$, either $\phi(x) \downarrow$ or $\phi(g(x)) \downarrow$. Let us write

$$\phi(x) = \Phi_c(x),$$

$$\phi(g(x)) = \Phi_f(x),$$

and let $h(x)$ be the computable function computed by the program

```

[A]    IF STP(1)(X, e, T) GOTO B
        IF STP(1)(X, f, T) GOTO C
        T ← T + 1
        GOTO A
[B]    Y ← φ(X)
        GOTO E
[C]    Y ← φ(g(X))

```

Thus, for each x , $h(x) = \phi(x)$ or $h(x) = \phi(g(x))$. When both are defined, the value of h is whichever is computed in fewer steps. Now, let $W_x \subseteq P$. Then, $\phi(x) \downarrow$. Hence $W_{g(x)} = W_x$. Thus, $\phi(g(x))$ and $\phi(x)$ both belong to $P - W_x$. Therefore, in any case, $h(x) \in P - W_x$. ■

In fact, we can do even better:

Theorem 3.2. A productive set always has a productive function which is total and one-one.

Proof. Using the previous theorem, let h be a total productive function for the productive set P . We shall show how to convert h into a one-one total productive function for P . We begin by writing the Σ_1 -predicate:

$$t \in W_x \vee t = h(x)$$

in the form $t \in W_{f(x)}$, where f is computable. Thus,

$$W_{f(x)} = W_x \cup \{h(x)\}.$$

We observe that if $W_x \subseteq P$, then, since $h(x) \in P$, we have also that $W_{f(x)} \subseteq P$.

We write

$$t_0 = x,$$

$$t_{i+1} = f(t_i),$$

$i = 1, 2, 3, \dots$, so that we have $W_x \subseteq P$ implies $W_{t_i} \subseteq P$ for all $i \in N$. Since $W_{t_{i+1}} = W_{t_i} \cup \{h(t_i)\}$, we see that if $W_x \subseteq P$, then $h(t_i) \in W_{t_{i+1}}$, but $h(t_i) \notin W_{t_i}$. Hence, if $W_x \subseteq P$ we can conclude that the numbers $h(t_i)$, $i = 0, 1, 2, \dots$, are all distinct.

Now, we show how to convert our given total productive function h into a one-one total productive function k . We begin by setting $k(0) = h(0)$. Assume that $k(t)$ has been defined for all $t < x$, and let $A_x = \{k(t) \mid t < x\}$. We will define $k(x)$ so that $k(x) \notin A_x$. We compute the numbers $h(t_i)$ as above, continuing until either a repetition occurs or a number not in A_x is encountered. In the first case, by what has been said, we must have $W_x \not\subseteq P$, and

nothing is demanded of the productive function; hence in this first case we can just take $k(x)$ to be the least number not in A_x . In the second case, we simply take $k(x)$ to be the number $h(t_i)$ we have encountered that is not in A_x . If $W_x \subseteq P$ and $k(x) = h(t_i)$, we know that $k(x) \in P - W_{t_i}$ and hence $k(x) \in P - W_x$. ■

Theorem 3.3. A productive set is not r.e.

Proof. Let P be productive with total productive function f , and suppose that P were r.e., say, $P = W_i$. But then, we would have $f(i) \in P - W_i = \emptyset$. ■

Corollary 3.4. A creative set is not recursive.

Proof. Since the complement of a creative set is productive, it cannot be r.e. ■

Theorem 3.5. K is creative.

Proof. We take $f(n) = I(n) = n$, and show that f is productive for \bar{K} . Recalling that

$$n \in K \Leftrightarrow n \in W_n, \quad (3.1)$$

we see that the condition

$$W_n \subseteq \bar{K}$$

implies that $n \notin W_n$. (For if $n \in W_n$, then $n \in \bar{K}$, i.e., $n \notin W_n$, a contradiction.) Thus,

$$W_n \subseteq \bar{K} \quad \text{implies} \quad n \notin W_n.$$

But $n \notin W_n$ implies $n \in \bar{K}$ by (3.1). Thus,

$$W_n \subseteq \bar{K} \quad \text{implies} \quad n \in \bar{K} - W_n. \quad \blacksquare$$

Theorem 3.6. Let P be productive. Then there is an infinite r.e. set U such that $U \subseteq P$.

The idea behind the proof is that if we have a finite set of elements $a_1, \dots, a_k \in P$, then we should be able to find a number i such that

$$W_i = \{a_1, \dots, a_k\} \subseteq P.$$

Thus if f is a total productive function for P , we should then be able to obtain an element $f(i) \in P - \{a_1, \dots, a_k\}$ which we can call a_{k+1} . The desired set is then simply $U = \{a_1, a_2, \dots\}$.

Proof of Theorem 3.6. Let $f(n)$ be a total productive function for P . Since the predicate

$$m \in W_n \vee m = f(n)$$

is Σ_1 , we can use the parameter theorem to obtain a recursive function $\phi(n)$ such that for all n ,

$$W_{\phi(n)} = W_n \cup \{f(n)\}. \quad (3.2)$$

Now, let p be some number such that $W_p = \emptyset$. Let g be the primitive recursive function defined by

$$g(0) = p,$$

$$g(n + 1) = \phi(g(n)).$$

Then by (3.2),

$$W_{g(n+1)} = W_{g(n)} \cup \{f(g(n))\}.$$

We define

$$U = \bigcup_{n \in N} W_{g(n)}.$$

Lemma 1. U is an r.e. set.

Proof.

$$\begin{aligned} x \in U &\Leftrightarrow (\exists n)(x \in W_{g(n)}) \\ &\Leftrightarrow (\exists n)(\Phi(x, g(n)) \downarrow), \end{aligned}$$

which is a Σ_1 predicate. ■

Lemma 2. For each k , $W_{g(k)}$ contains exactly k elements and $W_{g(k)} \subseteq P$.

Proof. For $k = 0$, $W_{g(0)} = \emptyset$ contains 0 elements, and of course $\emptyset \subseteq P$. Assume the result known for $k = m$, i.e., $W_{g(m)} \subseteq P$ and $W_{g(m)}$ contains exactly m elements. Then $f(g(m)) \in P - W_{g(m)}$. Thus,

$$W_{g(m+1)} = W_{g(m)} \cup \{f(g(m))\}$$

is a subset of P that contains all the elements of $W_{g(m)}$ plus one additional element, and thus consists of $m + 1$ elements. ■

It follows at once from Lemma 2 that U is an infinite subset of P . By Lemma 1, it is r.e. Thus the proof of the theorem is complete. ■

We have given only one example of a creative set, but in fact we have

Theorem 3.7. If C is m -complete for Σ_1 , then C is creative.

Proof. Let us consider the r.e. set

$$U = \{x \in N \mid l(x) \in W_{r(x)}\},$$

where, of course, l and r are the pairing functions. Since C is m-complete for Σ_1 , we have: $U \leq_m C$. Let $U = \{t \in N \mid f(t) \in C\}$, where f is recursive. Then

$$x \in W_y \Leftrightarrow \langle x, y \rangle \in U \Leftrightarrow f(\langle x, y \rangle) \in C. \quad (3.3)$$

Let us use the parameter theorem to obtain a recursive function g such that

$$f(\langle x, x \rangle) \in W_n \Leftrightarrow x \in W_{g(n)}, \quad (3.4)$$

which we can do, since the predicate on the left is Σ_1 . Now, suppose that $W_n \subseteq \bar{C}$. Then, using (3.3) and (3.4),

$$\begin{aligned} x \in W_{g(n)} &\quad \text{implies} \quad f(\langle x, x \rangle) \in W_n \\ &\quad \text{implies} \quad f(\langle x, x \rangle) \in \bar{C} \\ &\quad \text{implies} \quad x \notin W_x. \end{aligned}$$

Setting $x = g(n)$ (i.e., “diagonalizing”),

$$g(n) \in W_{g(n)} \quad \text{implies} \quad g(n) \notin W_{g(n)}.$$

Hence,

$$g(n) \notin W_{g(n)}. \quad (3.5)$$

By (3.4),

$$f(\langle g(n), g(n) \rangle) \notin W_n.$$

On the other hand, by (3.3) and (3.5),

$$f(\langle g(n), g(n) \rangle) \notin C.$$

Thus,

$$f(\langle g(n), g(n) \rangle) \in \bar{C} - W_n. \quad \blacksquare$$

Remarkably, the converse of the result just obtained is likewise true, and even with m-complete replaced by 1-complete. In obtaining this result we will need a strengthened form of the recursion theorem (Theorem 6.1 in Chapter 4), which we give only in the one variable case:

Theorem 3.8 (Parameterized Recursion Theorem). Let $g(z, y, x)$ be partially computable. Then there is a one-one recursive function $t(y)$ such that

$$g(t(y), y, x) = \Phi_{t(y)}(x).$$

For each fixed value of y , the existence of the corresponding number $t(y)$ follows from the recursion theorem. The additional content of the present result is that the number $t(y)$ is computable from y and by a one-one function. The proof follows the lines of the original version in Chapter 4.

Proof. We have for some number e

$$g(S_1^2(y, z, z), y, x) = \Phi^{(3)}(x, y, z, e) = \Phi(x, S_1^2(y, z, e)).$$

The result follows on setting $z = e$ and $t(y) = S_1^2(y, e, e)$, since by Theorem 2.5 in Chapter 16, $t(y)$ is one-one. ■

Theorem 3.9 (Parameterized Fixed Point Theorem). Let $f(x, y)$ be a given recursive function. Then, there is a one-one recursive function $t(y)$ such that for all x, y

$$\Phi_{f(t(y), y)}(x) = \Phi_{t(y)}(x).$$

Proof. Let $g(z, y, x) = \Phi_{f(z, y)}(x)$. Applying the previous theorem, there is a one-one recursive function $t(y)$ such that

$$\Phi_{f(t(y), y)}(x) = g(t(y), y, x) = \Phi_{t(y)}(x).$$

Using this result, we can prove

Theorem 3.10 (Myhill's Theorem). If P is productive, then $\bar{K} \leq_1 P$.

Proof. Let h be a one-one total productive function for P . Using the parameter theorem, the Σ_1 -predicate

$$z = h(x) \& y \in K$$

can be written in the form

$$z \in W_{f(x, y)},$$

where f is recursive. Thus,

$$W_{f(x, y)} = \begin{cases} \{h(x)\} & \text{if } y \in K \\ \emptyset & \text{otherwise.} \end{cases}$$

Applying the previous theorem, there is a one-one recursive function $t(y)$ such that

$$W_{t(y)} = W_{f(t(y), y)} = \begin{cases} \{h(t(y))\} & \text{if } y \in K \\ \emptyset & \text{otherwise.} \end{cases}$$

Now, suppose that $y \in K$. Then $W_{t(y)} = \{h(t(y))\}$. Since h is a productive function for P , it must have been the case that $W_{t(y)} \not\subseteq P$, i.e., that $h(t(y)) \notin P$.

If, on the other hand, $y \notin K$, then $W_{t(y)} = \emptyset$, so that certainly $W_{t(y)} \subseteq P$, and therefore $h(t(y)) \in P$. We have thus proved that

$$y \in \bar{K} \Leftrightarrow h(t(y)) \in P.$$

As was observed in the proof of Theorem 3.3 in Chapter 16, a function obtained by the composition of two recursive one-one functions is itself a one-one recursive function. Thus, $h(t(y))$ is such, and we can conclude that $\bar{K} \leq_1 P$. ■

Now, we can easily prove a truly remarkable result.

Theorem 3.11 (Myhill's Theorem—Second Form). The following assertions are equivalent:

- (1) S is 1-complete for Σ_1 .
- (2) S is m-complete for Σ_1 .
- (3) S is creative.

Proof. That (1) implies (2) is trivial. The content of Theorem 3.7 is that (2) implies (3). It therefore remains only to show that (3) implies (1).

Thus, let S be creative, i.e., let S be r.e., where \bar{S} is productive. Using Theorem 3.10 we see that $\bar{K} \leq_1 \bar{S}$, or by Theorem 3.7 in Chapter 16, $K \leq_1 S$. Let R be any r.e. set. By Theorem 3.9 in Chapter 16, $R \leq_1 K$, so that $R \leq_1 S$. This proves that S is 1-complete for Σ_1 . ■

Corollary 3.12 (Myhill's Theorem—Third Form). All creative sets are recursively isomorphic.

Proof. Let C, C' be creative sets. By the theorem, they are each 1-complete for Σ_1 . Hence, in particular $C \leq_1 C'$ and $C' \leq_1 C$, i.e., $C \equiv_1 C'$. By Theorem 9.1 in Chapter 16, $C \equiv C'$. ■

Exercises

1. Let A be a given set.
 - (a) Show that \bar{A}' is productive.
 - (b) Show that if $K \leq_1 A$, then A' is productive.
2. Let $g(t)$ be a given partially computable function and let

$$A = \{x \in N \mid \Phi_x = g\}.$$

Prove that A is productive.

3. $A, B \subseteq N$ are called *effectively inseparable* if $A \cap B = \emptyset$, and there is a partially computable function $g(x, y)$ such that

$A \subseteq W_x \& B \subseteq W_y \& W_x \cap W_y = \emptyset \quad \text{implies} \quad g(x, y) \downarrow \& g(x, y) \notin W_x \cup W_y.$

- (a) Show that if the above holds, it also does so with g total.
- (b) Prove that there is an effectively inseparable pair of r.e. sets.
- (c) Show that if A, B are effectively inseparable and r.e., then they are each creative. [Hint for (b): Try $A = \{x \in N | \Phi_x(x) = 0\}$ and $B = \{x \in N | \Phi_x(x) = 1\}$.]

4. Show that TOT and $\overline{\text{TOT}}$ are both productive.

5. Let A be productive and let B be r.e. Prove that

- (a) if $B \subseteq A$, then $A - B$ is productive;
- (b) if $A \subseteq B$, then $A \cup \bar{B}$ is productive.

6. Show that if A is creative, so is $\{\langle x, y \rangle | x, y \in A\}$.

7. If A is productive, show that $\{x \in N | W_x \subseteq A\}$ is productive.

4. Simple Sets—Dekker's Theorem

It is now natural to ask: Do there exist r.e. sets which are neither computable nor creative? Since by Theorem 3.6, the complement of a creative set has an infinite r.e. subset, we are led to seek sets which satisfy the following definition, which was first given by Emil Post in 1944:

Definition. A set S is called *simple* if

1. S is r.e.;
2. \bar{S} is infinite;
3. there is no infinite r.e. set U such that $U \subseteq \bar{S}$.

Of course, we do not yet know that there are any simple sets. But it follows at once from the definition that such sets cannot be recursive.

Theorem 4.1. If S is simple, then S is not recursive.

Proof. Let S be a simple set. If S were recursive, then \bar{S} would be an infinite r.e. subset of \bar{S} , contradicting the definition of simple set. ■

Of course, Theorem 3.6 implies

Theorem 4.2. A creative set is not simple.

We next show (using Post's original construction) that there are, in fact, simple sets. For each number i , we let

$$S_i = \{n \in W_i \mid n > 2i\}.$$

We shall define a partially computable function $\psi(i)$, related to these sets S_i as follows:

- (a) $S_i \neq \emptyset$ implies $\psi(i) \in S_i$, and
- (b) $S_i = \emptyset$ implies $\psi(i) \uparrow$.

For this purpose we note that the predicate

$$n \in W_i \& n > 2i$$

is Σ_1 , and hence we can write

$$n \in S_i \Leftrightarrow (\exists z)R(i, n, z),$$

where $R(i, n, z)$ is a recursive predicate. Then we simply let ψ be the partially computable function defined by the equation

$$\psi(i) = l(\min_t R(i, l(t), r(t))).$$

It is readily seen that ψ satisfies (a) and (b) above.

It is the *range* of this function ψ which we shall prove is simple.

Theorem 4.3. Let

$$S = \{\psi(i) \mid \psi(i) \downarrow\}.$$

Then S is simple.

Lemma 1. S is r.e.

Proof. This is immediate from Theorem 7.2 in Chapter 7. However, since we are not presuming that readers of this chapter have completed Chapter 7, we merely observe that S is the set of numbers n which satisfy the Σ_1 predicate

$$(\exists i)(\exists t)(n = l(t) \& R(i, l(t), r(t)) \& (\forall z)_{<t} \sim R(i, l(z), r(z))).$$

■

Lemma 2. There is no infinite r.e. set U such that $U \subseteq \bar{S}$.

Proof. Suppose that W_i is infinite and $W_i \subseteq \bar{S}$. Then, $W_i \cap S = \emptyset$. Since W_i is infinite, for some $n \in W_i$, we must have $n > 2i$, that is, $n \in S_i$. Therefore, $S_i \neq \emptyset$. By (a), $\psi(i) \in S_i$, and since $\psi(i) \in W_i$, $\psi(i) \in W_i \cap S$. This is a contradiction. ■

Lemma 3. \bar{S} is infinite.

Proof. We begin by asking: How many elements of S can there be which are $\leq 2n + 2$? That is, how many numbers $\psi(i)$ are there for which $\psi(i) \leq 2n + 2$? Now, if $i \geq n + 1$ and $\psi(i) \downarrow$ then $\psi(i) > 2i \geq 2n + 2$. Thus, we need only consider numbers $\psi(i)$ for which $i \leq n$. Thus the elements of S which are $\leq 2n + 2$ are at most those of the $n + 1$ values $\psi(0), \psi(1), \dots, \psi(n)$ which are defined. Since at most $n + 1$ of the first $2n + 2$ integers belong to S , we see that at least $n + 1$ of them must belong to \bar{S} . Hence \bar{S} is infinite. ■

Theorem 4.3 is an immediate consequence of Lemmas 1–3. ■

We next note that simple sets automatically settle the analog of Post's problem for m-degrees (or 1-degrees).

Theorem 4.4. No simple set can be m-complete for Σ_1 .

Proof. If A is m-complete for Σ_1 , then by Theorem 3.7, A is creative and hence, by Theorem 4.2, A is not simple. ■

However, using a construction different from Post's, Dekker was able to show that every r.e. degree other than 0 contains simple sets, so that the mere existence of simple sets cannot settle Post's problem. In proving Dekker's result, we will need an easy basic theorem about infinite r.e. sets:

Theorem 4.5. If A is an infinite r.e. set, there is a recursive unary one-one function F whose range is A .

Proof. For any non-empty r.e. set A , it is easy to obtain a recursive function whose range is A . If A is the set of numbers which satisfy the Σ_1 predicate $(\exists y)R(x, y)$, where R is recursive, and x_0 is some fixed element of A , then A is the range of the recursive function $g(x, y)$ defined by

$$g(x, y) = \begin{cases} x & \text{if } R(x, y) \\ x_0 & \text{otherwise.} \end{cases}$$

Furthermore, A is then also the range of the unary function

$$f(t) = g(l(t), r(t)).$$

(See also Theorem 7.1 in Chapter 7.)

We need to modify f so that no values are repeated. We define

$$F(0) = f(0),$$

$$F(t + 1) = f(\min_z (\forall i)_{\leq t} [F(i) \neq f(z)]).$$

Since A is infinite, F is total and the range of F is A . F is clearly one-one. To see that F is recursive, we use Gödel numbers, writing

$$G(x) = [F(0), \dots, F(x)],$$

so that $F(x) = (G(x))_{x+1}$. We also let

$$H(t, w) = f(\min_z (\forall i)_{\leq t} [(w)_i \neq f(z)]).$$

Finally, we have

$$G(0) = 2^{f(0)}$$

$$G(t + 1) = G(t) \cdot p_{t+2}^{H(t, G(t))}.$$

The last pair of equations shows that G , and therefore F , is recursive. (We have used Theorems 3.1 and 7.2 in Chapter 3. See also Exercise 8.3 in Chapter 3.) ■

Theorem 4.6 (Dekker's Theorem). For every r.e. degree $\mathbf{a} \neq \mathbf{0}$, there is a simple set B such that $\deg(B) = \mathbf{a}$.

Proof. Let A be an r.e. set with $\deg(A) = \mathbf{a}$. Since $\mathbf{a} \neq \mathbf{0}$, A is not recursive, and so A is certainly infinite. By Theorem 4.5, A is the range of a one-one recursive function $F(x)$. We let

$$B = \{n \in N \mid (\exists m)(m > n \ \& \ F(m) < F(n))\}.$$

We shall show that B is simple and has degree \mathbf{a} .

Lemma 1. B is r.e.

Proof. The defining predicate of B is Σ_1 . ■

Lemma 2. $B \leq_t A$.

Proof. We have

$$n \in B \Leftrightarrow (\exists k)_{< F(n)} [k \in A \ \& \ (\forall i)_{< n} (k \neq F(i))].$$

Hence (using Theorem 1.4 in Chapter 16), $B \leq_t A$. ■

Lemma 3. \bar{B} is infinite.

Proof. Let any $k \in N$ be given. Let $n > k$ be such that $F(n)$ has the least possible value. Then clearly $n \notin B$, i.e., $n \in \bar{B}$. ■

Lemma 4. Let $n \in \bar{B}$. Then, for all $x \in A$, either $x > F(n)$ or $x = F(i)$ for some $i = 0, 1, \dots, n$.

Proof. Since $n \in \bar{B}$, it must be the case that for all $m > n$, $F(m) \geq F(n)$. If $x \in A$, but $x \neq F(i)$, $0 \leq i \leq n$, then $x = F(m)$ for some $m > n$; hence $F(m) \geq F(n)$. Since F is one-one, $x = F(m) > F(n)$. ■

Lemma 5. $A \leq_t B$.

Proof. Let

$$n_0(x) = \min_n [x < F(n) \& n \in \bar{B}].$$

Since only finitely many $F(n)$'s can be $\leq x$ for any given x and \bar{B} is infinite, it follows that n_0 is a total function. The definition of n_0 implies that it is B -computable. Now, by Lemma 4,

$$x \in A \Leftrightarrow (\exists i)_{\leq n_0(x)}(x = F(i)).$$

Hence $A \leq_t B$. ■

Lemma 6. There is no infinite r.e. set $D \subseteq \bar{B}$.

Proof. Suppose that there were such a set D , and let it be the range of the recursive function g . Let

$$m_0(x) = \min_t [x < F(g(t))].$$

Since D is infinite and F is one-one, m_0 is total, and hence recursive. Thus, $g(m_0(x)) \in D \subseteq \bar{B}$ and $x < F(g(m_0(x)))$. Taking $n = g(m_0(x))$ in Lemma 4, we see that

$$x \in A \Leftrightarrow (\exists i)_{\leq g(m_0(x))}(x = F(i)).$$

But this would imply that A is recursive. ■

Now, by Lemmas 2 and 5, $\deg(B) = \deg(A) = \mathbf{a}$, and by Lemmas 1, 3, and 6, B is simple. ■

Exercises

1. Let A and B be simple sets:

- (a) Show that $A \oplus B$ is simple.
- (b) Show that $\{\langle x, y \rangle \mid x \in A, y \in B\}$ is not simple.

2. Let S be simple and let

$$R = \{\langle x, y \rangle \mid x \in S\}.$$

Show that R is neither creative, simple, nor recursive.

3. Let S be an r.e. set such that \bar{S} is infinite. Let the sequence

$$x_1 < x_2 < x_3 < \dots$$

enumerate all the members of \bar{S} in increasing order. Then S is called *hypersimple* if there is no recursive function $f(x)$ such that for all $n \in N$, $f(n) \geq x_n$.

- (a) Prove that a hypersimple set is simple.
- (b) Show that the set B constructed in the proof of Dekker's theorem is hypersimple.
- (c) Conclude that every r.e. degree $\neq \mathbf{0}$ contains a hypersimple set.

4. A set S is called *maximal* if S is r.e., \bar{S} is infinite, and for every r.e. set U such that $U \supseteq S$, either $U - S$ or \bar{U} is finite. Prove that a maximal set is hypersimple. (Note: All known proofs that maximal sets exist use priority constructions. See Section 6.)

5. Sacks's Splitting Theorem

In this section, we will obtain a number of key results about r.e. degrees, including the solution of Post's problem, from a single powerful result, Sack's splitting theorem. The proof of the theorem requires the intricate "priority" method, and we will leave the proof for the next section.

Theorem 5.1 (Sacks's Splitting Theorem). Let C be any r.e. set and let S be a simple set. Then there are r.e. sets A, B such that

- (1) $A \cup B = C, A \cap B = \emptyset,$
- (2) $S \not\leq_t A,$
- (3) $S \not\leq_t B.$

The applications of Sacks's splitting theorem depend on the following easy result.

Theorem 5.2. Let A, B be r.e. sets such that $A \cap B = \emptyset$. Then

$$\deg(A \cup B) = \deg(A) \vee \deg(B).$$

Proof. We need to show that

- (1) $A \cup B \leq_t A \oplus B;$
- (2) $A \leq_t A \cup B, B \leq_t A \cup B.$

Since

$$\begin{aligned} x \in A \cup B &\Leftrightarrow x \in A \vee x \in B \\ &\Leftrightarrow 2x \in A \oplus B \vee 2x + 1 \in A \oplus B, \end{aligned}$$

(1) is clear.

To prove (2) we note that, since $A \cap B = \emptyset$,

$$\bar{A} = \overline{A \cup B} \cup B.$$

Thus, \bar{A} is the union of two sets, one of which is $(A \cup B)$ -recursive and the other of which is r.e. Thus, each of these sets is $(A \cup B)$ -r.e., and hence so is their union, \bar{A} . Thus, A and \bar{A} are both $(A \cup B)$ -r.e., and hence $A \leq_t A \cup B$. A similar argument shows that $B \leq_t A \cup B$. ■

Theorem 5.3 (Degree Form of Sacks's Splitting Theorem). Let c, d be r.e. degrees, and let $d \neq 0$. Then there are r.e. degrees a, b such that

1. $c = a \vee b$,
2. $d \not\leq a$,
3. $d \not\leq b$.

Proof. Using Dekker's theorem (Theorem 4.6), there is a simple set S such that $d = \deg(S)$. Let $c = \deg(C)$ for some r.e. set C . By Sacks's splitting theorem, there are r.e. sets A, B such that (1)–(3) of the statement of that theorem hold. Let $a = \deg(A)$, $b = \deg(B)$. By Theorem 5.2,

$$c = \deg(C) = \deg(A \cup B) = a \vee b.$$

Finally, since $S \not\leq_t A$ and $S \not\leq_t B$, we have $d \not\leq a$ and $d \not\leq b$. ■

Theorem 5.4. Let c be an r.e. degree such that $c \neq 0$. Then there are r.e. degrees a, b such that

1. $c = a \vee b$,
2. $0 < a < c$, $0 < b < c$, and
3. $a \mid b$.

Proof. In Theorem 5.3, set $d = c$. Thus, $c = a \vee b$, $c \not\leq a$, $c \not\leq b$. But, using Theorem 1.2(6),

$$a \leq (a \vee b) = c,$$

$$b \leq (a \vee b) = c.$$

Since $c \not\leq a$ and $c \not\leq b$, we have $a < c$ and $b < c$. If we had $a = 0$, then using Theorem 1.2(8), we would have $c = (0 \vee b) = b$. Likewise, $b = 0$ would imply $c = a$. It remains only to show that $a \mid b$. But, if $a \leq b$, then by Theorem 1.2(7),

$$c = (a \vee b) \leq b.$$

Likewise, if $b \leq a$, then

$$c = (a \vee b) \leq a.$$

Finally, we present a solution to Post's problem by obtaining the following strengthening of the Kleene–Post theorem (Theorem 2.1):

Theorem 5.5 (Friedberg–Muchnik). There are r.e. degrees \mathbf{a}, \mathbf{b} such that

1. $\mathbf{0} < \mathbf{a} < \mathbf{0}'$;
2. $\mathbf{0} < \mathbf{b} < \mathbf{0}'$;
3. $\mathbf{a} \mid \mathbf{b}$.

Proof. In Theorem 5.4, take $\mathbf{c} = \mathbf{0}'$. ■

Definition. A degree \mathbf{c} is called *minimal* if $\mathbf{c} \neq \mathbf{0}$ and there is no degree \mathbf{b} with $\mathbf{0} < \mathbf{b} < \mathbf{c}$.

Corollary 5.6 (Muchnik). There are no minimal r.e. degrees.

Proof. Immediate from Theorem 5.4. ■

Although it is not at all easy to prove, minimal degrees (necessarily not r.e.) do exist. In fact there are even minimal degrees $\leq \mathbf{0}'$.

Here is one final consequence of Sacks's splitting theorem:

Theorem 5.7. If \mathbf{d} is any r.e. degree such that $\mathbf{0} < \mathbf{d} < \mathbf{0}'$, then there is an r.e. degree \mathbf{a} such that $\mathbf{a} \mid \mathbf{d}$.

Proof. In Theorem 5.3, set $\mathbf{c} = \mathbf{0}'$. We claim that either $\mathbf{a} \mid \mathbf{d}$ or $\mathbf{b} \mid \mathbf{d}$. For otherwise, $\mathbf{a} \leq \mathbf{d}$ and $\mathbf{b} \leq \mathbf{d}$. Hence, $\mathbf{0}' = \mathbf{c} = (\mathbf{a} \vee \mathbf{b}) \leq \mathbf{d}$, a contradiction. ■

6. The Priority Method

The Kleene–Post Theorem (Theorem 2.1) already shows that there are degrees \mathbf{a} such that

$$\mathbf{0} < \mathbf{a} < \mathbf{0}'.$$

But, as has already been emphasized, this does not yet constitute a solution to Post's problem because we have no reason to believe that the degree \mathbf{a} obtained in our proof of the Kleene–Post theorem is an r.e. degree. If we analyze the proof of Theorem 2.1, we see that the construction in stages involves repeated queries to a $\mathbf{0}'$ oracle. In order to get an r.e. degree, it is necessary to see how to do without such queries.

In this section, we use the same symbol for a set and its characteristic function. Thus $x \in A$ and $A(x) = 1$ will be regarded as equivalent statements;

similarly for $x \notin A$ and $A(x) = 0$. When we deal with A -computations, an oracle query m will be called *affirmative* if $m \in A$ and *negative* if $m \notin A$. (Recall Chapter 16, Section 1.)

Now if, for example, we wish to obtain a pair of sets A, B such that $A \not\leq_t B$, we need to satisfy the infinite set of *conditions*

$$(\exists x)(A(x) \neq \Phi_B(x, j)), \quad j = 0, 1, 2, \dots$$

In the construction used in proving the Kleene–Post theorem, the use of a \emptyset' oracle made it possible to be certain at a given stage that a condition, like one of those just mentioned, will ultimately be satisfied. To give up the use of such an oracle is to be condemned to the lack of certain knowledge that a condition has truly been satisfied at a given stage of a construction. We will proceed in such a manner that if all goes well, the condition will be satisfied. But we must reckon with the possibility that requirements of the construction at a later stage will undermine what has been accomplished. The way out involves two ideas:

1. each condition is to be dealt with infinitely often, so that if an early attempt to satisfy a condition is “injured,” this can be rectified at a later stage;
2. a “priority” scheme is to be set up to deal with “conflicts” between attempts to satisfy various conditions.

In the *finite injury priority method*, invented, independently, by Friedberg and Muchnik, the arrangements for dealing with each condition are injured only finitely many times; thus, ultimately the conditions prevail.

This somewhat vague account should assume substance as we proceed with the proof of Sacks’s splitting theorem.

Proof of Theorem 5.1. We write

$$W_i^s = \{m \leq s \mid \text{STP}^{(1)}(m, i, s)\}.$$

(This notation is not quite the same as that used in Chapter 16, Section 7; the relation between the two is given by $W_i^s = {}_s W_i^{s+1}$.)

Let C be a given r.e. set and let S be a given simple set. We need to show how to split C into disjoint r.e. sets A, B such that $S \not\leq_t A, S \not\leq_t B$. We choose numbers c, d such that

$$C = W_c, \quad S = W_d$$

and write

$$C^s = W_c^s, \quad S^s = W_d^s$$

for $s = 0, 1, 2, \dots$. For each $s \in N$, we shall show how to construct finite sets A^s, B^s such that

- (1) The predicates $x \in A^s, x \in B^s$ are computable.
- (2) $A^s \subseteq A^{s+1}, B^s \subseteq B^{s+1}$.

The desired sets A, B are then defined by

$$A = \bigcup_{s \in N} A^s, \quad B = \bigcup_{s \in N} B^s.$$

Since

$$A = \{x \mid (\exists s)(x \in A^s)\},$$

and

$$B = \{x \mid (\exists s)(x \in B^s)\},$$

we see by (1) that A and B will indeed be r.e. sets.

Our construction will begin by setting $A^0 = B^0 = \emptyset$ and will proceed by showing how to obtain A^{s+1}, B^{s+1} from A^s, B^s . The successive values of s are called *stages*. At each stage we maintain two finite sets, called the *A-list* and the *B-list*, of objects we call *requirements*. Each requirement is a triple (ρ, k, U) , where $\rho, k \in N$ and U is a finite subset of N . Such a requirement is called a ρ -requirement for input k . When (ρ, k, U) is on the *A-list* ρ will be *even*, and when it is on the *B-list* ρ will be *odd*. As we shall see, when a requirement (ρ, k, U) is on the *A-list*, there has been reason to desire that $U \subseteq \bar{A}$ (because the members of U were all negative oracle queries of some A^s -computation). Similarly for requirements on the *B-list*. We say that a number j belongs to a requirement (ρ, k, U) if $j \in U$. At stage $s = 0$ the *A-list* and *B-list* are both empty.

We now assume that the construction has been carried out through stage s , so that A^s and B^s are defined along with the current state of the *A-list* and the *B-list*. We must show how to obtain A^{s+1} and B^{s+1} , and how to update the *A-list* and the *B-list*. We let $j = l(s)$ and express our conditions in terms of j rather than in terms of s directly. Since the equation $j = l(s)$ is satisfied, for given j , by infinitely many values of s , this will ensure that each condition is treated infinitely many times.

We say that s is *C-ripe* if $j = l(s) \in C^s$ and $j \notin A^s \cup B^s$. If s is not *C-ripe*, we put $A^{s+1} = A^s, B^{s+1} = B^s$, and we let the *A-list* and the *B-list* remain unchanged. We now assume that s is *C-ripe*. Then, we shall define A^{s+1}, B^{s+1} so that either

$$A^{s+1} = A^s \cup \{j\}, \quad B^{s+1} = B^s$$

or

$$A^{s+1} = A^s, \quad B^{s+1} = B^s \cup \{j\},$$

where, as above, $j = l(s)$. In the first case, we say j goes in A , and in the second case, j goes in B . Now, if j belongs to no requirement on the *A-list*,

then j goes in A , and if j does belong to a requirement on the A -list but to no requirement on the B -list, then j goes in B . If j belongs to requirements both on the A -list and on the B -list, we insist that j go in either A or B (but not both), using a system of *priorities* to resolve the “conflicts” that arise. In such a case, let ρ be minimal such that j belongs to a ρ -requirement on the A -list or on the B -list. (We are giving such requirements with minimal ρ priority over the others.) If a ρ -requirement occurs on the A -list, then j goes in B , and all requirements on the B -list to which j belongs are deleted. Otherwise, j goes in A and all requirements on the A -list to which j belongs are deleted.

This completes the construction of A^{s+1} , B^{s+1} . However, the A -list and the B -list may require further processing.

We say that s is *A-ripe* if [with $j = l(s)$ as above] there is a $k \leq s$ such that

- (1) $\text{STP}_{A^s}^{(1)}(k, j, s)$,
- (2) $\Phi_{A^s}(k, j) = 0$,

(3) For every $2j$ -requirement for input h on the A -list at stage s , we have $h \neq k$, and either $h \notin S^s$ or $h > k$.

If s is not *A-ripe*, no further changes are made to the A -list. If s is *A-ripe*, we choose the smallest number $k \leq s$ which satisfies (1), (2), and (3), and take the opportunity to attempt to arrange matters so that $\Phi_A(k, j) = 0$. Let \mathcal{P} be the program such that $\#(\mathcal{P}) = j$, and let U be the set of negative oracle queries of the A^s -computation by \mathcal{P} with input k [of length $\leq s$ leading to the output 0; cf. (1), (2)]. Then, the requirement $(2j, k, U)$ is adjoined to the A -list.

B-ripe is defined exactly like *A-ripe* and the B -list is treated just like the A -list, except that we consider $(2j + 1)$ -requirements instead of $2j$ -requirements.

This completes the construction. It should be clear that the predicates $x \in A^s$, $x \in B^s$ are indeed computable. We have

Lemma 1. $A \cap B = \emptyset$, $A \cup B = C$.

Proof. The construction is such that $A^s \cap B^s = \emptyset$; $A^s \cup B^s \subseteq C^s$. Hence $A \cap B = \emptyset$, $A \cup B \subseteq C$. It remains to show that $C \subseteq A \cup B$.

Let $j \in C$. Let $s = \langle j, m \rangle$, where m is so large that $j \in C^s$. Then $j = l(s)$. If $j \in A^s \cup B^s$, then certainly $j \in A \cup B$. Otherwise s is *C-ripe* and $j \in A^{s+1}$ or $j \in B^{s+1}$. Hence $j \in A \cup B$. ■

A requirement which is on the A -list or the B -list at some stage s , but not at all stages $t > s$, is called *temporary*; if it is on the A -list or the B -list at all stages $t > s$ for some s , it is called *permanent*.

Lemma 2 (Finite Injury Lemma). For each ρ , there are only finitely many ρ -requirements which ever appear on the A -list or on the B -list.

Proof. We use mathematical induction to obtain the result. We suppose it known for each $\bar{\rho} < \rho$ that there are only finitely many $\bar{\rho}$ -requirements that ever occur on the A -list or on the B -list. We begin by considering *temporary* ρ -requirements. For each temporary ρ -requirement there is a number j which belongs to the requirement and such that j went in A or B when the requirement was deleted. But then j must also belong to a $\bar{\rho}$ -requirement with $\bar{\rho} < \rho$ which was on the B -list or the A -list at that stage. Using the induction hypothesis, we see that there can only be a finite number of such j 's. Since all the ρ -requirements which are deleted as the result of a particular number j going in A or B are deleted at the same stage, there can only be finitely many of them for each such j . Thus, there are only finitely many temporary ρ -requirements that are ever on the A -list or the B -list.

Next, we consider permanent ρ -requirements. Let E_ρ^s be the set of numbers k such that there is a ρ -requirement for input k on the A -list or the B -list at stage s . And, let E_ρ be the set of k such that there is a permanent ρ -requirement for input k . Then, clearly, the predicate $k \in E_\rho^s$ is computable. Since we now know that there are only finitely many temporary ρ -requirements, there must be some stage s_0 such that at stages $s \geq s_0$ all ρ -requirements still on the A -list or the B -list are permanent. Hence

$$E_\rho = \{k \in N \mid (\exists s)(s \geq s_0 \ \& \ k \in E_\rho^s)\},$$

and we can conclude that E_ρ is r.e. We will show that E_ρ is finite. Since at any stage s , for each k there is at most one ρ -requirement for input k (because a new one is not added to the A -list or the B -list if one is already present), we will be able to conclude that there are only finitely many permanent ρ -requirements, thus completing the proof of the lemma.

Suppose that E_ρ is infinite. Since S is simple and E_ρ is r.e., we know that $E_\rho \not\subseteq \bar{S}$. Hence, there is an $h \in S \cap E_\rho$. Suppose that ρ is even, and let us write $\rho = 2j$. Let m be so large that, for $s = \langle j, m \rangle$, we have $h \in S^s \cap E_\rho^s$. Thus, $j = l(s)$, and there is a $2j$ -requirement for input h on the A -list at all stages $t \geq s$. Hence, by (3) of the definition of "A-ripe," all $2j$ -requirements that are placed on the A -list at a stage $t \geq s$ are for inputs $k < h$. This implies that E_ρ is finite. If ρ is odd, we argue similarly. ■

Lemma 3. For each j , there is an $x \in N$ such that $S(x) \neq \Phi_A(x, j)$.

Proof. Suppose that for all $x \in N$,

$$S(x) = \Phi_A(x, j).$$

We begin by showing that $E_{2j} \subseteq \bar{S}$. Thus, let $k \in E_{2j}$. If $k \in E_{2j}^{s+1} - E_{2j}^s$, then by (2) of the definition of “ A -ripe,” we have

$$\Phi_{A^s}(k, j) = 0. \quad (6.1)$$

Since the requirement for input k created at stage s is permanent, none of the negative oracle queries in the A^s -computation entailed by (6.1) will be in A . Hence,

$$S(k) = \Phi_A(k, j) = 0,$$

i.e., $k \notin S$. We have proved that $E_{2j} \subseteq \bar{S}$.

Now since \bar{S} is infinite and E_{2j} is finite, there is a number k which belongs to $\bar{S} - E_{2j}$. Thus, for this number k ,

$$\Phi_A(k, j) = S(k) = 0.$$

Using Lemma 2, let s be selected with $l(s) = j$, sufficiently large that all of the following hold:

1. $k \leq s$.
2. $\text{STP}_{A^s}^{(1)}(k, j, s)$.
3. $\Phi_{A^s}(k, j) = 0$.
4. The A -list at stage s contains no temporary $2j$ -requirements and all of the permanent $2j$ -requirements.

We claim that

Stage s is A -ripe.

To see this, consider a $2j$ -requirement for input h which is on the A -list at stage s . Since this $2j$ -requirement must be permanent, $h \in E_{2j}$. Since $k \notin E_{2j}$, we have $h \neq k$. Since $h \in E_{2j}$, we must have also $h \in \bar{S}$, and hence certainly $h \notin S^s$. This proves our claim.

Now, since s is A -ripe, a new $2j$ -requirement (for input k) will be placed on the A -list. This is a contradiction. ■

Lemma 4. For each j , there is an $x \in N$ such that $S(x) \neq \Phi_B(x, j)$.

Proof. The proof is similar to that of the preceding lemma. ■

Lemma 5. $S \not\leq_t A, S \not\leq_t B$.

Proof. Immediate from Lemmas 3 and 4. ■

Sacks's splitting theorem now follows at once from Lemmas 1 and 5. ■

Suggestions for Further Reading

C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

A very readable treatise on resolution-based algorithms for satisfiability in quantification theory.

Martin Davis, *Computability and Unsolvability*. Dover, New York, 1983.

Originally published in 1958. The 1983 reprint includes an appendix on unsolvable problems in number theory.

Martin Davis (editor), *The Undecidable*. Raven, New York, 1965.

A collection of basic papers in computability theory. Included are the original papers in which Church announced his "thesis," in which Turing defined his machines and produced a universal computer, in which Post stated his "problem," and in which Turing introduced "oracles."

Herbert P. Enderton, *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

An introductory textbook on mathematical logic for mathematically mature readers.

Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

This treatise includes a comprehensive list of NP-complete problems.

Michael Harrison, *Introduction to Formal Language Theory*. Addison-Wesley, Reading, Massachusetts, 1978.

A comprehensive up-to-date readable treatise on formal languages.

Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

Another introduction to theoretical computer science.

Donald W. Loveland, *Automated Theorem Proving: A Logical Basis*. North-Holland Publ., Amsterdam, 1978.

A well-organized account of resolution theory.

Michael Machtey and Paul Young, *An Introduction to the General Theory of Algorithms*. North-Holland Publ., Amsterdam, 1978.

A well-written account of computability and complexity theory.

Hartley Rogers, *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

The classic comprehensive treatise on computability and noncomputability.

Joseph R. Shoenfield, *Degrees of Unsolvability*. North-Holland Publ., Amsterdam, 1971.

A short and clearly written monograph on the subject going well beyond the material covered in this book.

Robert I. Soare, *Recursively Enumerable Sets and Degrees: The Study of Computable Functions and Computably Generated Sets*. Springer-Verlag, Berlin and New York, to appear 1984.

A clear and complete account of priority constructions.

Index

A

- Ackermann, W., 305
Almost everywhere, 303, 314
 $\alpha(x)$, 38
Alphabets, 3, 4, 70
 of Turing machines, 98
Arithmetical hierarchy, 370–384
Arithmetical predicates, 378
Assignments, 232
Atoms, 231
Axiomatizable theories, 281–283, 287

B

- Bar-Hillel's pumping lemma, 187–190, 196
BASIC, 181
Blum axioms, 313–316
Blum, Manuel, 313
Blum speedup theorem, 328
Boolean algebra, 234
Bounding Theorem, 301
 converse to, 307–310
Bracket languages, 199–204

C

- Cartesian product, 2
Case, John, 68
Chang, C. L., 417
Chomsky hierarchy, 218–220
Chomsky normal form grammars, 185–188,
 200
 separators of, 200, 201, 209
Chomsky–Schützenberger representation
 theorem, 203, 204
- Church, Alonzo, 283
Church's thesis, 54, 55, 61, 67, 93, 110,
 111, 311, 336, 338, 353
Clauses, 238
 linked sets of, 247, 271, 275
 minimally unsatisfiable sets of, 247, 268
Closure properties
 for context-free languages, 190–194
 for context-sensitive languages, 220, 226,
 227
 for regular languages, 158–162
CNF, 238–241, 267, 268, 335, 337, 342–347
COBOL, 215
Compactness theorem
 for predicate logic, 278, 280
 for propositional calculus, 252, 266
Compilers, 215–217
Complexity measures, 313
Composition, 32, 33, 293, 356
Computable functions, *see* functions,
 computable
Configurations
 of linear bounded automata, 220
 of pushdown automata, 207
 of Turing machines, 99, 336, 342–346
Conjunctive normal form, *see* CNF
Constant symbols, 253
Context-free grammars, 171–181, 218
 ambiguity of, 198, 200
 branching, 179, 181
 Chomsky normal form, *see* Chomsky
 normal form grammars
 derivations in, 172
 derivation trees in, *see* Derivation trees

- infinite, 196, 197
- kernels of, 172
- leftmost derivations in, 177–179
- positive, 171, 173, 174, 218
- regular, 181–183, 218
- right-linear, 184
- rightmost derivations in, 177–179
- self-embedding, 185
- unsolvable problems involving, 198
- Context-free languages, 172, 174, 190, 212, 213, 216–219, 228
 - closure properties for, 190–194
- Context-sensitive grammars, 136, 137, 218–220
 - unsolvable problems involving, 228
- Context-sensitive languages, 218–228
 - closure properties for, 220, 226, 227
 - open questions concerning, 226, 228
- Cook–Karp thesis, 336
- Cook’s theorem, 341–346
- Course-of-values recursion, 49, 50
- Creative sets, 396, 398–400, 402–405

- D**
- Davis, Martin, 417
- Davis–Putnam rules, 242–246, 248, 249, 267, 268, 271, 335
- Definition by cases, 40
- Degree
 - of function symbols, 253
 - of relation symbols, 253
- Degrees, *see* Turing degrees
- Dekker, J. C. E., 405
- Dekker’s theorem, 406–407
- Derivation trees, 176–181
 - pruning and splicing of, 180, 181, 189
- De Morgan identities, 2, 6, 7, 160, 233, 237, 258, 259
- dfa, 153, *see also* Finite automata
- Disjunctive normal form (DNF), 238–241, 335
- Dovetailing, 61, 62
- Duality, general principle of, 234, 239
- Dyck languages, 203, 204

- E**
- Effective inseparability, 403
- Enderton, Herbert P., 417
- Enumeration principle, 250, 251, 278
- Enumeration theorem, 62

- F**
- Fermat’s last theorem, 55
- Fibonacci numbers, 49
- Finite automata, 149–152
 - nondeterministic, 153–158, 215
 - nonrestarting, 159, 212
- Finiteness theorem, 360
- Finite satisfiability, 250, 251
- Fixed point theorem, 67
 - parametrized, 401
- Floyd, R. W., 128
- Formulas
 - atomic, 254
 - in predicate logic, 254
 - propositional, 231
- FORTRAN, 181, 215, 217, 291
- Friedberg, Richard, 395, 410, 411
- Functions, 2, 3
 - binary, 3
 - computable, 27, 28, 35, 142
 - computed by a Post–Turing program, 87, 88
 - computed by a program, 17, 20–22, 25–28
 - computed strictly, 88, 98, 100
 - computed by a Turing machine, 98
 - domain of, 3
 - G-computable, 354, 355
 - G-partial recursive, 354
 - G-recursive, 354
 - n-ary, 3, 355
 - nowhere defined, 3, 28
 - partial, 3
 - partially computable, 27, 28, 141
 - partially G-computable, 354, 355
 - partial recursive, 27
 - polynomial time computable, 336
 - primitive recursive, 32–50, 60, 291–297
 - range of, 3, 139
 - recursive, 27
 - on strings, 73
 - total, 3, 27, 33, 34, 292, 354, 355
 - unary, 3
- Function symbols, 253

- G**
- Γ-trees, 175–179
 - paths in, 179

- Gap theorem, 317
Garey, Michael R., 417
Gödel, Kurt, 48
Gödel incompleteness theorem, 280–283
Gödel numbers, 48, 49, 52, 53, 141, 355, 356, 359
Grammars, 133–137, *see also* Context-free grammars; Context-sensitive grammars
 derivations in, 134, 172
 languages generated by, 133, 172, 218, 219
 null productions of, 171, 218, 228
 phrase structure, 133, 218
 positive context-free, 171, 218
 productions of, 133, 171
 regular, 181–183, 218
 right-linear, 184, 201
 self-embedding, 185
 start symbol of, 133, 171
 terminals of, 133, 171
 unsolvable problems concerning, 137, 138, 198, 228
 variables of, 133, 171
Graphs, 347–350
Greibach normal form grammars, 187
G-r.e. sets, 366–370
Ground clauses, 267
Ground resolution, 249, *see also* Resolution, in propositional calculus
Ground resolution theorem, 249
Growth limitation property, 300–301, 302
- H**
- Halting problem**, 53–55
 for Turing machines, 107, 108, 353
Harrison, Michael, 417
Herbrand instances of sentences, 285–286
Herbrand's theorem, 263, 266
Herbrand universe, 264, 267
Hilbert, David, 283
Hypersimple sets, 408
- I**
- Index sets**, 67
Induction, 9–12
 course-of-values, 10–12
 loading, 10
Initial functions, 34, 141, 293, 356
- Instructions**, 15, 16, 23, 291
 assignment, 291
 conditional branch, 16, 31, 311
 decrement, 16
 end, 291
 increment, 16, 291
 loop, 291
 oracle, 353
 numbers of, 51, 52, 356
 while, 311
 zero, 291
Interpretations of vocabularies, 255, 259
Intractability, 335
Iteration theorem, *see* Parameter theorem
- J**
- Join of degrees**, 390
Johnson, David S., 417
- K**
- Kleene–Post theorem**, 392–395, 410
Kleene's hierarchy theorem, 371
Kleene's theorem, 162–165
Kuroda, S. Y., 221
- L**
- Labels**, 16, 23, 51
Landweber, P. S., 221
Languages, 3
 accepted by a finite automaton, 151
 accepted by a linear bounded automaton, 221–226
 accepted by a nondeterministic finite automaton, 153
 accepted by a nondeterministic Turing machine, 110, 133, 135
 accepted by a pushdown automaton, 207–213
 accepted by a Turing machine, 104, 135
 context-free, *see* Context-free languages
 context-sensitive, *see* Context-sensitive languages
 generated by grammars, 133, 172, 218
 in NP, *see* NP
NP-complete, *see* NP-completeness
NP-hard, 340
polynomial-time decidable, *see* P
r.e., *see* r.e. languages
recursive, 106, 219

recursively enumerable, *see* r.e. languages
 regular, *see* Regular languages
 spanning sets for, 168
 Lee, R. C. T., 417
 Lewis, Harry R., 417
 Lexical analysis, 215, 216
 L'Hospital's rule, 334
 Linear bounded automata, 220–227
 deterministic, 228
 Literals, 237
 Logical consequence, 258, 259
 Loveland, Donald W., 418

M

Machtey, Michael, 319, 418
 Macro expansions, 17–22, 78, 79, 293
 Macros, 18, 28–31, 78, 79, 293
 for Post-Turing programs, 89, 90
 Majorizing, 303
 Many-one reducibility, 362–366
 Markov, A. A., 126
 Mates, 238, 247
 Mathematical induction, 9–12
 m-completeness, 365, 372, 391, 400, 402, 405
 Minimalization, 43–46, 140–142, 311, 356
 proper, 142
 Models, 257–259
 m-reducibility, *see* Many-one reducibility
 Muchnik, A. A., 395, 410, 411
 Myhill, John, 168, 386, 401, 402
 Myhill–Nerode theorem, 168, 169
 Myhill's theorem, 401, 402

N

Natural numbers, 1
 ndfa, 153, *see also* Finite automata, non-deterministic
 Nerode, Anil, 168
 Normal form theorem, 141, 311
 Normal processes, 142–144, 146
 Normal productions, 142
 NP, 338, 340, 341, 346–350
 NP-completeness, 340, 341, 346–350, 363
 n-tuple, 2

O

One-one reducibility, 362–366
 Oracles, 353, 355, 395, 411

P

P, 336–338, 340, 341
 Pairing functions, 47, 139–141, 359
 Palindromes, 153, 170
 Papadimitriou, Christos, H., 417
 Parameter theorem, 64–66, 320, 364, 379, 396
 relativized, 360–362
 Parsing, 216, 217
 Partially computable functions, *see* Functions, partially computable
 Partial recursive functions, *see* Functions, partial recursive
 Pascal, 181, 215
 Pigeon-hole principle, 166, 170
 Π_n , 370–384
 as a collection of sets, 370
 as a property of predicates, 372
 PL/I, 181, 215
 Polynomials, 333
 Polynomial-time reducibility, 340, 346, 363
 Post correspondence problem, 128–132,
 146, 197, 198, 214
 Post correspondence system, 128
 Post, Emil, 83, 126, 128, 396, 403–405
 Post's lemma, 125
 Post's problem, 392, 395, 396, 405, 408, 410
 Post's theorem, 374–378
 Post–Turing programs, 82–88, 356
 Post words, 120–126, 134
 PRC classes, 34, 35, 39–44, 355
 Predicates, 4–6
 arithmetic, 378
 computable, 30, 40, 54
 primitive recursive, 39, 40, 42, 43, 60
 Prenex normal form, 260
 Prenex sentences, 260
 Prime numbers, 43, 45, 46, 48, 49
 Primitive recursive functions (*see* functions, primitive recursive)
 Priority method, 395, 408, 411, 413
 finite injury, 411, 414
 Productive function, 396, 398, 399, 401
 Productive sets, 396, 398, 401, 403

- Programs, 15–25, 353
computations of, 25
empty, 23, 53
functions computed by, 17, 20–22, 25–
28, 292, 354
G-computations of, 354
instantaneous description of, 24
length of, 23
loop, 291–312
numbers of, 51–53
with oracles, 353
Post–Turing, *see* Post–Turing programs
snapshots of, 24, 25, 314, 353
states of, 23, 24, 353
for string computations, 77–81
universal, 58
- Projection functions, 34
- Proof by contradiction, 7–9
- Propositional connectives, 231
- Propositional variables, 231
- Pumping lemma, 166–169, *see also* Bar-Hillel's pumping lemma
- Pure literal rule, 243
- Pushdown automata, 204–214
atomic, 210, 211
computations by, 207
deterministic, 207, 210, 211
generalized, 214
languages accepted by, 207–213
transitions of, 206
- Q**
- Q*-degrees, 389, 390
- Quadruples, 97, 220–223, 339, 340
- Quantifiers, 6
alternating, 378
bounded, 42
in predicate logic, 253, 254
- Quintuples, 101
- R**
- Rates of growth, 331–334
- Recursion, 33, 34, 293, 294, 356
- Recursion theorem, 66–68, 322
parametrized, 400
- Recursive functions, *see* Functions, recursive
- Recursive isomorphisms, 386–388, 402
- Recursively enumerable languages, *see r.e. languages*
- Recursively enumerable sets, *see r.e. sets*
- Recursive permutations, 385–388
- Recursive relatedness theorem, 316
- r.e. degrees, 392, 395, 405, 406, 409, 410
- Reducibilities, 362–364
completeness with respect to, 363, 364
- Regular expressions, 164–166
- Regular grammars, 181–183
- Regular languages, 151, 154, 162, 165, 168, 170, 191–193, 201, 218
closure properties for, 158–162
infinite, 167
- r.e. languages, 104–106, 110, 135, 218, 267, 281
- Relation symbols, 253
- Relativization, 355
of computability, 354
of parameter theorem, 360–362
of recursive enumerability, 366–370
of step-counter theorem, 358, 359
of universality theorem, 356–358
- Remainder, 44, 70, 71
- Requirements, 412
permanent, 413–415
temporary, 413–415
- r.e. sets, 60–63, 136, 139, 140, 145, 364–
366, 385, 396, 398–400, 403–406
infinite, 405, 406
maximal, 408
- Resolution
derivations, 248
in predicate logic, 275–277
in propositional calculus, 247–250, 277
refutations, 248, 277
- Rice–Shapiro theorem, 385
- Rice's theorem, 67–69, 384–385
- Robinson's general resolution theorem, 277
- Rogers, Hartley, 418
- Rooted sentences, 285, 286
- Running time, 297–302
- S**
- Sacks's splitting theorem, 408–415
- SAT, 337–346
- Satisfiability
in predicate logic, 257, 266, 267, 283
problem, 241

- truth-functional, 232, 239, 247, 266, 267, 337–346
- Self-embedding grammars, 185
- Semi-Thue processes, 118–127, *see also* Thue processes
- Semi-Thue productions, 118
 inverses of, 123, 124, 225
- Sentences, 254
 prenex, 260
 rooted, 285, 286
 satisfiable, *see* Satisfiability, in predicate logic
- Skolemized, 260, 261
- universal, 261
- valid, 257, 267
- Sequence number theorem, 49
- Sets, 1
 characteristic functions of, 5
 complement of, 2
 difference of, 1
 empty, 1
 finite, 2
 infinite, 2
 intersection of, 1
 union of, 1
- Shoenfield, Joseph, 418
- Σ_n , 370–384
 as a collection of sets, 370
 as a property of predicates, 372
- Simple sets, 396, 403, 408, 411, 414
- Simulation, 82, 88–96, 112–117, 119–124, 224, 225
- Skolemization, 260–263
 generalized, 262, 263, 279
- Skolem–Löwenheim theorem, 280
- s-m-n theorem, *see* Parameter theorem
- Snapshot of a program, 24, 25
 successor of, 24
 terminal, 24
- Soare, Robert I., 418
- Socrates, 268
- Speedup theorem, 319–329
- Splitting rule, 243
- Statements, 23, *see also* Instructions
- States, 23, 24, 97, 150, 206
 accepting, 150, 206
 final, 150, 220, 221
 initial, 150, 206, 221
- Step-counter theorem, 59
 relativized, 358, 359
- Strings, 3, 4
 concatenation of, 74
 length of, 3
 numerical representation of, 70–74
- Subset, 1
 proper, 1
- Subsumption, 247
- T**
- Tautological inference, 235, 236
- Tautologies, 232, 239
- Terminals, 133, 171
- Terms, 253, 254
- Thue processes, 124–127, 283
- TOT, 145, 378–384, 403
- Transition functions, 150
- Turing, Alan, 83, 283
 analysis of computation process, 83, 97
- Turing degrees, 390–395
 minimal, 410
- Turing machines, 97–117, 336, *see also* Configurations, of Turing machines; Linear bounded automata
 functions computed by, 98
 languages accepted by, 104, 133–135
 multiple tape, 116, 117
 multiple track, 112–117, 226, 227
 nondeterministic, 98, 108–111, 119–124, 338, 341–346
 one-way infinite, 112
 universal, 103, 104
- Turing reducibility, 362, 363, 366
- U**
- Unconditional branch, 17
- Unification, 274, 275
- Unit clauses, 242, 243
- Unit rule, 243
- Universality theorem, 55
 relativized, 356–358
- Universal sentences, 261
- Unsolvability of
 halting problem, 54, 55, 107, 108
 Post's correspondence problem, 128–131
 problems involving grammars, 137–138, 198, 228
 satisfiability problem in predicate logic, 283
 word problems, 124–127, 145

V

Validity, *see* Sentences, valid
Variables, 15, 23, 51
 bound, 254, 255
 free, 254, 255
 in grammars, 133, 171
 input, 15, 23
 local, 15, 16, 23
 occurrences of, 254
 output, 15, 16, 23

 in predicate logic, 253
 propositional, 231
Vocabularies, 253
 interpretations of, 255

W

Words, 3, 4, *see also* Strings

Y

Young, Paul, 319, 418

Computer Science and Applied Mathematics
A SERIES OF MONOGRAPHS AND TEXTBOOKS

Editor
Werner Rheinboldt
University of Pittsburgh

HANS P. KÜNZI, H. G. TZSCHACH, AND C. A. ZEHNDER. Numerical Methods of Mathematical Optimization: With ALGOL and FORTRAN Programs, Corrected and Augmented Edition

AZRIEL ROSENFELD. Picture Processing by Computer

JAMES ORTEGA AND WERNER RHEINBOLDT. Iterative Solution of Nonlinear Equations in Several Variables

AZARIA PAZ. Introduction to Probabilistic Automata

DAVID YOUNG. Iterative Solution of Large Linear Systems

ANN YASUHARA. Recursive Function Theory and Logic

JAMES M. ORTEGA. Numerical Analysis: A Second Course

G. W. STEWART. Introduction to Matrix Computations

CHIN-LIANG CHANG AND RICHARD CHAR-TUNG LEE. Symbolic Logic and Mechanical Theorem Proving

C. C. GOTLIEB AND A. BORODIN. Social Issues in Computing

ERWIN ENGELER. Introduction to the Theory of Computation

F. W. J. OLVER. Asymptotics and Special Functions

DIONYSIOS C. TSICHRITZIS AND PHILIP A. BERNSTEIN. Operating Systems

PHILIP J. DAVIS AND PHILIP RABINOWITZ. Methods of Numerical Integration

A. T. BERZTISS. Data Structures: Theory and Practice, Second Edition

N. CHRISTOPHIDES. Graph Theory: An Algorithmic Approach

SAKTI P. GHOSH. Data Base Organization for Data Management

DIONYSIOS C. TSICHRITZIS AND FREDERICK H. LOCHOVSKY. Data Base Management Systems

JAMES L. PETERSON. Computer Organization and Assembly Language Programming

WILLIAM F. AMES. Numerical Methods for Partial Differential Equations, Second Edition

ARNOLD O. ALLEN. Probability, Statistics, and Queueing Theory: With Computer Science Applications

ELLIOTT I. ORGANICK, ALEXANDRA I. FORSYTHE, AND ROBERT P. PLUMMER. Programming Language Structures

ALBERT NIJENHUIS AND HERBERT S. WILF. Combinatorial Algorithms, Second Edition

AZRIEL ROSENFELD. Picture Languages, Formal Models for Picture Recognition

ISAAC FRIED. Numerical Solution of Differential Equations

ABRAHAM BERMAN AND ROBERT J. PLEMMONS. Nonnegative Matrices in the Mathematical Sciences

BERNARD KOLMAN AND ROBERT E. BECK. Elementary Linear Programming with Applications

- CLIVE L. DYM AND ELIZABETH S. IVEY.** Principles of Mathematical Modeling
- ERNEST L. HALL.** Computer Image Processing and Recognition
- ALLEN B. TUCKER, JR.** Text Processing: Algorithms, Languages, and Applications
- MARTIN CHARLES GOLUMBIC.** Algorithmic Graph Theory and Perfect Graphs
- GABOR T. HERMAN.** Image Reconstruction from Projections: The Fundamentals of Computerized Tomography
- WEBB MILLER AND CELIA WRATHALL.** Software for Roundoff Analysis of Matrix Algorithms
- ULRICH W. KULISCH AND WILLARD L. MIRANKER.** Computer Arithmetic in Theory and Practice
- LOUIS A. HEGEMAN AND DAVID M. YOUNG.** Applied Iterative Methods
- I. GOHBERG, P. LANCASTER AND L. RODMAN.** Matrix Polynomials.
- AZRIEL ROSENFELD AND AVINASH C. KAK.** Digital Picture Processing, Second Edition, Vol. 1, Vol. 2
- DIMITRI P. BERTSEKAS.** Constrained Optimization and Lagrange Multiplier Methods
- JAMES S. VANDERGRAFT.** Introduction to Numerical Computations, Second Edition
- FRANÇOISE CHATELIN.** Spectral Approximation of Linear Operators
- GÖTZ ALEFELD AND JÜRGEN HERZBERGER.** Introduction to Interval Computations. Translated by Jon Rokne
- ROBERT R. KORFHAGE.** Discrete Computational Structures, Second Edition
- MARTIN D. DAVIS AND ELAINE J. WEYUKER.** Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science

In preparation

- LEONARD UHR.** Algorithm-Structured Computer Arrays and Networks: Architectures and Processes for Images, Percepts, Models, Information
- O. AXELSSON AND V. A. BARKER.** Finite Element Solution of Boundary Value Problems: Theory and Computation
- PHILIP J. DAVIS AND PHILIP RABINOWITZ.** Methods of Numerical Integration, Second Edition
- ROBERT H. BONCZEK, CLYDE W. HOLSAPPLE, AND ANDREW B. WHINSTON.** Micro Data Base Management: Practical Techniques for Application Development.
- NORMAN BLEISTEIN.** Mathematical Methods for Wave Phenomena