Prentice-Hall

Series in Automatic Computation

George Forsythe, editor

BATES AND DOUGLAS, *Programming Language/One*
BAUMANN, FELICIANO, BAUER, AND SAMELSON, *Introduction to ALGOL*
BOWLES (editor), *Computers in Humanistic Research*
CESCHINO AND KUNTZMAN, *Numerical Solution of Initial Value Problems*
DESMONDE, *Computers and Their Uses*
DESMONDE, *Real-Time Data Processing Systems: Introductory Concepts*
EVANS, WALLACE, AND SUTHERLAND, *Simulation Using Digital Computers*
FORSYTHE AND MOLER, *Computer Solution of Linear Algebraic Systems*
GOLDEN, *Fortran IV: Programming and Computing*
GOLDEN AND LEICHUS, *IBM 360: Programming and Computing*
HARTMANIS AND STEARNS, *Algebraic Structure Theory of Sequential Machines*
HULL, *Introduction to Computing*
MARTIN, *Design of Real-time Computer Systems*
MARTIN, *Programming Real-Time Computer Systems*
MINSKY, *Computation: Finite and Infinite Machines*
MOORE, *Interval Analysis*
SCHULTZ, *Digital Processing: A System Orientation*
SNYDER, *Chebyshev Methods in Numerical Approximation*
STROUD AND SECREST, *Gaussian Quadrature Formulas*
TRAUB, *Iterative Methods for the Solution of Equations*
VARGA, *Matrix Iterative Analysis*
WILKINSON, *Rounding Errors in Algebraic Processes*
ZIEGLER, *Shared-Time Data Processing*

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LTD., *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*

COMPUTATION:

FINITE AND INFINITE MACHINES

MARVIN L. MINSKY

Professor of Electrical Engineering
Massachusetts Institute of Technology

PRENTICE-HALL, INC.
ENGLEWOOD CLIFFS, N. J.

Current printing (last digit):

10 9 8 7 6 5 4 3 2

Library of Congress Catalog Card No. 67-12342
Printed in the United States of America

In memory of HENRY MINSKY

PREFACE

INTRODUCTION

Man has within a single generation found himself sharing the world with a strange new species: the computers and computer-like machines. Neither history, nor philosophy, nor common sense will tell us how these machines will affect us, for they do not do "work" as did machines of the Industrial Revolution. Instead of dealing with materials or energy, we are told that they handle "control" and "information" and even "intellectual processes." There are very few individuals today who doubt that the computer and its relatives are developing rapidly in capability and complexity, and that these machines are destined to play important (though not as yet fully understood) roles in society's future. Though only some of us deal directly with computers, all of us are falling under the shadow of their ever-growing sphere of influence, and thus we all need to understand their capabilities and their limitations.

It would indeed be reassuring to have a book that categorically and systematically described what all these machines can do and what they cannot do, giving sound theoretical or practical grounds for each judgment. However, although some books have purported to do this, it cannot be done for the following reasons: a) Computer-like devices are utterly unlike anything which science has ever considered—we still lack the tools necessary to fully analyze, synthesize, or even think about them; and b) The methods discovered so far are effective in certain areas, but are developing much too rapidly to allow a useful interpretation and interpolation of results. The abstract theory—as described in this book—tells us in no uncertain terms that the machines' potential range is enormous, and that its theoretical limitations are of the subtlest and most elusive sort. There is no reason to suppose machines have any limitations not shared by man.

We nevertheless expect our findings to crystallize in the years ahead—and we are confident that the ideas and the formal methods that fill this book will remain in the mainstream of that theory which is yet to come.

We know this is so for several reasons. a) The central concept of *effective computability* appears necessary, even inevitable, once one has grasped it. In a convergence of ideas rare in science and mathematics, all sorts of independent attempts to form a "Theory of Machines" have been swept in this direction; and b) The sheer simplicity of the theory's foundation and the extraordinarily short path from this foundation to its logical and surprising conclusions give the theory a mathematical beauty that alone guarantees it a permanent place in computer theory.

THE MAIN GOAL

The main goal of this book is to introduce the student to the concept of *effective procedure*—a technical idea that has crystallized only fairly recently but already promises to be as important to the practical culture of modern scientific life as were the ideas of geometry, calculus, or atoms. It is a vital intellectual tool for working with or trying to build models of intricate, complicated systems—be they minds or engineering systems. Its most obvious application is to computation and computers, but I believe it is equally valuable for clear thinking about biological, psychological, mathematical, and (especially) philosophical questions. This claim must seen impossibly pretentious, and I had better explain it: The theory of effectiveness is useful not only to prove things about complex systems, but is also necessary to prove things about *proof* itself!

The overall strategy of the book is very simple. We will try to engage the concept of effective procedure from such a variety of contexts, formalisms, and viewpoints, that its character will become thoroughly understood—in the sense that one can recognize it and adapt it, almost without thinking, wherever and whenever it is appropriate. This concept has so many different aspects that one cannot hope to know them all equally well. It appears unexpectedly in aspects of practical computation, linguistic theories, and mathematical logic. We must know some of these aspects very thoroughly, and others at least casually, if we are to be able to sense its style of operation in new situations. Both the text and the problems are designed with this polyvalence in mind: we pursue a few lines much further than most, but all these approaches keep linking and intersecting to form a web of weirdly different but connecting threads. Most fortunately, the theory is almost entirely self-contained, so that it has no formal mathematical "prerequisites" beyond a reasonable high-school algebra course. A superficial glance through these pages might give a bewildering impression of the use of many complex and technical formalisms, but this is most definitely not the case. The book is not written *in* the formalisms, it is merely *about* them, and one is neither required nor expected to ever have seen them before!

The book has also a second, more playful theme. The many-headed concept which we call *effective computability* often appears in an especially potent form—the *Universal* machine or system. This character has an astonishing variety of secret hiding places which we keep trying to ferret out. It is easily found in the ponderous operation of a modern digital computer's order-code, or the gross, redundant structure of a FORTRAN or ALGOL-like programming language. (It is not important, at least in this book, that you know what these are.) But it can also lurk within the tiny structure of a "computer" with two memory registers or two instruction-tapes (Chapter 14), or in the seemingly childish bead-stringing primitives of a Post normal system (Chapter 13), or even in a machine that (almost) neither reads nor writes on its tapes. While there is not much practical use in finding these "minimal Universal systems" they support our main goal by making us aware of exactly what is essential rather than merely practical or convenient in the concept of computation.

Perhaps even more important, we learn of many different ways in which great complexity of behavior can arise from the interactions of simple devices, simple actions, simple descriptions, or simple concepts. We see also many different ways in which the *same* complex behavior can arise from what appear to be utterly different kinds of interactive systems.

The reader is prepared for both of these explorations in Part I, which is a training expedition into a smaller but essential (and in many ways analogous) territory. Within the *finite-state machine* area there exists as wide a variety of appearances of its own indigenous "central concept" (we explore fewer of these, though) and almost as surprising an assembly of its own minimal hiding places.

I wish I could have captured in the different approaches to the theory a sense of the different styles of thinking that lead the early explorers—Post, Church, Turing, Kleene, McCulloch, and the others—to approach the same problems with such different (technically and esthetically) formal methods. The personalities are obscured because in my attempt to modernize, simplify, and unify the results I have removed the individual details of the original developments. It would be well worth one's time perusing Davis' (1965) collection just to gain additional personal contact with the masters!

MATHEMATICAL TECHNIQUES

As we stated above, the book is largely self-contained, given enough high-school mathematics to know about functional notation (even this is reviewed in Chapter 7) and mathematical induction (reviewed in Chapter 4). MIT freshmen and sophomores have encountered no difficulty with the material, and though they may be more sophisticated mathe-

matically than most college students, I cannot imagine any good science student or computer programmer having serious trouble. I can, however, imagine educators (even mathematicians who have not encountered "recursive function theory") becoming suspicious and wondering whether I have somehow constructed a simplified or diluted version of the theory by "leaving out the calculus." Quite the contrary: the calculus was never in it! This really is a branch of mathematics that stems directly from non-numerical, logical foundations. Indeed, a number of research workers are currently attempting to bring together this theory with that of Newton and Leibniz; the resulting new subject (currently called "recursive analysis") has barely gotten off to a start.

It is worth a few words to try to explain why classical mathematical analysis plays such a small role here. In developing a Theory of Computation we are trying to deal with systems composed of a great many parts, or very intricate structures. Classical mathematical methods can do this only in very special situations, and their limitations are very serious. Classically, one is unable to cope with even a few simultaneous non-linear equations, to say nothing of a few dozen, or a few million. Now it is true that under certain special conditions mathematical analysis can "revive," as it were, when the situation gets complex in such a way that the parts of the system can be treated as individually and independently random—this is what happens in Statistical Thermodynamic theories. But it must be stated, explicitly and emphatically, that this is just what does *not* happen when, as in a computation system, the structure has a more organized, purposeful structure. The statistical analysis works beautifully for things like gases. It works for precious little else. There simply is no reason to suppose that as computations grow large, one will discover anything non-trivial by trying to "average-out" the effects of many events. The effect of the "conditional" (see Chapter 10) is too strong to allow anything like a "conservation" concept to have a place in the theory.

Fortunately, the systems of computation have other features that make possible some analysis, though of a very different kind. Instead of the *statistically defined* events used in physics, we use *logically-defined classes* of computations or expressions. They are tied together, not by geometric or energetic properties, but by their relations to similar machines or similar definitions. We can use machine parts so simple and with such simple interactions that we can apply the utterly transparent *Logic of Propositions*, where for an equivalent actual physical machine we would have to solve hopelessly opaque analytic equations. Chapters 1, 5, and 12 explain in detail how the problem is reduced until this kind of analysis can be made. One could even complain that the main results are obtained by a most deceptive sort of parlor trick: instead of trying to *solve* equations, we study their descriptions! It must be admitted that in the long run this

method avoids entirely too many problems, and we can be quite sure that the development of the Theory of Computation, in the years to come, will be centered around the problem of how to back off a little from this simplification, without going all the way back. The trouble now is that we do too much in one step.

AS A TEXT

This book has the form of a textbook, and I have used earlier versions of it at MIT for several years. Students specializing in computer-related studies can absorb most of the material in a one-term course, since anyone with computer-programming experience is already subconsciously primed with the essential concepts, and needs only to see them precisely formulated. For them, or for students with some higher mathematics courses, I would be inclined to assign Part I as a reading exercise, have them work the problems, and begin "teaching" with Part II. In any case, I feel that the material of Part I ought to be integrated very early somewhere in every scientific or engineering curriculum; the material of Parts II and III can come considerably later (with the exception of Computer Science students). This edition is also designed for independent reading outside the classroom, and that is why I have given substantial hints for solving most of the harder problems.

THE PROBLEMS

The reader is asked to solve a number of problems scattered throughout the book. Only a few of these are routine: most of them point toward branches the theory could take, and a few of them are difficult research problems. The emphasis is on the application of new concepts to new situations rather than on acquiring technical proficiency through drill and repetition. My view is that the best way to understand a new idea is to discover by oneself how to use it to solve one or two reasonably hard problems; the point can easily escape after a hundred simple exercises. Solving a hard problem once can make the ideas a part of one's thinking resources.

The reader is therefore enjoined not to turn too easily to the solutions; not unless a needed idea has not come for a day or so. Every such concession has a price—loss of the experience obtained by solving a new kind of problem. Besides, even if reading the solutions were enough to acquire the ability to solve such problems (which it is not), one rarely finds a set of ideas which are at once so elegant and so accessible to

workers who have not had to climb over a large set of mathematical prerequisites. Hence, it is an unusually good field for practice in training oneself to formalize ideas and evaluate and compare different formalization techniques.

ACKNOWLEDGMENTS

Writing this book required a considerable amount of time and a number of sources of support must be acknowledged. These include the RAND Corporation and numerous parts of MIT, namely, the Mathematics Department, the Electrical Engineering Department, the Research Laboratory of Electronics, the Lincoln Laboratory, and Project MAC. A first draft was written while the author was a Junior Fellow of Harvard's Society of Fellows. I explicitly want to acknowledge the support of Calvin Mooers of the Zator Company and Rowena Swanson of the AFOSR for full-time writing for one period. I would also like to thank James Winslow for his reading and comment of an early manuscript, and James E. Ricketson for many ingenious suggestions leading to the final version. Many technical points are due to Professor Manuel Blum of MIT, who used the manuscript as course notes, constructed (and solved) many of the problems, and contributed much to the final version. I want to thank my colleagues, especially Martin Davis, John McCarthy, Seymour Papert, Hartley Rogers, Jr., Dana Scott, Oliver Selfridge, and Hao Wang, for innumerable ideas in and about this area. Finally, it is even more obligatory to thank my teachers, especially Andrew Gleason and George Miller at Harvard, Professors Bochner, Fox, Lefshetz, von Neumann, Tucker, and Tukey at Princeton, and Drs. Herbert Zim and Alexander Joseph in earlier years, for the sequence of opportunities that led to this volume, and I also thank my wife, Gloria, for her enthusiasm and encouragement.

MARVIN MINSKY

CONTENTS

1 PHYSICAL MACHINES AND THEIR ABSTRACT COUNTERPARTS

1.0 WHAT IS A MACHINE?

When the term "machine" is used in ordinary discourse, it tends to evoke an unattractive picture. It brings to mind a big, heavy, complicated object which is noisy, greasy, and metallic; performs jerky, repetitive, and monotonous motions; and has sharp edges that may hurt one if he does not maintain sufficient distance.

There are many reasons why "machine" or "mechanical" have come to arouse feelings of distaste, contempt, and fear. Even today, most of the machinery we see is concerned with the use of brute power to distort and transform crude materials. Most present-day machines really *are* dangerous. Unlike our bodies, production machines are made of large, sturdy parts that need no soft sheathing for their protection. Now while it is unnecessary to attribute callousness or antipathy to a rolling-mill, it is practically impossible to attribute anything more friendly. Perhaps we even fear that any more sympathetic attitude might lead to a well-intentioned but disastrous embrace. There are occasional exceptions to this feeling. We may admire in the works of a small watch that craftsmanship required to create miniatures; we may admire the quiet competence of a high-speed computer. Neither of these are involved in the distortion of materials; they are still machines, however, and not too far to be trusted.

Somehow the notions called "mechanism" and "determinism" always seem to get mixed up with these feelings about machines. Most people believe that, by their nature, machines can do only what they are told, and will do it relentlessly. Perhaps this is why it seems to most people incredible that it might be possible to build machines capable of imaginative or creative activity. And many people feel it equally preposterous, or even

insolent, to suggest that the theory of machines might contribute to the explanation of the workings of anything so exemplary as our own minds.[1†]

People regularly resent the allegation that they might be in some sense predictable or predetermined. We will take up some of the questions clustered around the idea of "determinism" later on. There is another, much simpler, basis for the feeling that it would be undignified to be, or be like, a machine. With few exceptions, the machines that man has designed in the past have not been intellectually challenging or interesting. Even where, as computers, they have helped in problem-solving, it was usually a clear-cut matter of economy or labor-saving. No great acts of unanticipated discovery emerged directly from machine operation. Judged by past performance, the machines have shown little for men to respect.

> This popular conception of "machine" is no longer appropriate since the avalanche of practical and theoretical developments that accompanied the emergence of digital computers in the 1950's. We are now immersed in a new technological revolution concerned with the mechanization of intellectual processes. Today we have the beginnings: machines that play games, machines that *learn* to play games; machines that handle abstract—nonnumerical—mathematical problems and deal with ordinary-language expressions; and we see many other activities formerly confined within the province of human intelligence. Within a generation, I am convinced, few compartments of intellect will remain outside the machine's realm—the problems of creating "artificial intelligence" will be substantially solved.[2]

Such matters are not properly within the scope of what is studied here. However, it is important to understand from the start that our concern is with questions about the ultimate theoretical capacities and limitations of machines rather than with the practical engineering analysis of existing mechanical devices.

To make such a theoretical study, it is necessary to abstract away many realistic details and features of mechanical systems. For the most part, our abstraction is so ruthless that it leaves only a skeleton representation of the structure of sequences of events inside a machine—a sort of "symbolic" or "informational" structure. We ignore, in our abstraction, the geometric or physical composition of mechanical parts. We ignore questions about energy. We even shred time into a sequence of separate, disconnected moments, and we totally ignore space itself! Can such a theory be a theory of any "thing" at all? Incredibly, it can indeed. By abstracting out only what amount to questions about the logical consequences of certain kinds of cause-effect relations, we can concentrate our attention sharply and clearly on a few really fundamental matters. Once we have grasped these, we can bring back to the practical world this

†Superior numbers refer to notes at the end of each chapter.

understanding, which we could never obtain while immersed in inessential detail and distraction.

Actually, the exposition in these chapters is as concrete and worldly as seems compatible with the ideas. All the topics could be handled much more precisely and thoroughly by using more formal mathematical representations. In fact, most of the ideas here appeared first in mathematical publications. I do not claim that it is possible, by sufficiently skillful exposition, to "clear away all the mathematics" without any loss. For, as we shall see, some of the best ideas about the theory of machines are really inherently mathematical—*or are about mathematics itself.* In discussions that concern the nature of symbolic mathematical expressions, for instance, we will develop by definitions and examples the mathematical formalism necessary for our purposes. But my intention was to make the test accessible even to readers with no more than good high-school mathematics backgrounds, and I have explicitly marked as optional the few sections where this goal seemed completely unreasonable.

1.1 ABOUT DEFINITIONS

The term "machine" raises some serious problems of definition. For one thing, an intuitively adequate definition would have to be very complicated. This is because "machine" cannot usefully be defined as "a member of [a certain class of physical objects]." For the decision as to whether something is a machine depends on what that thing is actually *used for*, and not just on its composition or structure.

When we talk about a machine we have in mind not only (1) an object of some sort, but (2) an idea of what that object is supposed to do. When is such a pair (1, 2) thought of as a machine? A typewriter or printing press is considered to be a machine; a paperweight is not; neither is a filing cabinet. There is a continuum between the magnifying glass and the automatic-scanning, particle-counting microscope: the first certainly is not regarded as a machine, the latter certainly is. One expects a machine to have a significant number of "parts" and to perform some reasonably complex operation on something,[3] but it is difficult to capture in the same definition machines which peel apples and machines which transform coded information signals!

We often find great difficulty in giving a precise definition to a word from our ordinary non-technical language. Over some ranges we are quite sure which things belong and which don't; in other ranges we are uncertain. We may think we have a clear intuitive notion of what is involved, but when we try to define it in terms of precisely specified classes and properties, the uncertain area causes trouble.

In order to make a definition precise, sharp boundaries must be im-

posed on something. This forces us to become aware of those areas in which our intuition itself is uncertain. This is why finding appropriate definitions is so often the major effort involved in creative scientific work. If a new definition helps classify objects whose status was formerly uncertain, then some new notion must be involved. While on the surface a definition is just a convention, intellectually its acceptance may have a much more active role.

> Consider, for instance, the term "living." When is an object alive? How about viruses, genes, crystals, self-reproducing machines? No one has been able to give a definition of "living" that, in such questions, satisfies scientists in general. There are objects which are clearly living, e.g., mice; objects which are clearly not, e.g., rocks; and a now-important uncertain area. Biologists (or rather, biology teachers) used to make up lists like:
>
> (1) Self-reproducing
> (2) Irritable
> (3) Metabolizing
> (4) Made of "protoplasm," or protein, carbohydrates, DNA, etc.
>
> But (1) puts out the mule, (2) and (3) the spore, while if those conditions are dropped, (4) will admit the frankfurter. One can go on to extend the list with more careful qualifications, but questions remain until the list grows to include special mention of everything we can think of. Usually one ends up evading the problem by introducing such exotic properties as "sentient" or "adaptive," which give rise to equally serious definitional problems. Such evasions, while born of necessity, show little relation to invention.

When an intuitive idea so steadfastly resists acceptable precise definition, one must consider the possibilities either that it is not a very good idea or else that it is too comprehensive or too qualitative to serve in a technical capacity. Thus, there remains today little reason to believe that the ancient notion of *living-vs.-inanimate* has any further *technical* utility; there is no reason to expect that sharpening this old distinction will help us formulate good technical questions. For, in the borderline area, one now needs finer, sharper tools. The old terms move from within the technical discussions to the outside, taking successively the forms of chapter headings, book titles, and names of professions.[4]

1.2 MACHINES AS PHYSICAL MODELS OF ABSTRACT PROCESSES

To recognize an actual machine, we have to have some idea of what it is supposed to do. The same set of parts could be arranged either as an adding machine or as a modern sculpture. But if you know "how an

adding machine works," you should be able to recognize one. (Of course, one recognizes gears, cranks, screws, etc., as things people use to *make* machines, but that isn't the same thing. A "Meccano" set is not itself a machine.) You can even recognize a "broken" adding machine, i.e., something whose arrangement is very much like that of an adding machine except in some matter of detail that will keep it from "working properly." Obviously, the distinction between "intact" and "broken" is one which can't be defined fully in terms of physical objects—there must be an ingredient of intent (or, at least, of history). One has to have an idea of what the thing is supposed to do, and, while this idea may involve only a very general constraint on the details of the material construction, this constraint is crucial.

It doesn't much matter, for example, what the parts of an adding machine are actually made of. The gears of a digit-wheel adder could be metal, wood, or plastic—just so long as they are able to preserve certain aspects of their form to a sufficient degree. If they are too flimsy, the machine will be unreliable, or short-lived, but you can still tell that it was *supposed* to be an adding machine. In other machines, e.g., electrical circuits, even the physical (geometric) form may be quite unimportant; some other constraint is more essential.

When a particular machine is described to us, we do not first ask questions about its material construction. Given an engineering drawing, a circuit diagram, a patent description—something must first convince us that we understand how it works *in principle*. That is, we must see how it is "supposed" to work. We inquire only later whether this member will stand the stress, or whether that oscillator is stable under load, etc. But the *idea* of a machine usually centers around some *abstract* model or process.

There is a curious contrast between this idea of a machine and the idea of a "theory." Consider some "theory" of physics, e.g., Newton's mechanics. This theory (or any other theory of physics) is supposed to be a generalization about some aspect of the behavior of objects in the physical world. *If the predictions that come from the theory are not confirmed, then (assuming that the experiment is impeccable) the theory is to be criticized and modified*, as was Newton's theory when the evidence for relativistic and quantum phenomena became conclusive. After all, there is only one universe and it isn't the business of the physicist to censure *it*, much as he might like to.

For machines, the situation is inverted! The abstract idea of a machine, e.g., an adding machine, is a *specification* for how a physical object *ought* to work. If the machine that I build wears out, I censure *it* and perhaps fix it. Just as in physics, the parts and states of the physical object are supposed to correspond to those of the abstract concept. But in con-

trast to the situation in physics, we criticize the *material* part of the system when the correspondence breaks down.

Now we can formulate one of the central questions that will concern us. How can we be sure, in any particular case, that there is any way at all to build the machine we have in mind? How do we know that the physical world will permit it? In some particular cases, no reasonable person will have any doubts. Can I really build an adding machine? When we look at the structure of a simple adding machine we see that it should work; it cannot help working—this is as clear as it is that 2 + 2 = 4.

> The cliché is illuminating. Some people believe that simple mathematical statements are "self-evident." Other people maintain that they are based on rather obscure but still empirical observations. Either view may be held for machines. That a conventional adding machine will work (if properly constructed) seems on the one hand equivalent to the arithmetic propositions involved, and on the other hand seems derived from our empirical experience about the ways in which objects behave in the world. *Perhaps one could even maintain the view that belief in an arithmetic statement is equivalent to the belief that certain machines, if properly built, will work.* Thus, I know that the order of summation in addition is irrelevant. I can think of this as a property of abstract number, or as an empirical generalization from experience with counting, or as a necessary property of any machine which adds numbers correctly. Of course, there can be no belief or confidence in any process which *forms* empirical generalizations which is not ultimately based on some sort of *a priori*, unsupported, or heuristic proposition.
>
> In any case, when a realization of one of our machines fails to do what our logic and mathematics predict of it, we are quite sure that either it must be defective or broken, or else our logic must be inconsistent. And surely most people would agree that it would be as disastrous, and *as unthinkable, for an adding machine to err but not be defective* as it would be for 2 + 2 to occasionally equal 5.

Our conclusions about such questions will fall into two families. In the first part (chapters 2–4) we study the kinds of machines that can be built of finite numbers of simple parts. There is no reasonable doubt at all that *all these can be physically realized*, within easily estimated economic bounds, reliably and practically. In later sections, we show that *there are certain kinds of behavior that cannot be realized at all*, even by ideal machines with infinite numbers of parts. The theory also tells us quite a bit about problems of intermediate character.

There would be little profit in giving a more detailed picture of our exploration now. The next few chapters discuss the capacities and limitations of the "finite" machines. After that, we will be able to resume this general discussion of intuitive ideas about machines and theories.

NOTES

1. For discussion, and a theory, of why people find it difficult to accept the hypothesis that they are machines of any kind, see Minsky, "Matter, Mind and Models" [1965].† This paper attempts to explain why the "mind-brain" problem is so difficult to think clearly about and proposes some definite ways to think about consciousness, determinism, creativity, and such matters.

2. For an introduction and survey of the new field of "artificial intelligence," see *Computers and Thought*, a source book of reprints of important papers in this field, edited by Feigenbaum and Feldman [1963]. *Computers and Thought* contains a large indexed bibliography of earlier work in this area. See also the author's paper in the Sept. 1966 issue of *Scientific American*.

3. The classical idea of "simple machine"—lever, wheel, inclined plane, etc.—does not capture the spirit of what is involved in today's machines because it doesn't help understand anything except the transmission of force. We cannot explain in those terms even some parts of clockwork, such as the ratchet (an information-storage device) or the spring (an energy-storage device).

4. These kinds of questions come up continually in discussions about computers and artificial intelligence: What is intelligence? What is learning? When can one credit a machine with solving a problem, and when must one credit its designer or its programmer? A most incisive analysis of this kind of question of definition is found in chapter 1 of Lottka's *Elements of Physical Biology* [1956], originally published (in 1924) when the question of whether biological mechanisms could be accounted for by ordinary physical principles was of more serious concern than it is today. Lottka concludes (and the years have certainly confirmed) that there is little harm and perhaps much gain in recognizing our inability to state clearly the difference between living and nonliving matter—without commitment on the question of whether a well-defined difference can be found. I cannot resist quoting Lottka's citation of Sir William Bayliss' citation of Claude Bernard's citation of Poinsot's statement: "If anyone asked me to define *time*, I should reply: Do you know what it is that you speak of? If he said Yes, I should say, Very well, let us talk about it. If he said No, I should say, Very well, let us talk about something else."

†See the Bibliography for complete data on works referred to in the text and notes.

PART I

FINITE-STATE MACHINES

2 FINITE-STATE MACHINES

2.0 INTRODUCTION

In the first part of this book we will be concerned with the structure and behavior of the class of machines known as *finite-state machines* or *finite automata*. These are the machines which proceed in clearly separate "discrete" steps from one to another of a finite number of configurations or states. These highly idealized machines are of special interest for a number of reasons.

Because of their peculiarly limited, finite nature, the structure and behavior of these machines is easily described *completely*, without any ambiguity or approximation. It is much harder to deal with more realistic models of mechanical systems, in which variable quantities like time, position, momentum, friction, etc., vary smoothly over continuous, imperceptibly changing ranges of values. To analyse such "continuous" systems, one has to introduce mathematical abstractions and approximations of all sorts in order to get a workable model, and the resulting systems of equations can't usually be solved in practice without introducing further simplifications. Curiously enough, it turns out that we don't always pay a high price for restricting our attention to the finite automata—the "digital" systems as they are sometimes called. These systems are surprisingly powerful, in some respects. And in certain important ways they can be made to approximate anything that can be done by other finite physical systems. (We will discuss this further in chapter 5.)

We will deal with the finite automata in two ways. In this chapter we use a completely abstract formulation in which we treat at once the entire con-

figuration of the machine as a single quantity (its *total state*) which changes in accord with the history of the machine and its environment. In the following chapters we think of the machine as composed of many *parts*, each interacting with its neighbors, and look to see what happens, in detail, inside the machine. For these two viewpoints we use different methods of description. In the first "total-state" formulation we need only describe the set of possible states and the conditions which cause one state to change to another. In the second formulation we have to describe precisely how each part works, and how the parts are interconnected to influence one another. To make this simple, we choose for our parts some very simple devices, the "neurons" of McCulloch and Pitts, and we eventually demonstrate that the theory of all finite automata is equivalent to the theory of these particularly simple elements.

The outstanding feature of finite-state machines is the simple relation between their structure and their behavior. Once we have (1) the description of an automaton, (2) its initial condition (state), and (3) a description of the signals that will reach it from its environment, we can calculate what its state will be at each successive moment. This is worked out by a step-by-step development of the machine's successive states through time. This calculation is such a simple matter that it is always easy, in principle, to build physical models with the same sequences of events. Of course, we might not be able to afford a model large or fast enough for some purposes.

2.0.1 Moments of Time

The theory of finite-state machines deals with *time* in a rather peculiar way. The behavior of a machine is described as a simple, linear sequence of events in time. These events occur only at discrete "moments"—between which nothing happens. One may imagine these moments as occurring regularly like the ticking of a clock, and we identify the moments with integers—the variable for time, t, takes on only the values 0, 1, 2, ... Since there is no particular meaning to an origin in time, we are generally careless about where t starts, but usually we will take $t = 0$ to be the moment when the machine was first put into operation.

If one likes, he may think of the machine as operating continuously, and of our moments as corresponding to the instants at which he takes a sequence of "snapshots" of the machine's condition. Any determinate system may be viewed in this way, but it is not a very useful thing to do unless the system meets the conditions we will impose—that by the end of each interval the machine does, in effect, "settle down" into one of a finite number of describable states.

2.1 STATES AND SIGNALS

From the point of view of the user (or of the environment), a machine can usually be regarded as a closed box with input and output channels. (See Fig. 2.1-1.) From time to time the user acts on the machine through the input channels, and from time to time the machine acts on the user through the output channels. The user doesn't normally need to know just what really takes place *inside* the box. That is, unless he is particularly interested in understanding the "works" of the machine, or in modifying it, he needs to know only what are its "input-output" properties. When dealing with a machine in this way, we refer to it as a "black box," indicating our unconcern with its interior.

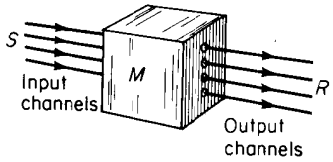


Fig. 2.1-1. A "black-box" machine.

We will treat the input and output channels with similar unconcern. For (in the theory of automata) we are no more concerned with the physical nature of the signals that pass along the channels than we are with the physical composition of the machine's parts. The signals might, for example, be electric pulses, as in a modern computer. They might be electrochemical impulses, as in nerve fibers. Or they might be mechanical impulses, like those that transfer "carry" information from one wheel to another in an adding machine. We don't care which, in automata theory.

What does concern us is the *set of distinguishable states* that might characterize a channel at a given moment. At each of our discrete moments in time, each channel will be found in one or another of a *finite* number of possible states or conditions. These states may be given any variety of symbolic names, e.g., letters of the alphabet, numbers, or even words like "yes" and "no." In most of the examples we will use, each channel will be capable of just *two* possible states—this is the usual situation in computer design.

An electric switch might be found in an "on" or an "off" position. Nominally, there are no other positions for a simple switch. The state of a switch determines, of course, whether signals will go through it—i.e., appear at the output after being presented at the input. For our purposes such a switch would be a *two-state* device, transmitting signals in one state and blocking them in the other state. (There are also "switches" with more than two states; the standard American VHF television set has a twelve-state switch for "program" selection. Any twelve symbols could be used to designate the states; the symbols in actual use are the integers 2, 3 . . . , 13.)

The assumption that the number of possible states of a channel is *finite* is one of the key postulates of automata theory. Our theory simply

doesn't concern itself with continuous or "analog" devices (like variable resistors or optical irises) which can be set to transmit any of an infinite set of fractions of a signal. Of course, when one tries to construct a physical realization of a switch one is liable—indeed certain—to encounter some degree of non-discrete behavior; most actual switches *will* change their properties just a little when the handle is pressed more firmly. But when this happens, we censure the switch (and not the automata theory) if it has any noticeable effect on the operation of the system.[1]

Returning to our black-box machine, suppose that it has just one input and one output channel. Let these be connected to an environment, E. (See Fig. 2.1-2.)

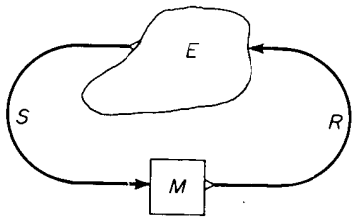


Fig. 2.1-2. Machine coupled with environment.

We will call the input channel S and the output channel R (for "stimulus" and "response"). The input, S, is capable of certain states or "signals" $s_1, \ldots, s_n$ and the output has possible states $r_1, \ldots, r_m$. At each moment t the environment E determines some input state of S as a stimulus; the chosen signal will be denoted by $S(t)$. At each moment the machine selects some output state which will act in some way on the environment; the chosen response at time t will be called $R(t)$.

Now, in order to describe a black-box machine more completely, we have to specify how it behaves—how its outputs depend on its inputs. What will be the output (state) of R at a certain time t? In general, this will depend on the entire previous "history" of the system. One restriction we will introduce here is this—the output state at the moment t does *not* depend on the input state at that same time t. This is to say that we won't admit the possibility of signals traversing a unit instantaneously. But the output at time $t + 1$ will usually depend, at least in part, on the input signal at the previous moment t.

> This restriction reflects the physical impossibility of instantaneous transmission from one place to another. It is necessary here for a simpler reason: to prevent unrealizable—paradoxical—descriptions of situations when channels are interconnected, and to prevent infinitely fast behavior—also physically unrealizable.

2.1.1 The "momentary response function" of a machine

Imagine that a machine M has been operating in interaction with some environment E for some unknown time. The machine has been receiving

signals from E and has been responding to these signals. Let $H(t)$ denote the *history* of this process up to the present time t. That is, $H(t)$ is supposed to describe, in some way, the entire record of states of affairs concerning M, from the time M was created and set into operation, up to the time t. The history, $H(t)$, of M is supposed also to include a record of all the stimuli that have entered M since it was started.

If, at time t, we were to disconnect M from its environment and insert the signal s_i, the machine would respond (at time $t + 1$) with some signal r_j. Just which signal r_j occurs at $t + 1$ would depend, of course, both on which signal s_i is chosen at time t and on the state of affairs inside M at time t. *Assuming that this state of affairs inside M is determined by the history H(t) of M,*[†] *there must be some relation, F, of the form*

$$R(t + 1) = F(H(t), S(t))$$

Unfortunately any such relation would be too hopelessly cumbersome to deal with directly, because of its explicit dependency on the entire history. It would be of great value if we could replace it by some much more direct relation between the stimulus $S(t)$ and the response $R(t + 1)$. In the next section we will show that by making a special restriction on the nature of M—namely that the machine is in some sense finite—we can obtain such a relation in a very simple and useful form.

2.2 EQUIVALENT HISTORIES: INTERNAL STATES

For a given machine M at a given time t, we can imagine an infinite variety of possible histories. The one that has actually occurred will determine the machine's response to the next stimulus. Now it may be that some events from the very remote past may contribute to determining this response function. If this is the case, one can say that the machine shows some "trace," or "memory," of those remote events. If every ancient event left a separate, independent trace, the machine would need to have infinite capacity, in some sense, to store them.

It is our plan to study just the kind of machine that can be made from a *finite* set of discrete parts, such as switches and relays, or cores and transistors. The ability of such a machine to store information about past events must have some limits. No such machine could possibly store a complete record of the events of arbitrarily long histories.[2]

[†] It could be, for some machine M, that the output $R(t + 1)$ is *not* in fact determined by the input and the history. In that case we have a "non-determinate machine." For example, the input and the history might only determine the *probabilities* of each of the possible responses. In that case we would be dealing with a "probabilistic machine." We might also consider non-determinate machines which aren't probabilistic, but, except for certain technical advantages which won't concern us, there is no reason to do so here.

If its memory is limited, the machine cannot remember everything that has happened to it; *it cannot distinguish, in its behavior, between all possible histories.* It is important to consider the following definitions carefully.

> We need to make precise the notion of two histories being equivalent for a machine. Imagine that once there were two identical copies of a machine M. But at time t, perhaps because of different past environments, M_1 has history $H_1(t)$ and M_2 has a different history $H_2(t)$. We will say that H_1 and H_2 are *equivalent histories* with respect to M if, *for every possible subsequent sequence of signals* $S(t)$, $S(t+1)$, $S(t+2)$, ... both M_1 and M_2 would yield *the same output sequences*. That is, there is no way to tell which machine is M_1 and which is M_2 by testing them with input sequences and observing their responses.
>
> Now define the "equivalence class" of any history to be the set of all histories equivalent to it. Since two histories equivalent to the same history are equivalent to each other, all members of an equivalence class are equivalent in pairs. Clearly, two equivalence classes cannot partially overlap; they must be completely separate ("disjoint" classes) or completely identical.

This brings us to the key postulate of the theory of finite automata. We assume that *the machine can distinguish, by its present and future behavior, between only some finite number of classes of possible histories. These classes will be called the "internal states" of the machine.* We will designate the internal state of the machine, at time t, by the symbol $Q(t)$, and we will call the states themselves $q_1, \ldots, q_p$.

Evidently, there is a connection between the equivalence classes of histories and the kinds of things a machine can remember about its past. In chapter 4 we will examine in detail the nature of such memories.† It is very interesting that, given only the postulate that the number of equivalence classes is finite, we can say quite a good deal about the possible structures of such classes.

2.2.1 The defining functions, F and G

By our definition of internal state, the machine's response $R(t+1)$ to $S(t)$ can depend only on $Q(t)$. Otherwise the behavior could depend on things other than the distinguishable classes of histories. A better way to put this is to write

$$R(t+1) = F(Q(t), S(t)) \qquad (F)$$

†One can imagine, of course, that there are parts inside the machine which remember things that are *never* expressed in the machine's behavior. For example, the machine might have some parts with inputs but no outputs. Our definition ignores any such internal information, and it might be better to talk of our classes of histories as "external states." Cf. Moore [1956]. Our states are those of Moore's "reduced machines"—i.e., machines that have been simplified as far as possible, with "all unessential parts of its description removed."

i.e., the output at $t+1$ depends only on the input state at time t and the internal state at time t. The input state or signal $S(t)$ depends, of course, on the environment. On what does the internal state $Q(t)$ depend?

The internal state at time t depends, of course, on the whole history of the machine. Fortunately, *we can obtain a statement which separates the dependencies with respect to the immediate past and the remote past.*

The history of the machine at time t is a listing of all the things that have ever happened to it—the event in which it was first put into operation and the sequence of inputs it received after that. The history at time $t+1$ differs from the history at time t only in having an extra term; this describes what happened at time t, namely the signal $S(t)$. By hypothesis, nothing else can influence what happens within the black box. Now what *internal state* has the machine at time $t+1$? *The state* $Q(t+1)$ *of the machine can depend only on the previous input* $S(t)$ *and on the previous internal state* $Q(t)$. $Q(t+1)$ could not depend on more than this, or the machine could distinguish, in future behavior, between two histories which were both supposed to be in the same class $Q(t)$ and hence indistinguishable.† We can express this conclusion by writing

$$Q(t+1) = G(Q(t), S(t)) \qquad (G)$$

The two functions or relations F and G give us as complete a description of the machine as we could possibly want, so long as we remain unconcerned about what actually happens *inside* the black box. For, suppose that we are told the internal state $Q(t)$ of the machine at a certain time t and also the sequence $S(t)$, $S(t+1)$, $S(t+2)$, etc., of stimuli that it is to receive from that time on. Then relations F and G tell us what output $R(t+1)$ to expect at the next moment $t+1$ and also the internal state $Q(t+1)$ the machine will have at that time. Using this latter fact and relations F and G again, we can determine the output and internal state at $t+2$. *We can continue this calculation, step by step, to find the response and state of the machine at any future time (so long as we know what the inputs will be).*

Machines characterized by a finite number of internal states will be called *finite-state machines*. Other names have been used, e.g., *finite-state processes, finite automata, finite-state transducers*, etc. We will now study some of these machines in great detail. The finite-state restriction may at first seem severe, but in a practical sense every machine we can build can be approximated as closely as we like, by defining sufficiently many states.

†Thus, once the dependency of $Q(t+1)$ on $Q(t)$ and $S(t)$ has been stated, there can be no further dependency, e.g., on $S(t-1)$, that is not already included. The logic of this paragraph is quite intricate, and only a mathematically mature reader can follow it without going back, once or twice, to the definition of equivalent histories. A less intricate alternate approach is given immediately below in section 2.2.2.

Chapter 5 discusses this alarming thesis in more detail. In Part II we discuss machines which are, in a sense, infinite. But we do this mainly for technical reasons which make it easier to learn about the ultimate *limitations* of all machines, including finite machines. So long as we are concerned with *negative* results, as is usually the case in Part II, the admission of infinite machines only weakens the assumptions, and hence strengthens the conclusions, so far as the finite machines are concerned!

2.2.2 Remarks, and other definitions of finite-state machines

The idea of a finite-state machine, and in particular the notion of internal state, is usually developed in a less abstract manner than in the previous section. Our goal is ultimately to explain some ideas about automata theory that involve a rather abstract way of looking at sequences of stimuli. By introducing part of this abstraction right in the definitions, the remaining jump (in chapter 4) will not seem so large.

The immediately following sections are of a more practical nature, and we hasten to give a more informal picture of finite-state machines.

Imagine that inside our black box are a number of interconnected parts. Suppose that each part has just a small number of distinct states or positions. Then we can describe the *total state* of affairs inside the box by simply listing the states (at that moment) of each and all of the parts. If there are only a finite number of parts, then there is possible only a finite number of distinct total states of affairs. These total states of the machine are what we call its "internal states." If, e.g., there were n parts, each capable of K states by itself, then the machine could have up to K^n distinct total states.† (The number of internal states actually available in operation might be much less than this if there were constraints between the parts, as is the case in most real machines.) We designate these internal states (the possible states of affairs inside the box) by the symbols $q_1, \ldots, q_p$, and the internal state at time t by the symbol $Q(t)$.

Now, the input and output channels connected to our box must connect to some parts inside the box:

The input channel connects to certain parts of the machine so that the input signals may affect the states of some of those parts. Obviously, the internal state at time $t + 1$ depends on (and only on) the state of affairs at time t and the signals that came in at time t. So again we can say that

$$Q(t + 1) = G(Q(t), S(t)) \qquad (G)$$

†Because two of these "total internal states" might be indistinguishable in the machine's external behavior, the "internal states" defined here may correspond to smaller classes of histories than do those defined in section 2.2. But it makes absolutely no difference, in what follows, which definition we use!

The output channel originates in certain parts of the machine so the state of the output channel depends on the states of those parts. Since the states of these parts at time $t + 1$ depends only on their states at time t and on $S(t)$ we can also say that

$$R(t + 1) = F(Q(t), S(t)) \qquad (F)$$

While we can think of the machine as a single process (described completely by the functions F and G), there may be many things going on simultaneously inside the "black box." The input signals directly affect only some of the parts. At the next moment these parts affect the states of other parts. In turn, more remote parts may be affected at later times. Activity and information can thus spread and circulate to remote parts of the network inside the machine. But before we examine some of these possibilities in detail, there are a number of things to be said of the machine viewed as a unit.

An outsider could influence the machine by pouring signals into the S channel. Similarly, the resulting output of the machine may succeed in influencing the outsider's behavior. Referring to the earlier diagrams (Figs. 2.1-1 and 2.1-2), we can note that the situation from the point of view of the machine, as shown in Fig. 2.2-1, is pretty much the same. The

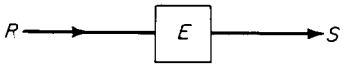


Fig. 2.2-1. An environment.

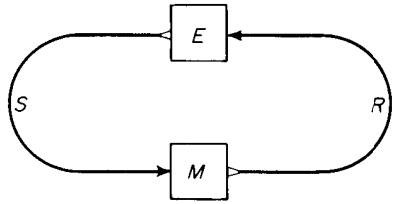


Fig. 2.2-2.

only schematic difference is that the names of the communication channels are interchanged and now the *environment* is inside the box. If we are willing to treat the environment also as a black-box machine, we have the symmetrical situation of Fig. 2.2-2.

The question of where a particular machine ends and its environment begins can be settled only by a convention or definition. The body can be regarded as the environment for the brain or for the liver. The distinction between the essential part of a machine and the inessential trim depends on one's choice of values. When we cannot grasp a system as a whole, we try to find divisions such that we can understand each part separately, and also understand (in that framework) how they interact. When we make such a division for purpose of analysis, each part is treated in turn as the machine of interest and the remainder as its environment. One cannot usefully make such divisions completely arbitrary because an unnatural division of a system into "parts" will not yield to any reasonable analysis.

2.3 STATE TRANSITION TABLES AND DIAGRAMS

A finite-state machine is completely described when we are given its "transition" functions F and G. Each of these functions is defined for only a finite number of input values so that the functions themselves can be represented as simple tables. Our first example is a "memory" or "delay" device whose transition functions are given below.

EXAMPLE 1: A "MEMORY" MACHINE

The input channel can transmit just two kinds of signals, called s_0 and s_1, and there are just two internal states, q_0 and q_1, and two output signals, r_0 and r_1:

		STATE	
	G	q_0	q_1
INPUT	s_0	q_0	q_0
	s_1	q_1	q_1

		STATE	
	F	q_0	q_1
INPUT	s_0	r_0	r_1
	s_1	r_0	r_1

The tables show that the state at time t, (or the output at time $t + 1$) tells whether the signal digit entered at time $t - 1$ was s_0 or s_1. (Problem: Show that a machine whose output at time $t + 1$ tells what was the signal at time t requires only one internal state.[3]) In view of the built-in delay, of one time unit, between input and output, the machine just described may be regarded as having a rudimentary ability to "remember"—in this case to "store" the digit that occurred at time $t - 1$.

The next simple machine keeps track of the *parity* (odd- or even-ness) *of the number of* 1's *it has received.* From here on we represent s_0 and s_1 *and also* r_0 and r_1 by 0 and 1, respectively.

EXAMPLE 2: A "PARITY" MACHINE

G	q_0	q_1
0	q_0	q_1
1	q_1	q_0

F	q_0	q_1
0	0	1
1	1	0

The transition tables for this machine show that the state (and output) *remains the same* when a 0 is entered, but *changes* when 1 is entered. Hence any *even* number of 1's will cause no net change of state. Thus if we knew that at some (perhaps remote) time in the past the machine was in state 0 (say), we could tell at any later moment whether the number of 1's between the first time and the second was even or odd. Or if we knew (say) that the machine began its existence in state 0, we could tell whether the total number of 1's that had *ever* entered it was odd or even.

This machine can also be said to have a feeble kind of memory. This memory is certainly limited in capacity—it can distinguish only two classes of past histories. Let us note, however, that its memory *isn't* limited by any fixed span of time, at least in the usual sense. Unlike human memory, its state at a given moment is no more (and no less) influenced by the signals of recent incidence than by those of arbitrarily ancient occurrence. (Contrast this with the memory of the delay machine of Example 1, which is influenced exclusively by the most recent signal.) The point to be noted is that *although finite-state machines are limited in memory capacity, this limit need not take the form of a simple bound on the duration of the effect of a particular signal.* We will see just what the limitations are in chapter 4.

When we deal with larger numbers of states, these table descriptions become awkward and it is convenient to use more graphic descriptions. We call them *state-transition diagrams*, or just *state diagrams*. The state diagrams for the two simple machines just described are shown in Figs. 2.3-1 and 2.3-2.

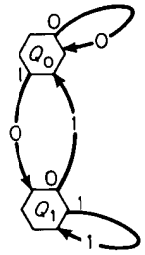


Fig. 2.3-1. Memory machine.

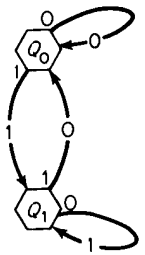


Fig. 2.3-2. Parity machine.

Here are the rules for interpreting these diagrams: We will always use *hexagons* to represent states. *Arrows* represent the transitions from one state to another. Each input symbol occurs at the tail of one of the arrows leaving each state. Each arrow shows what will happen, given the state and input conditions shown at its tail. The head points to the subsequent state as given by the values of the function G. The symbol inserted in the middle of each arrow designates the output signal that will occur, as given by the values of the function F.

When there is no ambiguity we will feel free to omit some of the symbols.

EXAMPLE 3. A TWO-MOMENT DELAY MACHINE

Figure 2.3-3 represents a delay machine which remembers the *two* previous digits; its state at time t, or its output at time $t + 1$, tells which signal entered at time $t - 2$.

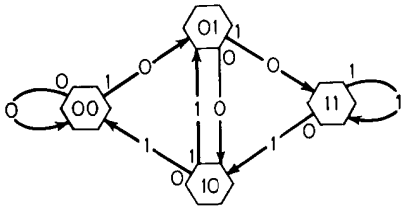


Fig. 2.3-3. Two-moment delay machine.

G	q_{00}	q_{01}	q_{10}	q_{11}
0	q_{00}	q_{10}	q_{00}	q_{10}
1	q_{01}	q_{11}	q_{01}	q_{11}

F	q_{00}	q_{01}	q_{10}	q_{11}
0	0	0	1	1
1	0	0	1	1

One could write, briefly

$$G(q_{ij}, s_k) = q_{jk} \qquad F(q_{ij}, s_k) = i$$

EXAMPLE 4. A THREE-MOMENT DELAY MACHINE

Figure 2.3-4 shows a machine which will remember three digits. It is easy to remember longer sequences of digits by increasing the number of states. Of course, the number of states will grow rapidly. There is no escape from the requirement that there must be 2^n states to remember n binary digits. (Why not? Describe a technique for making an n-digit memory, and give an argument for why it needs at least 2^n states.)

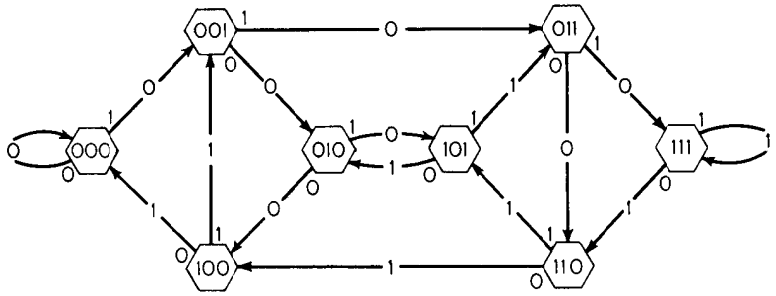


Fig. 2.3-4. Three-moment delay machine.

EXAMPLE 5. A BINARY SERIAL ADDING MACHINE

A final example of a state diagram will be a binary adding machine.[4] This machine is given for its inputs two strings of binary digits, simultaneously, where each string represents a number in binary form *with least significant digits first*. The output signals of this machine are the binary digits of the sequence which represents the sum of the two numbers put into the machine. Beginning at some starting moment, the machine receives a sequence of pairs of binary digits (where each digit of a pair belongs to one of the two numbers being fed into the machine). This means that there are really four possible input signals, which we can call 00, 01, 10, 11. Only two states are needed; a "no-carry" state q_0 and a

"carry" state q_1. (See Fig. 2.3-5.) For example, the sum 45 + 57 = 102 has the binary form

$$\begin{array}{r} 1\ 0\ 1\ 1\ 0\ 1 \\ +\ \ 1\ 1\ 1\ 0\ 0\ 1 \\ \hline =\ 1\ 1\ 0\ 0\ 1\ 1\ 0 \end{array}$$

or

$$\begin{array}{r} 32 + 0 + 8 + 4 + 0 + 1 \\ +\ 32 + 16 + 8 + 0 + 0 + 1 \\ \hline 64 + 32 + 0 + 0 + 4 + 2 + 0 \end{array}$$

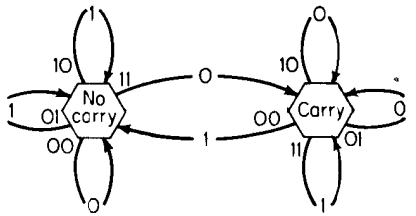


Fig. 2.3-5. Serial binary adding machine.

and would cause the sequence of events shown in Table 2.3-1.

Table 2.3-1

t	0	1	2	3	4	5	6	7	...
45	1	0	1	1	0	1†	0	(0)	
57	1	0	0	1	1	1†	0	(0)	
signal	11	00	10	11	01	11	00	(00)	
state	q_0	q_1	q_0	q_0	q_1	q_1	q_1	q_0	
output	–	0	1	1	0	0	1	1†	

† Note that the digits of the binary numbers are written backwards.

PROBLEM 2.3-1. Write out the F and G tables for the machine of Fig. 2.3-5.

2.4 THE STATE-TRANSITION DIAGRAM OF AN ISOLATED MACHINE

What happens if a finite-state machine is left to itself—that is, if it receives no information from the outside world? That would be the case if the machine receives only a constant input—that is, a repetitive, unchanging signal. In this case the state-transition diagram becomes very simple. Since there is only a single signal involved, each state uniquely determines what the next state will be. In terms of the diagram, *there is only a single arrow leading from each state.* Of course, it is still possible for several arrows to lead *into* the same state. The diagram must therefore look something like Fig. 2.4-1. Note that this is not a collection of examples; it is the diagram of a *single* machine.

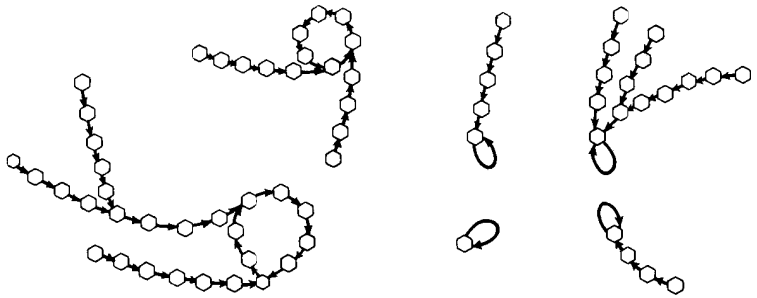


Fig. 2.4-1. State diagram of a certain isolated machine. Note that this is all one machine—not several.

Choose some state for a start, and set the machine into operation. The machine will progress from state to state, moving along one of the chains in a diagram. Now there are only a finite number of states. Clearly, if the machine is run for a long enough time, it must eventually re-enter a state it has previously been in. And this means that it must, from that time on, continue in a periodically repeating pattern! Each chain must eventually lead to a closed loop or "cycle." Of course, different starting states may lead into the same cycle—i.e., different starting chains may *merge. But two paths, once merged (by entering a common state) can never diverge again.*[†] Hence the state diagram contains a finite number of separate, closed loops or "cycles." Each cycle has, leading into it, a number of distinct "trees" or merging sets of paths. There is no way to pass from the tree system of one cycle to that of another.[‡]

We can conclude from these observations about the state diagram, that *any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern.* The duration of this repeating pattern cannot exceed the number of internal states of the machine, and could be much less. Of course, neither can the duration of the epoch *before* the machine becomes repetitive exceed this number.

This conclusion shows that in a certain sense, finite machines are rather trivial. But in making such an interpretation, one should keep in mind the very large numbers of states that practical machines may have. A modern digital computer may have, in its high-speed "memory," the order of a million small parts, each of which has two states. The number of *total states* of such a machine is the *product* (not the sum) of these, since any set

[†] Why not?

[‡] We call each such set, consisting of a cycle plus all the trees attached to it, a (connected) "component" of the diagram. The distinct components are necessarily completely detached from one another. (Note that if we use a different "constant" signal we might get a different set of components—trees and loops.)

of conditions of these parts is possible. This means that the machine as a whole has available roughly $2^{1,000,000}$ states! (This is a 1 followed by about three hundred thousand zeros.) And the state diagram of such a machine (without input) can indeed contain cycles whose lengths are of such magnitudes. Even if such a machine were to operate at the frequencies of cosmic rays, the aeons of galactic evolution would be as nothing compared to the time of a journey through such a cycle.

The enormous magnitudes of the numbers of states in the memory of a modern computing machine suggest that one should be suspicious about the apparent implications of the finite-state condition. In the following chapters we will see a number of theoretical limitations of finite automata. But the magnitudes involved should lead one to suspect that theorems and arguments based chiefly on the mere finiteness of the state diagram may not carry a great deal of practical significance.

PROBLEM 2.4-1. If the reader is familiar with the operation code of some digital computer, he should try to discover the program with the longest possible running time (before repeating or halting). He will find that the solution will be, essentially, a program that treats the entire memory, save the space for the program itself, as a single gigantic accumulator. A fine point: a common mistake is to try to include, in the giant cycle, the states of the additional "hardware" available, e.g., index registers and accumulators. This turns out to be a mistake (unless there is a large systematic array of such units), for the additional *program* required to exploit these registers will consume more states than are gained by this tricky strategy.

2.5 STATE TRANSITIONS IN THE PRESENCE OF EXTERNAL SIGNALS

It is much more difficult to give a general description of what happens when there are variable inputs to the machine. One kind of complete description in mathematical terms is given in chapter 4; here we can go a small step in this direction by considering the case in which there is only an occasional signal from the outside.

External signals can induce state transitions not permitted in the constant or no-signal state diagram. Normally the machine will be moving around in one or another component of its no-signal, tree-cycle state diagram. An external signal can throw the machine into a state of a different component (or to a distant part of the same component). Since the new state depends on both the signal *and* the old state, the component into which the machine goes will depend on the exact state (and not just the component) at the previous moment.

One might regard each component of the state diagram as a separate *computation* the machine can perform. If the signal leaves the machine in the same component, it can be regarded as merely advancing or retarding

that computation. If the component is changed, then the signal changes the computation. Alternatively, the selection of a component could be regarded as storage in the machine of some information about the signal. In the long run, memory cannot be stored in the form of individual states, but only as different components of the state diagram.†

For example, the parity counter (Fig. 2.3-2) has a state diagram which (in the absence of 1 signals) divides into two components. The occurrence of a 1 always causes a transition from one component to the other. (See Fig. 2.5-1.) If no 1 occurs, this machine can "remember" arbitrarily long.

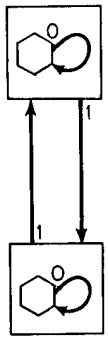


Fig. 2.5-1.

The machine of Fig. 2.3-4 *in the presence of a constant signal,* e.g., 0, *has only one component,* as in Fig. 2.5-2. It cannot remember anything from the *remote* past; indeed it can remember no trace of anything that happened more than three moments ago.

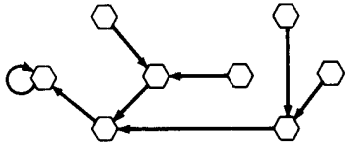


Fig. 2.5-2.

When input signals change more than occasionally, this analysis in terms of components of the state diagram isn't very helpful, because the components themselves are only defined with respect to particular constant signal conditions. Then analysis requires methods like those of chapter 4.

2.6 THE "MULTIPLICATION PROBLEM": A PROBLEM THAT CANNOT BE SOLVED BY ANY FINITE-STATE MACHINE

The machine of example 5 of section 2.3 describes a finite-state machine which can add, serially, two binary numbers of arbitrary length. The reason that this is possible, for a single, fixed-size machine, is that it is never necessary to "carry" or "remember" very much information during *addition*—a single carry digit will suffice.

Multiplication is different in this respect. Depending on the size of the numbers involved, it can become necessary to carry arbitrarily large amounts of information during the course of multiplications. We have

†Except for phase information, which may be useful if there is an independent synchronized clock.

to "accumulate partial sums" to get the final answer. In fact we will prove:

THEOREM

No fixed finite-state machine can multiply arbitrarily large pairs of binary (or decimal) numbers.

Suppose that there were a machine M that could multiply arbitrarily long strings of (say) binary numbers. As in the addition example (example 5, section 2.3), we suppose that the input quantities are presented simultaneously digit by digit, least significant first. We will ask M to perform a relatively simple multiplication; to multiply the number 2^n by itself. Now 2^n has the form of a single 1 followed by n zeros in binary notation. And the product of 2^n with itself, namely 2^{2n}, is a 1 followed by $2n$ zeros. Since there are $2n + 1$ digits in this product, and only $n + 1$ in either of the multiplicands, this means that the machine *has to print n zeros after the input has stopped, and then print the 1.*

But once the input has stopped, M must operate as a machine with constant (0) input (see section 2.4). If M has fewer than n states, then, if it does succeed in printing n zeros this must be because it is in a loop of its state diagram which yields only zero outputs. Hence it must go on printing zeros forever, and it cannot ever print the required 1. Since for any fixed finite-state machine, we can always make n larger than the number of states, there can be no finite-state multiplying machine which will work for arbitrarily large numbers.

We have discovered an example of a kind of problem beyond the reach of any finite-state machine.† This is the first of a number of limitations we will discover. Here the trouble is quite clearly due to the machine's finite memory capacity. In Part II we will find further limitations that persist even when the machine is given, in effect, an infinite memory capacity—so that it need never repeat a "total state."

2.7 PROBLEMS

PROBLEM 2.7-1. Describe (in words) the class of histories represented by some internal state of each of the examples in this chapter.

PROBLEM 2.7-2. Each output signal—or output state—of a machine represents an equivalence class of histories. These may not be the same as

†The informal proof given here may seem rather flimsy. For example, one might still wonder whether multiplication could be done with the numbers presented with their digits in another order—or in some different number system. But the idea of the proof will in fact carry over to all such changes in the problem.

those represented by the internal states, although they are closely related. What history is represented by the output 0 of the machine in Fig. 2.7.1? Output 1?

PROBLEM 2.7-3. Show that no isolated machine can output the infinite sequence:

$$1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1 \ldots 0^n\ 1\ 0^{n+1}\ 1 \ldots$$

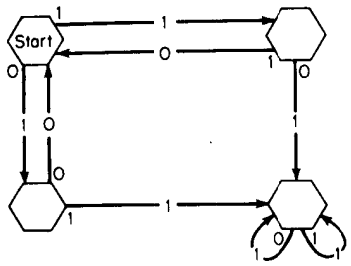


Fig. 2.7-1.

PROBLEM 2.7-4. Why is it unnecessary to include a history of outputs of a machine in its history $H(t)$?

PROBLEM 2.7-5. We reprint here an interesting problem from Moore, *Sequential Machines* [1964, pp. 213-214].

"The problem known as the *firing squad synchronization problem* was devised about the year 1957 by John Myhill, but so far as I know the statement of the problem has not yet appeared in print. It has been widely circulated by word of mouth, and has attracted sufficient interest that it ought to be available in print. The problem first arose in connection with causing all parts of a self-producing machine to be turned on simultaneously. The problem was first solved by John McCarthy and Marvin Minsky, and now that it is known to have a solution, even persons with no background in logical design or computer programming can usually find a solution in a time of two to four hours. The problem has an unusual elegance in that it is directly analogous to problems of logical design, systems design, or programming, but it does not depend on the properties of any particular set of logical elements or the instructions of any particular computer. I would urge those who know a solution to this problem to avoid divulging it to those who are figuring it out for themselves, since this will spoil the fun of this intriguing problem.

"Consider a finite (but arbitrarily long) one-dimensional array of finite-state machines, all of which are alike except the ones at each end. The machines are called soldiers, and one of the end machines is called a General. The machines are synchronous, and the state of each machine at time $t + 1$ depends on the states of itself and of its two neighbors at time t. The problem is to specify the states and transitions of the soldiers in such a way that the General can cause them to go into one particular terminal state (i.e., they fire their guns) all at exactly the same time. At the beginning (i.e., $t = 0$) all the soldiers are assumed to be in a single state, the quiescent state. When the General undergoes the transition into the state labeled 'fire when ready,' he does not take any initiative afterwards, and the rest is up to the soldiers. The signal can propagate down the line no faster than one soldier per unit of time, and their problem is how to get all coordinated and in rhythm. The tricky part of the problem is that the same kind of soldier with a fixed number K of states is required to be able to do this, regardless of the length n of the firing squad. In particular, the soldier with K states should work correctly, even when n is much larger than K. Roughly speaking, none of the soldiers is permitted to count as high as n.

"Two of the soldiers, the General and the soldier farthest from the General, are allowed to be slightly different from the other soldiers in being able to act without having soldiers on both sides of them, but their structure must also be independent of n.

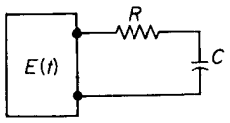
"A convenient way of indicating a solution of this problem is to use a piece of graph paper, with the horizontal coordinate representing the spatial position, and the vertical coordinate representing time. Within the (i, j) square of the graph paper a symbol may be written, indicating the state of the ith soldier at time j. Visual examination of the pattern of propagation of these symbols can indicate what kinds of signaling must take place between the soldiers.

"Any solution to the firing squad synchronization problem can easily be shown to require that the time from the General's order until the guns go off must be at least $2n - 2$, where n is the number of soldiers. Most persons solve this problem in a way which requires between $3n$ and $8n$ units of time, although occasionally other solutions are found. Some such other solutions require $\frac{5}{2}n$ and of the order of n^2 units of time, for instance. Until recently, it was not known what the smallest possible time for a solution was. However, this was solved at M.I.T. by Professor E. Goto of the University of Tokyo. The solution obtained by Goto used a very ingenious construction, with each soldier having many thousands of states, and the solution required exactly $2n - 2$ units of time. In view of the difficulty of obtaining this solution, a much more interesting problem for beginners is to try to obtain some solution between $3n$ and $8n$ units of time, which, as remarked above, is relatively easy to do."

NOTES

1. Computer specialists often talk of a distinction between "digital" and "analog" quantities, where a digital quantity is one that takes on only one of a fixed, finite set of values while an analog quantity has values from a continuum, e.g., the voltage measured across a resistor or capacitor. The term "analog" comes from the era in which most "computers" were electrical or mechanical devices designed to approximate the differential equations of continuous physical systems, in such a way that one could think of an analogy between the physical behavior of the computer and the other system. In this sense, we are deciding here to develop our theory around ideas of finite mathematics rather than infinite, continuous, differential, mathematics. In chapter 9 we discuss one way of bringing these viewpoints back together.

2. If one were to consider "analog" devices (see note 1) and were also to ignore basic physical considerations about noise, quantum measurement uncertainty, and the like, then one might consider devices like the following to have infinite



memory: Suppose that the device E in the accompanying figure is a voltage source that has been, *for infinite past time,* producing a voltage $E(t)$ with the property that for each integer j, $E(t) = e_j$ if $j < t \leq j + 1$ where e_j is either 0 or 1. Suppose also that R and C are such that whenever $E(t)$ is zero, the charge on C will decay to half its value in one time unit. Then at any integer time t_0, the voltage on C will be given by the expression

$$\sum_{k=0}^{\infty} E(t_0 - k)2^{-k}$$

so that by measuring this voltage with infinite precision and expressing it as a real binary fraction, one can recover infinite information about the past. We do not want to admit such a physical absurdity, and so we build our theory upon basically finite ingredients, extending later to the infinite question by another route. See also the deeper analysis in chapter 5.

3. The fact that we can make a delay—or memory—machine with only one state, namely

G	q_0
s_0	q_0
s_1	q_0

F	q_0
s_0	r_0
s_1	r_1

suggests that there is something unnatural about the way we defined the output function. Indeed, we could reformulate the theory so that the output at time $t + 1$ depends *only* on the state at time $t + 1$, *or* we could define it to depend only on the state at time t. Technically, these would lead to the same basic results; each has its own slight advantages and disadvantages. Our choice of definition was made so that the constructions of chapter 3 and 4 would come out with the least complication.

4. Most readers will be familiar with binary numbers. Just as, in the decimal system, a number like 3065 means

$$\underline{3 \times 1000} + \underline{0 \times 100} + \underline{6 \times 10} + \underline{5 \times 1}$$

a number like 1011, in the binary system, means

$$\underline{1 \times 2^3} + \underline{0 \times 2^2} + \underline{1 \times 2^1} + \underline{1 \times 2^0}$$

or

$$8 + \qquad\qquad 2 + \qquad 1 = 11$$

The rule for addition of binary numbers is given in detail in chapter 3, section 3.2.6. It is just like decimal addition, except that when one has to add 1 + 1, one has to recognize that the sum is "10," i.e., "write zero and carry the one." Thus

$$\begin{array}{r} 45 \\ +57 \\ \hline ** \\ 102 \end{array} \quad \leftrightarrow \quad \begin{array}{r} 101101 \\ +111001 \\ \hline * \ ** \ * \\ 1\ 100110 \end{array}$$

where the asterisk shows where "carries" have to be made. Binary multiplication (used in section 2.6) is done, also in analogy to decimal arithmetic, by repeated addition or, better, by shifted addition:

$$\begin{array}{r} 11 \\ \times\ 5 \\ \hline 55 \end{array} \quad \leftrightarrow \quad = \begin{array}{r} 1011 \\ \times\ 101 \\ \hline 1011 \\ +\ 0000 \\ +\ 1011 \\ \hline 110111 \end{array}$$

Almost any introductory text on computers will give a thorough introduction to binary arithmetic; we will use it only for examples.

3 NEURAL NETWORKS. AUTOMATA MADE UP OF PARTS

3.0 INTRODUCTION

The black-box approach (section 2.1) is useful only when we understand a machine's external behavior completely and don't care what is inside. Otherwise the machine's structure has to be considered in more detail, i.e., in terms of its parts and their interconnections. Once we understand perfectly each part and how it interacts with the others we have a chance of understanding the machine as a whole.

The machines discussed in this chapter are made of particularly simple parts. In fact, each part is nothing more than a very simple two-state machine. The interconnections, too, are quite simple. But from a very small variety or "stock" of these simple parts we can build up the equivalent of any other finite-state machine.

We will use for our parts the elements developed by McCulloch and Pitts [1943] as models for certain aspects of brain function.

> It should be understood clearly that neither McCulloch, Pitts, nor the present writer considers these devices and machines to serve as accurate physiological models of nerve cells and tissues. They were not designed with that purpose in mind. They are designed for the representation and analysis of the logic of situations that arise in any discrete process, be it in brain, computer, or anywhere else. In theories which are more seriously intended to be brain models, the "neurons" have to be much more complicated. The real biological neuron is much more complex than our simple logical units—for the evolution of nerve cells has led to very intricate and specialized organs.[1] At this point, the McCulloch-Pitts "cells" or "neurons" are quite sufficient for our purposes. Our present goal is only to indicate how, starting with a set of very simple elements, one can construct machines of all sorts.

Accordingly, the automata made up of these elementary units are usually called *neural networks*.

3.1 THE "CELLS" OF McCULLOCH AND PITTS

Our machines will be assembled by interconnecting certain basic elements called "cells." Each cell will be represented in our diagrams by a circular figure:  . From each cell leads a single line or wire, called the "output fiber" of the cell. This output fiber may branch out after leaving the cell; each branch must ultimately *terminate* as an input connection to another (or perhaps the same) cell. Two types of termination are allowed. One, called an *excitatory input*, is represented by a little black arrow: ⟶. The other, called an *inhibitory input*, is represented by a little outline circle: —○. We permit any number of input connections to a cell. Figure 3.1-1 shows a few typical cell configurations, with names whose meaning will become evident later in the chapter.†

We build up more complex machines by interconnecting cells to form "neural networks." The only restriction on the formation of such nets is that, while we permit output fibers to branch out to form arbitrarily many

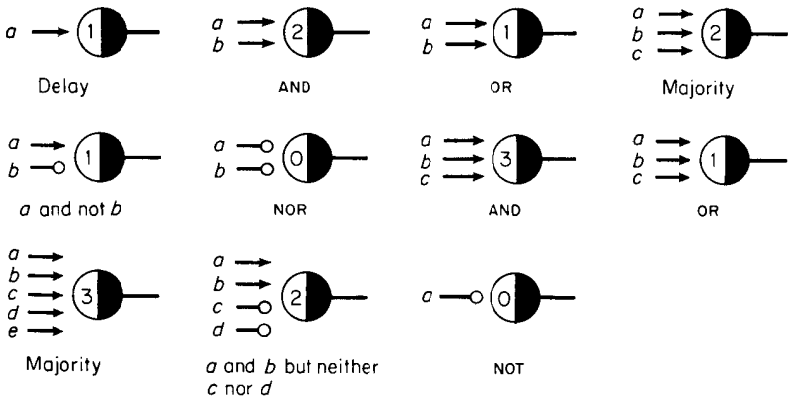


Fig. 3.1-1. Some McCulloch-Pitts cells.

†The diagrammatic notation in use by neurophysiologists seems startling when one discovers that in the diagram below, the signals flow from left to right! It is hard to believe there exists a culture in which the arrow points the wrong way, so to speak. Of course, this is due to an attempt to caricature the actual appearance of some nerve cells in some nervous systems. I could not bring myself to further propagate this traditional nuisance.

input terminations, we do not allow output fibers from different cells to fuse together. A glance at the illustrations in this chapter will show what is permitted.

Each cell is a finite-state machine and accordingly operates in discrete moments. The moments are assumed synchronous among all cells. At each moment a cell is either *firing* or *quiet*; these are the two possible states of the cell. For each state there is an associated output signal, transmitted along the cell's fiber branches. It is convenient to imagine these signals as short waves or pulses transmitted very quickly along the fiber. Since each cell has only two possible states, it is even more convenient to think of the firing state as producing a pulse while the quiet state yields *no pulse*. (One may think of "no pulse" as the name of the signal associated with the quiet state.)

Cells change state as a consequence of the pulses received at their inputs. In the center of the circle representing each cell there is written a number, called the *threshold* of that cell. This threshold determines the state-transition properties of a cell C in the following manner.

At any moment t, there will be a certain distribution of activity on the fibers terminating upon C. We ignore all fibers which are quiet at this time, and look to see if any *inhibitory* inputs are firing. *If one or more of the inhibitors are firing, then C will not fire at time* $t + 1$. Otherwise—if no inhibitor is firing—we count up the number of excitatory inputs that are firing; *if this number is equal to or greater than the threshold of C* (the number in the circle), *then C will fire at time* $t + 1$.

In other words: *a cell will fire at time* $t + 1$ *if and only if, at time t, the number of active excitatory inputs equals or exceeds the threshold, and no inhibitor is active.* A cell with threshold 1 will fire if *any* excitatory fiber is fired and *no* inhibitor is. A cell with threshold 2 requires *at least two* excitations (and no inhibition) if it is to fire at the next moment. A cell with threshold 0 will fire at any time *unless* an inhibitor prevents this. Fig. 3.1-2 illustrates the behavior of a number of different cells.

Remarks

(1) The reader will note that the state of a cell at $t + 1$ doesn't depend on its state at time t. These are very simple "neurons" indeed. One notable property of *real* neurons is that once having fired, there is an interval during which they can't be fired again (called the refractory interval), and this illustrates a real dependency on the previous state. However, we will see (section 3.6) that it is easy to "simulate" such behavior with groups of McCulloch-Pitts cells.

(2) Our "inhibition" is here absolute, in that a single inhibitory signal can block response of a cell to any amount of excitation. We might equally well have used a different system in which a cell fires if the *difference* between the amounts of excitation and inhibition exceeds the

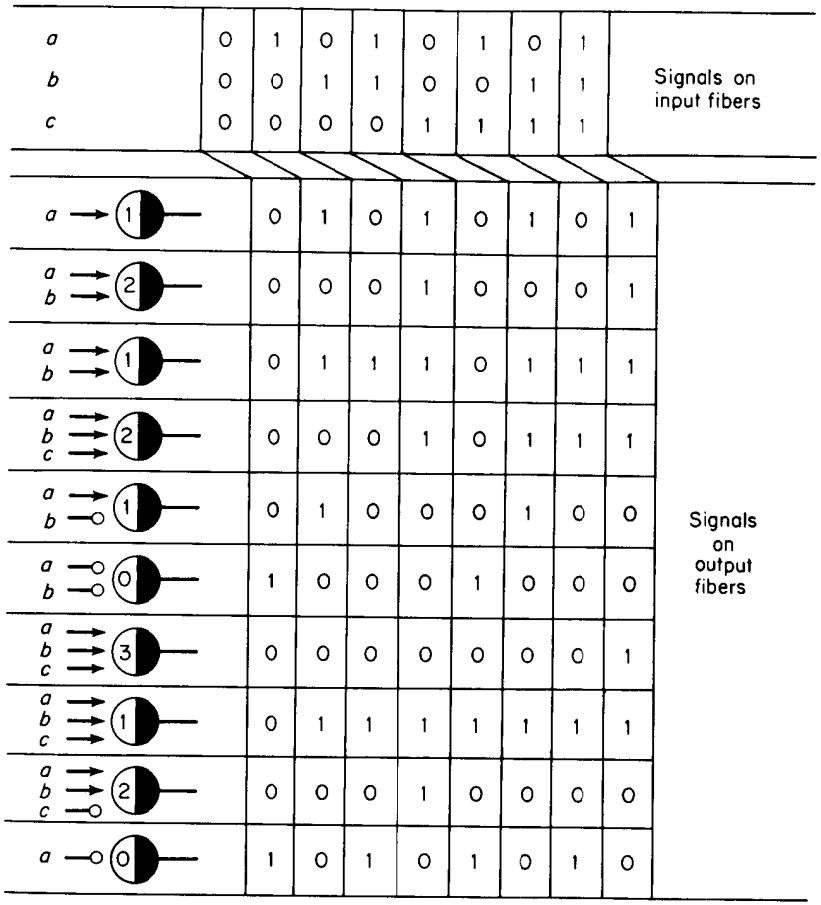


Fig. 3.1-2. Behavior of some cells. (The response columns are displaced to show the always-present delay.)

threshold. This is equivalent to having the inhibition signals increase the threshold. McCulloch, himself, [1960] has adopted this model for certain uses.[2]

(3) *Delays.* We assume a standard delay between input and output for all our cells. In more painstaking analyses, such as those of Burks and Wang [1957] and Copi, Elgot, and Wright [1958], it has been found useful to separate the time-dependency from the other "logical" features of the cells and to introduce special time-delay cells along with instantaneous logical cells. The use of instantaneous logical cells forces one to restrict the ways in which the elements can be connected (lest paradoxical nets be drawn). We will avoid them except briefly in section 4.4.1.

(4) *Mathematical Notations.* In the original McCulloch-Pitts paper [1943], the properties of cells and their interconnections were represented

by a mathematical notation employing expressions from a "temporal propositional calculus." Because most of our arguments can be based on common-sense reasoning about diagrams, it did not seem necessary here to bring in this mathematical apparatus. Of course, anyone who intends to design efficient complex nets or computers will have to learn to use the appropriate modern mathematical representations. For reasons discussed in note 3 of chapter 4, the original McCulloch-Pitts paper is recommended not so much for its notation as for its content, philosophical as well as technical.

3.2 MACHINES MADE UP OF McCULLOCH-PITTS NEURONS

In this section we show how one can go about constructing some useful, more complicated machines, using a few useful kinds of neural nets. Our goal is to collect enough devices to make a general-purpose computer!

3.2.1 Delays

Examples 1, 3, and 4 of section 2.3 give the state-transition structures for machines which remember, and read out, the last one, two, and three binary signals they have received. The network machines in Fig. 3.2-1 do

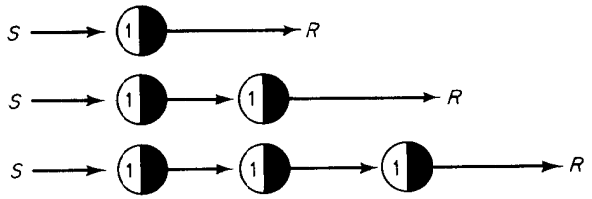


Fig. 3.2-1. Delay nets.

precisely the same things. (Compare with Figs. 2.3-1, -3, and -4.) The behaviors of these nets are described by the equations

$$R(t) = S(t - 1)$$

$$R(t) = S(t - 2)$$

$$R(t) = S(t - 3)$$

respectively.

Note that while it requires 2^n states to remember n digits (see 2.3), it takes only n cells. Of course, the network delay machine with n cells does indeed have 2^n total states. For each cell can have 2 states, and one must

multiply these numbers together for all the cells. Remember precisely why the state diagram (Fig. 2.3-4) of the 3-delay net needs fully $8 = 2 \times 2 \times 2$ states.

This example already shows two reasons why we turn from state diagrams to network machines made of parts, as we start to consider complicated machines. First, the state diagrams become too large, as they grow exponentially with the number of distinct items or signals the machine must remember. Second, the division into *parts* rather than *states* makes the diagrams more meaningful, at least to the extent that different aspects of the behavior are localized in, or accounted for, by different physical parts of the machine.

3.2.2 Gates and switches. Control of flow of information

Suppose that it is desired to control the flow of signals or information from one place to another (Fig. 3.2-2). The information is traveling along

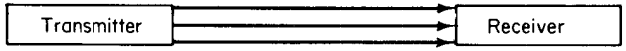


Fig. 3.2-2.

an already established path, and we wish to control its flow by signals under our own control. We introduce (Fig. 3.2-3) cells of threshold 2 ("AND" cells) into each fiber: in the figure we use three parallel fibers just to show that we can handle any number at once. Now, during any interval in which we want information to flow, we can send a string of pulses along the "control" or "gating" fiber. So long as the control pulses are present, the output fibers will carry the same signals that come into the (upper) inputs; if the control pulses are absent there will be no output.

The insertion of the AND cells into the signal path will cause a unit time delay in the transmission of signals. We could avoid this by admit-

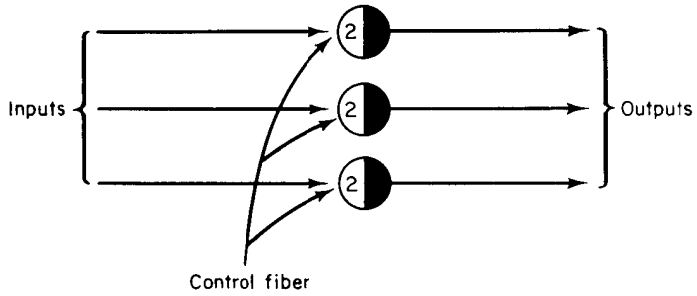


Fig. 3.2-3. Gating network (facilitation type).

ting a gating, or inhibitory, effect with no delay—by somehow interrupting transmission along the fiber. This has been represented (e.g., by McCulloch [1960]) by a termination with one (inhibitory) fiber encircling another, as in Fig. 3.2-4. But it is against our policy here to introduce instantaneously responding elements.

We *can* use the inhibitory effect to control information flow, within the family of cells we are allowed to use, by using a network like that of Fig. 3.2-5. Note that here, information flows through only when the

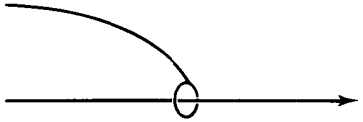


Fig. 3.2-4

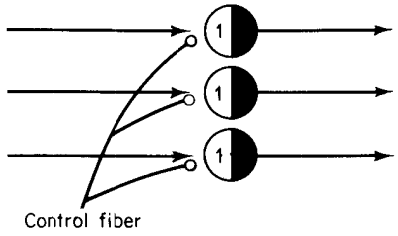


Fig. 3.2-5. Gating network (inhibition type).

control fiber *does not* fire. In the "facilitation" (to use the neurophysiologist's word) gate of Fig. 3.2-3, information flows only when the control fiber *does* fire.

Now consider the network obtained when we use both "inhibition" and "facilitation" gates (Fig. 3.2-6)! When the control fiber is quiet, the signals fire the threshold-1 (OR) cells, and the information flows to

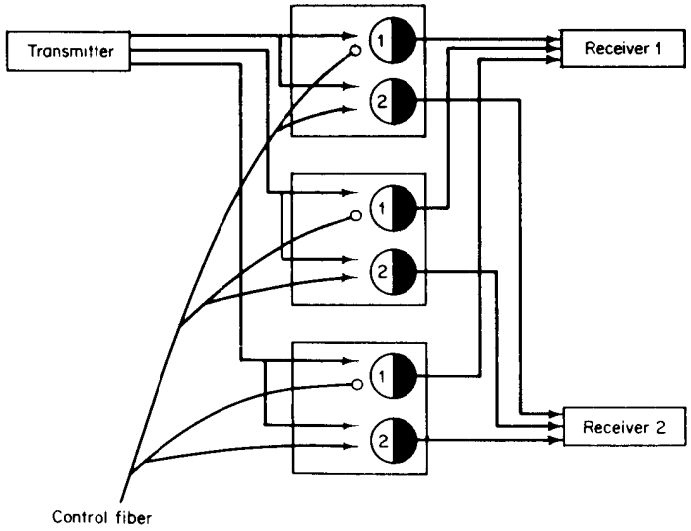


Fig. 3.2-6. An information-route switching device.

Receiver 1. The threshold-2 (AND) cells do not fire. When the control fiber is active, the AND cells transmit the signals to Receiver 2, just as in Fig. 3.2-3. Also, the OR cells are now all "inhibited," and no signals can get to Receiver 1. Thus the activation of the control fiber has the effect of flipping a double-throw (three-pole) switch between the transmitter and the two receivers.

3.2.3 Memory

The control system just described is not very convenient because its operator has to continue to send control pulses throughout the interval during which he wants signal flow. The net in Fig. 3.2-7 permits the operator to initiate signal transmission with a single signal, and to terminate it with another; *no attention is required during the intervening period of transmission.*

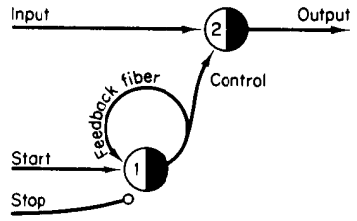


Fig. 3.2-7. Gating network with memory.

The trick here is to use a "feedback" fiber that runs from the output fiber of a cell back to an excitatory termination at *the input of the very same cell. Once this cell has been fired (by a signal from the "start" fiber) it will continue to fire at all successive moments, until it is halted by a signal on the inhibitory "stop" fiber.* Throughout this interval of activity it will send pulses to the "gate" cell and permit the passage of information.

This little net has some memory. It remembers whether it has received a "stop" signal since it last received a "start" signal. We will shortly see how more complex events can be "remembered" or "represented" by the internal state of a net.

It is easy to see that, in our nets, *any* kind of more-or-less permanent memory must depend on the existence of closed or "feedback" paths. Otherwise, in the absence of external stimulation, all activity must soon die out, leaving all cells in the quiet state. This would leave no representation of what had happened in the more remote past. Actually, the same is true even in the presence of external stimuli of any kind, but this is harder to see at this point. The internal state of a net without any "loops" or "cycles" can depend only on the stimulation that has occurred in a bounded portion of the immediate past. The bound is just the length of the longest path in the net. In a net *with* cycles there isn't any bound on the length of possible paths within the net, so there is no limit to the duration of information storage.

Here is another difference between our "cells" and biological neurons. It is evident that in the brain not all information is stored in this "dynamic"

circulating form; it is not known to what extent circulation, or as it is often called, "reverberation," is important.[3]

Although this "circulating" or "dynamic memory is the only way information can be stored in our McCulloch-Pitts nets, we could have started with component parts capable of some kind of "static" memory, e.g., the equivalent of a magnetic-core, switch, or "flip-flop." The nets of the next section show how we can simulate these with our present components.

3.2.4 Binary scalers

Another kind of memory net is the *binary scaler*, which produces *one* output pulse for every *two* input pulses. In many computers, such devices serve as the basis for counting and other arithmetic operations. The net in Fig. 3.2-8 is the simplest one with this behavior.

An initial input pulse will start the memory cell A into cyclic operation. A second pulse at some later time will fire the cell B. The ensuing inhibitory pulse on fiber y will *extinguish* the cyclic activity of cell A. At that same time a pulse will appear on the output fiber of B, indicating that two input pulses have now occurred. The net then returns to its original "resting" state. Hence the "scaler" divides the number of input pulses by a factor of two. Note that Fig. 3.2-7 is contained within Fig. 3.2-8.

Inspection will show that this simple net really won't work as described if input pulses occur too close together, i.e., on successive moments. This may be seen more clearly if we draw the *state diagram* of the machine (Fig. 3.2-9).

The machine has two *parts*—cell A and cell B. Each cell has two states, so there are four possible total states. The two possible inputs to the machine—"pulse" and "no pulse"—are represented by 1 and 0. It just happens that the fourth state never occurs—no arrows of the state diagram lead into it.

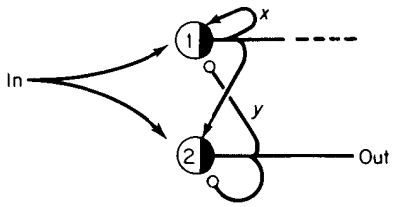


Fig. 3.2-8. Defective binary scaler.

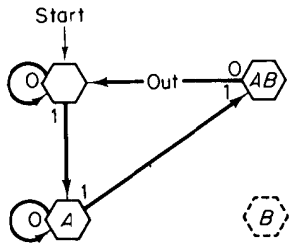


Fig. 3.2-9. State diagram of Fig. 3.2-8.

The names of the states indicate which cells are firing; since the event of "B firing alone" never occurs, we draw it as a ghost. An output occurs only when leaving state AB. The flaw in the device is this: in the transition from state AB to the resting state, the machine ignores its input. It will miss counting a pulse that occurs then. This condition occurs only after a pulse has just arrived (in the transition from ⟨A⟩ to ⟨AB⟩); hence pulses spaced by more than one time unit won't cause difficulty. A slightly more complicated net with three cells is given in Fig. 3.2-10, and it can be seen from its state diagram (Fig. 3.2-11) that it will always work. This net is

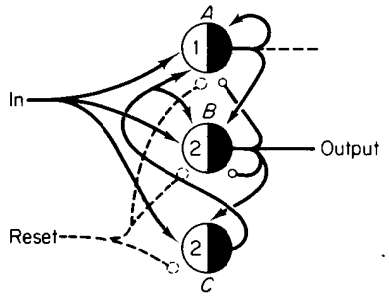


Fig. 3.2-10. Corrected binary scaler.[†]

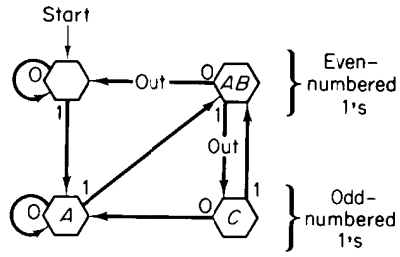


Fig. 3.2-11. State diagram of Fig. 3.2-10.

the same as the first except for an additional cell, C, which recognizes the troublesome condition and corrects it. (There are eight possible states now, only four of which can occur.) To see that this is foolproof, note that odd-number 1's always put the machine in the lower half of the diagram, even-numbered 1's bring it to state ⟨AB⟩, and an output pulse occurs precisely when leaving that state—i.e., after each even-numbered 1.

3.2.5 Counters

The binary scaler of the preceding paragraph could be regarded as a primitive counting device which can count up to two. By similar methods we can make nets which will produce pulses, one for each M pulse inputs, where M is any previously chosen number. Consider what happens if we assemble a number of binary scalers in a sequence—the output of each connected to the input of the next. (See Fig. 3.2-12.) We represent the scalers by boxes, each of which has two inputs—for pulse and reset signals—and two outputs—the fibers coming from cells A and B of the nets just described. Fiber B carries a single pulse for each pair of input pulses.

[†]We have provided an extra "Reset" channel for forcibly returning the device to its resting state; we will use it in the next section.

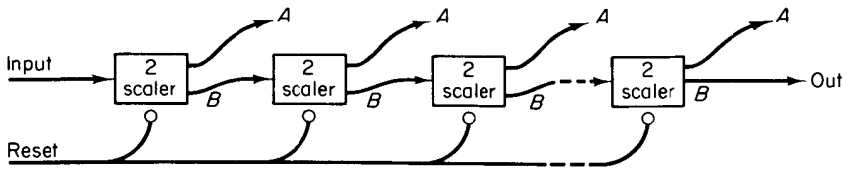


Fig. 3.2-12. 2^K-ary scaler.

Fiber A is quiet after even pulses, but (usually) *fires steadily after odd pulses.*† If we string together K such scalers, we obtain a net which can count up to 2^K and then start over. If we fire the common reset fiber we can restart the net at any time.

How can we count up to a number which isn't exactly a power of 2? We can use a trick which involves the binary number system representation of that number. Take, for example $M = 13$. Then $M = 8 + 4 + 1 = 1101$ in binary form. Let T be the number of 1's in the binary number. We create a single cell with threshold T (3 in this case) and run excitatory connections to this cell from the steady (A) outputs of those scalers which correspond to the positions of the 1's of the binary form of M. (See Fig. 3.2-13.) Now, generally speaking, the set of A fibers which

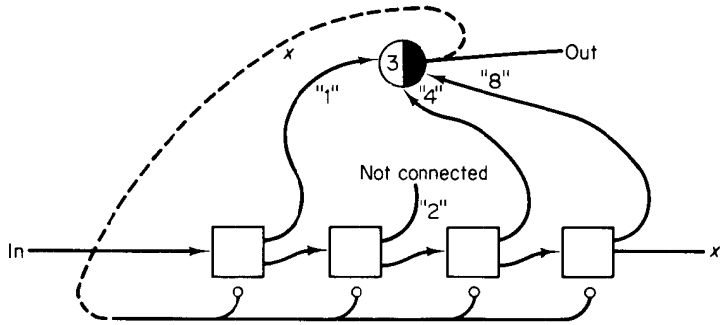


Fig. 3.2-13. M-ary scaler for $13 = 8 + 4 + 1$.

will be active after the Mth input pulse will be just those in the positions for 1's in the binary expansion of M. And the new cell with threshold T will fire when first all these positions become active together, i.e., on the Mth count. The path labeled with an x will cause the reset line to fire when this event occurs so that the counter will start over after each group of M pulses. With this connection in operation, the whole net becomes an "M-ary scaler."

†See problem 3.2-2, p. 44.

PROBLEM. Why don't we need to run inhibitory connections from the other scaler stages to the output cell?

We run again into timing problems if pulses enter the net too rapidly. Correct operation is certainly assured if pulses are spaced by intervals of K moments, where K is the number of binary scalers in the chain. Otherwise there is a chance that the top cell will miss its chance to fire. The trouble is due to the fact that there are transient states during which the output of the chain doesn't match the binary expansion of the number of inputs that has occurred. These states occur while the net is "propagating the carries." Happily, these transient states never correspond to numbers which will cause spurious firing of the top cell (as the reader may care to verify), so the top cell won't fire when it shouldn't. But the total count might pass through M while the carrying is going on and the top cell could miss that count. We can prevent this by spacing the inputs so that the net has a chance to "settle down" between counts.

EXAMPLE

Consider a three-stage counter like that in Fig. 3.2-12. Table 3.2-1 shows the firing pattern in response to the input sequence given in the top line. "Number" is the binary number $A_1 + 2A_2 + 4A_3$ that appears

Table 3.2-1

Signal	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	...
A_1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1 ...
B_1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1 0 ...
A_2	0	0	0	0	1	1	1	1	0	0	0	1	1	1	1	1 0 ...
B_2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1 0 0 ...
A_3	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1 0 0 ...
B_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 0 1 0 0 0 ...
Number	0	1	1	1	2	3	3	3	2	5	5	5	6	7	7	7
Should be	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8

currently at the A output fibers. Note that the count '4' never appeared in the A output fibers; this is because we did not space our 1's far enough apart (since $k = 3$) and new inputs came in while the net was "carrying" from 3 to 4. Nevertheless, the network as a whole never really lost count, and it catches up soon.

Another kind of counter which is conceptually much simpler, but also less efficient in its use of cells, is shown in Fig. 3.2-14. This counter might be called *unary* rather than *binary*—it corresponds to counting on fingers without a "radix" or power-based number system. In this net the activity

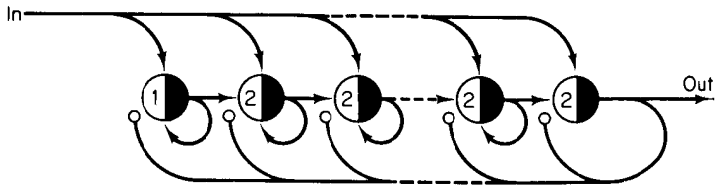


Fig. 3.2-14. Unary scaler.

advances, with each input pulse, one cell further down the chain. When it reaches the end of the chain it causes an output pulse and resets the whole system to the resting state. (At this moment it is capable of missing a count. This defect, as usual, could be fixed by adding extra cells.) To count up to M now requires M cells instead of $\log_2(M)$, which is the very much smaller number required by the binary scaler chain. To appreciate this the reader will recall that $\log_2(1000)$ is about 10 and $\log_2(1{,}000{,}000)$ is about 20, etc. Recall the remarks in section 3.2.1. Here we are using one whole cell for each *state* of the binary machine.

PROBLEM 3.2-1. Verify the statement that the M-ary scaler of this section cannot yield spurious outputs because of carry-propagation conditions, although it can miss proper outputs. This is rather complicated.

PROBLEM 3.2-2. The statement that fiber A in Fig. 3.2-12 fires steadily after odd pulses is false because state $\langle C \rangle$ does not yield an output on fiber A_1. Can this be corrected without increasing the over-all delay of the outputs? This defect need not affect the construction of Fig. 3.2-13 because one can run fibers from cell C as well as from cell A to the M-detector cell.

PROBLEM 3.2-3. Design a binary M-ary scaler that does not miss counts under any circumstances. This is very complicated. (Note that even the non-binary scaler of Fig. 3.2-14 can miss an input when it resets. Fix it!) It is clear that the difficulties vanish if we can be sure that signals are separated enough. The troubles arise from attempting to do complex operations in times of the order of the response delays of the individual parts. It is remarkable how far computer designers have been able to go, in getting around the carry-propagation delays one might suppose to be inescapable in arithmetic devices. For a review of some of these techniques, see several papers in Proc. Inst. Radio Engrs. [Jan., 1961].

3.2.6 Nets to do arithmetic

It is interesting that only a very simple net is required to add two arbitrarily large numbers presented in a binary serial form. The state diagram for such a machine was pictured at the end of section 2.3 (Fig. 2.3-5). (Note: Unless the summands are presented with the least significant digit first, no finite machine could do this job. For the value of each digit of the sum depends functionally on *all* the input digits of lower order. The highest digits can't be computed until all the others are available and would require unlimited memory if arbitrarily large numbers were to be added, high digits first.)

Let A_i and B_i be the ith digits of the summands.† Let S_i be the ith digit of their sum. Then S_0 is the "mod(2)" sum of A_0 and B_0—that is, it is 1 or 0 as $(A_0 + B_0)$ is odd or even. But, just as in decimal addition, the higher digits of the sum depend not only on the corresponding digits of the summands but also on whether or not there was a carry from the previous state. Let C_i be 1, or 0, according to whether there is, or is not, a carry from the $(i - 1)$-th stage. Of course, C_0 is 0.

Then the rules for (binary) addition are

$$S_i = \begin{cases} 1 & \text{if } (A_i + B_i + C_i) \text{ is odd} \\ 0 & \text{if } (A_i + B_i + C_i) \text{ is even,} \end{cases}$$

and

$$C_{i+1} = \begin{cases} 1 & \text{if } (A_i + B_i + C_i) \text{ is 2 or greater} \\ 0 & \text{if } (A_i + B_i + C_i) \text{ is 1 or less.} \end{cases}$$

Note the complete symmetry of the roles of A_i, B_i, and C_i in these rules!

The net in Fig. 3.2-15 realizes both these calculations with just six cells; the output is delayed by three time units. Note how the firing of the threshold-2 cell represents a carry to the following digit and how the input digits and the carry signal are treated symmetrically. (But note also that here we have a situation in which the state diagram is somewhat simpler than the best McCulloch-Pitts net we could invent. Here is a machine that doesn't divide so well into parts!)

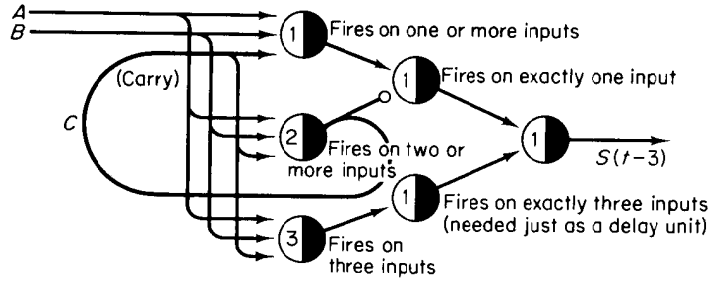


Fig. 3.2-15. A serial binary addition network.

†That is, A_0 is the least significant digit.

If we use "subtractive inhibition"[2] so that an inhibitory signal adds 1 to the threshold, we can use the elegant net of Fig. 3.2-16, which shows how engineering economies can come through the use of the so-called "threshold logic" of subtraction.

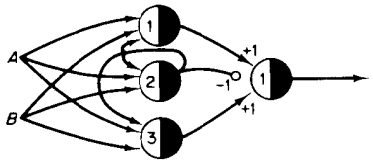


Fig. 3.2-16

Addition in number systems other than binary is somewhat more complicated but is the same in principle. Multiplication can be done by successive addition, as it is in desk calculators. But note that in multiplication the dependency on earlier calculations *can't* be expressed by a single carry digit; one must somehow provide storage space for the accumulation of the partial sums. The required storage space must increase relentlessly with multiplier size so that, as we observed in section 2.5, we are restricted to a limited number range for any given machine. Subtraction and division are roughly analogous to addition and multiplication respectively, although division involves some further complications.

PROBLEM 3.2-4. Given that one of two numbers is bound in size, show that a finite machine can be built (for each bound) which will multiply any number so bounded by any arbitrarily large number presented digit-by-digit. Show that if N is a bound on the number of digits, then each digit of the product can be computed by remembering not many more than N digits from the past. Exactly how many digits of memory are really required?

3.3 DECODERS AND ENCODERS FOR BINARY SIGNALS. SERIES-PARALLEL CONVERSION

As we have noted (in section 3.2.5), the total number of states of a machine is the product of the numbers of states of all of its parts. It is the huge numbers of states so obtained that makes practical the digital computer, and other interesting machines. If one had to represent each state of a process by the excitation of a different part of a machine, one would be limited in practice to rather simple processes. But by using different *configurations* of a relatively few parts to represent states of a process we can do much better.

This means that in efficient machines, information must be represented in something like binary-coded form. (In chapter 6 we will discuss—for theoretical reasons only—some machines which use the unary form, and it will be seen how inefficient they are.) The states of the process carried out by the machine will then be represented by *configurations* of activity

in *groups* of cells or parts, rather than by the activity of single parts. Many different process states can be represented by different activity configurations in the same group of cells. Eventually, however, it will usually be necessary to *detect* some such configurations and represent them as distinct signals—i.e., as impulses along some distinct output fibers. For this and other reasons, we have to show that signals can be converted freely between the unary- and binary-coded forms.

Another way of making efficient use of machine total states is by encoding information in the form of time-sequential configurations. That is, we can represent a process state by a time configuration of activity on a single cell, or a small group of cells. And again, it will be necessary to be able to decode such configurations—to convert them to and from single pulses on particular fibers. The next few sections show various techniques for converting information from one form to another; and these techniques are then brought together to give a powerful general technique for design of arbitrary finite-state machines.

3.3.1 Decoding a binary time sequence

Suppose that along a given fiber there may appear, from time to time, various sequences of pulses. We want to distinguish between the different sequences by activating a different output for each sequence. We assume that there are just a certain finite number of such patterns, and take N to be the duration of the longest such sequence. A set of sequences might be, for example:

```
****  ***1  **1*  **11  *1**  *1*1  *11*  *111
1***  1**1  1*1*  1*11  11**  11*1  111*  1111
```

where '1' signifies the presence of a pulse and '*' the absence of a pulse at the corresponding moment of time. Here we have chosen as our example the full set of sixteen possible sequences that may occur over an (N = 4)-moment interval. (One might not necessarily want to distinguish between all of these.)

We have to suppose also that there is available some special signal, called the *start pulse*, which indicates when the sequence transmission begins. Otherwise we could not hope to be able to distinguish between, e.g., ***1, **1*, *1**, and 1***, all of which are represented by a single pulse, and differ only in what is chosen to be the time-origin of the transmitted sequence. Given signals coming in on an *input fiber* and a *start pulse* coming in on a *start fiber*, our decoding problem is solved by the network of cells in Fig. 3.3-1.

Each rectangle in the diagram contains a copy of the same two-cell

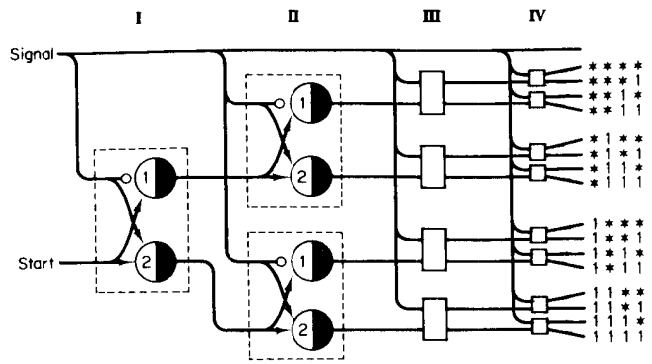


Fig. 3.3-1. Binary sequence decoder.

network seen to the left.† It can be seen that nothing can happen in this net until a start pulse enters. If the start pulse is not accompanied by an input pulse, the upper cell in stage I will fire, while if the start pulse is accompanied by an input pulse the lower cell will fire (and the upper cell will not). In either case one and only one of the two stage I cells will fire.

At the next moment, the pulse output from stage I will serve as a start pulse for one section of stage II. The other half of stage II will remain quiescent. (By repeating this argument we see that the first element of the input sequence has already determined which half of the total output bundle will have an output.) A similar selection occurs in stage II; after the second moment there will be a pulse on exactly one of the four fibers bridging stages II and III. And after K moments there will be just one pulse on the 2^K fibers between the Kth and $(K + 1)$-th stages. At the end, there will be just one output fiber excited, and this will be the fiber cor-

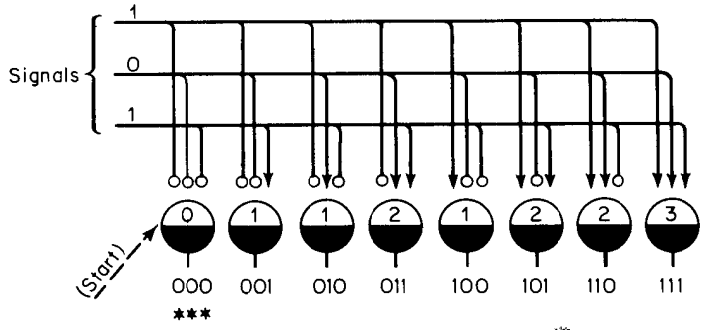


Fig. 3.3-2. Three-digit binary parallel decoder.

†These are the same "switches" used in section 3.2.2, but note the reversal of input-connection significance.

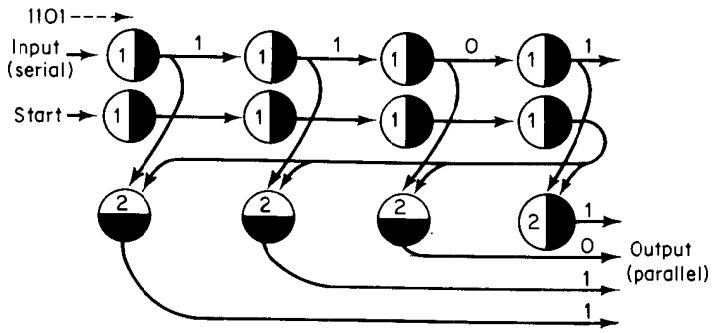


Fig. 3.3-3. Four-digit serial-to-parallel converter.

responding to the binary number represented by the input sequence. Thus each distinct serial pulse sequence excites a distinct output fiber.

We will see how to construct a different kind of binary sequence decoder by using the nets of the next section. (Specifically, one connects the output of a net like Fig. 3.3-3 to the input of a net like Fig. 3.3-2, matching, of course, the numbers of digits.) If one counts the numbers of cells, this alternative method might seem more efficient than the present one. But if one also considers the *cost of connections*, and that high-threshold cells will presumably be expensive, one sees that the binary net of Fig. 3.3-1 will have advantages for large N.

3.3.2 Other decoders

If the coded information comes in the form of single pulses occurring *simultaneously* along a number of parallel fibers, we can use the net below. Here we utilize high-threshold cells with many connections.

Figure 3.3-2 shows the net for $N = 3$. The threshold of each cell has to equal its number of excitatory connections to prevent firing on a signal containing fewer 1's than required. The cell at the left has a threshold of 0; it fires unless inhibited. (Unless such cells are used, the "null" signal

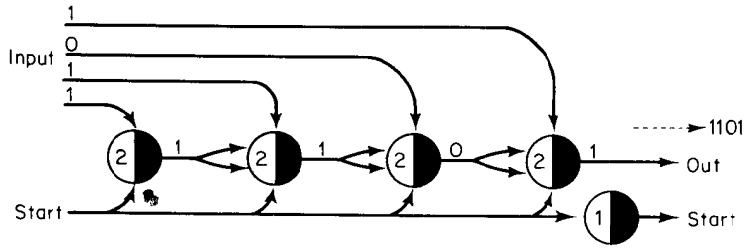


Fig. 3.3-4. Parallel-to-serial converter. Note that it needs only one cell per stage.

can't be distinguished actively, unless some other source of signals, like the "start" fiber shown, is provided.)

The serial and parallel modes of operation are, broadly speaking, interchangeable. The nets in Figs. 3.3-3 and 3.3-4 perform conversions in each direction.

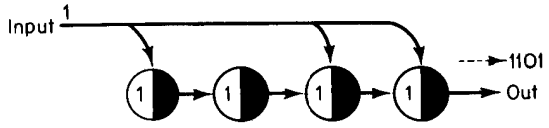


Fig. 3.3-5. Serial encoder.

3.3.3 Encoders

Encoders are trivial for either direction. To obtain the serial signal from a single pulse, for instance, the net in Fig. 3.3-5 will suffice. And for the parallel case one needs only the proper branching, as in Fig. 3.3-6.

Fig. 3.3-6. Parallel encoder.

By combining these two encoding techniques we can construct more complicated input-output relations. Suppose that whenever a pulse occurs on some input fiber f_1, we want to generate a certain time-sequence firing pattern on some output fibers $g_1, \ldots, g_n$. For example, if f_1 fires at time t, we might want g_1 to fire at $t + 2$ *and* at $t + 3$, and g_2 to fire at $t + 1$ *and* $t + 3$. Suppose also that when a pulse occurs at time t on another input fiber f_2 we want g_1 to fire at $t + 1$ *and* at $t + 2$, and g_2 to fire at $t + 2$ *and* at $t + 3$. We can represent these specifications by

$f_1(t)$:	$t+1$	$t+2$	$t+3$
g_1		x	x
g_2	x		x

$f_2(t)$:	$t+1$	$t+2$	$t+3$
g_1	y	y	
g_2		y	y

The net in Fig. 3.3-7 realizes these specifications and shows clearly a general method for making such nets. The relation between the response-specification table and the net connections is seen more easily by drawing the net with signals propagating from right to left. We can extend this method to handle any number of input and output fibers, and responses of any duration, by extending the diagram in appropriate dimensions.

If we measure the cost of a machine only by its number of cells, such nets are very economical. It takes just n cells to realize, in one net, 2^n different firing patterns. However, if one counts the number of connections into the cost, the cost rises more or less in proportion to the number of patterns to be realized.

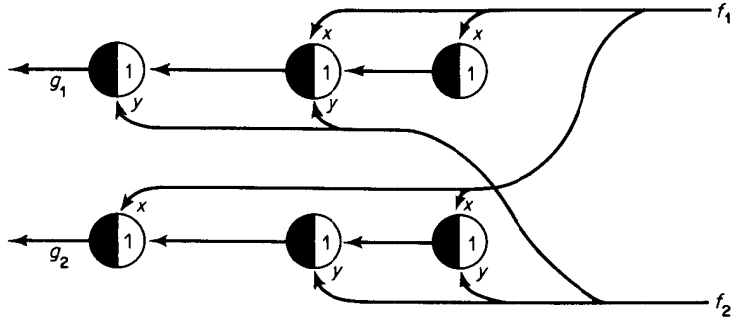


Fig. 3.3-7. Firing pattern encoder.

In a real physical system there are always limitations on the numbers of connections that can be brought together in one region. Basically, the limitation is one of signal-vs.-noise levels. *In practice one must make sure that there is not too great a "load" on a source which "fans out" to drive many other elements.* Otherwise the signal level may drop below the noise level inevitable in any receiver device, and the channel will become unreliable. *There are similar constraints on the amount of "fanning in" allowed.* There are other problems in the construction of practical threshold elements—problems of resolution vs. stability—and at this writing computers do not often use elements analogous to McCulloch-Pitts neurons with high thresholds. As we will see in section 3.6, everything we want to do can be done with a fixed bound on the degree of "fanning" permitted, but this involves inescapable delays—slowing down computation speeds.[4]

PROBLEM. Characterize the behavior of the above encoder when signals arrive closely together along the input lines, so that the response patterns are forced to overlap. How does this behavior depend on the fact that all cells have threshold 1 and there are no inhibitors?

3.4 REALIZATION OF MORE COMPLEX STIMULUS-RESPONSE SPECIFICATIONS. THE BEHAVIOR OF NETS WITHOUT CYCLES

One might want more elaborate stimulus-response behavior in which *stimuli*, as well as responses, are time patterns along a number of fibers. We can achieve this by combining several of the techniques of section 3.3. We will do this in such a way that the resulting nets will have the peculiar property that they contain no feedback—circular signal paths—no routes

along which a pulse can return to the same cell again and again. It is of interest to see precisely what such nets can do and what they cannot do, for these are exactly the machines without any long-term memory.

The method can be shown best by example. Suppose that the situation is like that in section 3.3.3 except that the stimuli are temporal patterns along several fibers $e_1, \ldots, e_m$. We might, for example, want two stimulus response pairs

		t	$t+1$
S_1	e_1		x
	e_2	x	
	e_3	x	x

should yield

	t	$t+1$	$t+2$
g_1		x	x
g_2	x		x

R_1

and

		t	$t+1$
S_2	e_1	x	
	e_2	x	x
	e_3		x

should yield

	t	$t+1$	$t+2$
g_1	x	x	
g_2		x	x

R_2

Of course, we cannot ask for the response to begin until after the stimulus ends; there must be some delay. The net of Fig. 3.4-1 realizes these specifications with a delay of one additional moment. Note how the tables above are "wired" into the net.

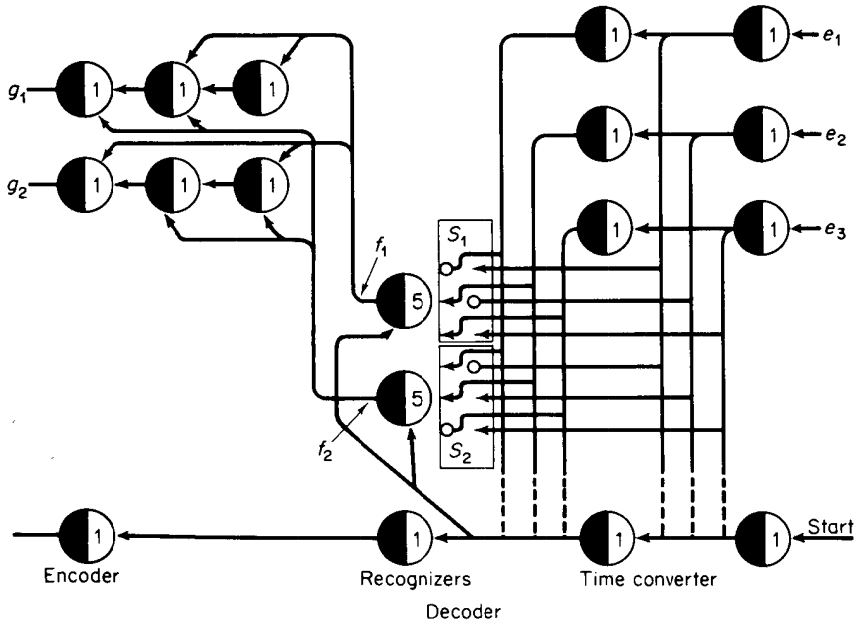


Fig. 3.4-1. Stimulus-response recoder.

Our "re-coder" net has three parts. In the part to the right, the space-time input patterns are converted by a simple series-parallel converter into purely spatial patterns. This reduces the problem to one of recognizing a purely spatial distribution of signals. In the middle of the net is a set of "recognizing" cells, one for each of the specified stimuli. Their input connections and thresholds are arranged so that each recognizing cell will fire on precisely one of the stimulus patterns to be detected. The start pulse is transmitted to the recognizing cells with the proper delay so that they work only when the entire stimulus has arrived. Note that the output (encoder) part is the same as Fig. 3.3-7.

The output fibers from the recognizing cells produce the required response patterns exactly as in section 3.3.3. The technique is easily extended to handle more input and output channels, more S-R pairs, and longer pattern durations. Note that this system requires only one recognizing cell for each desired S-R pair. Of course, there are further expenses in the many connections associated with each such cell and in the increased numbers of connections to the other cells.

3.4.1 Delays. Inefficiency of the canonical method

There are certain inescapable delays in any computation, due to the fact that certain events have to occur before other events. In general, a lag of two time units (and no more) is required for an arbitrary finite table of stimulus-response requirements. This is shown by our construction above. Roughly speaking, one needs a level (delay 1) for the recognizing cells or their equivalent, and one needs another output level (another delay) to provide for the bringing together of the outputs from different recognizing cells. Of course, delay 2 is not always necessary, and it is sometimes possible to realize the desired behavior with delay 1 or even delay 0 or -1, etc. (This can happen if, as in the present example, one can distinguish the stimulus patterns before they are completed.) The net in Fig. 3.4-2 has, in this sense, delay 0, since the output begins concurrently with the last (second) volley of the stimulus. This net happens to have the same two stimulus-response reactions as does the net of Fig. 3.4-1 (but will react differently to other stimuli). It turns out that the information on fiber e_2 was redundant; so we don't need it.

The stimulus-response relation used above for our examples was not chosen arbitrarily but was designed also to illustrate the possibility of exchanging the roles of *time* and *location* in a fiber bundle. The two stimulus-response pairs given explicitly are instances of a set of sixty-four relations which have the property that if input e_i *fires at* j moments after the input start pulse, then output g_j *will fire at* i moments (plus the system delay) after the output start pulse. This whole set of relations can be

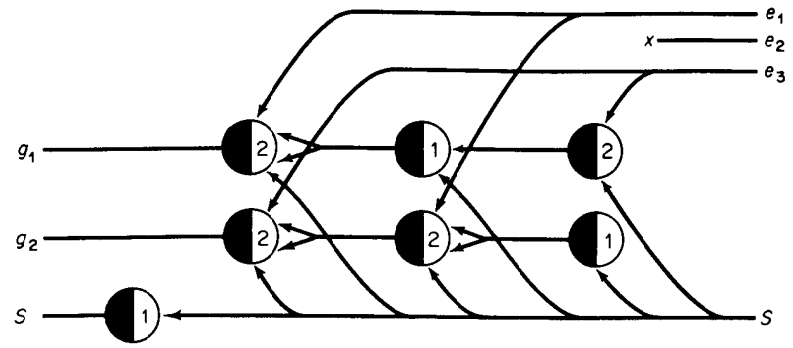


Fig. 3.4-2

realized by a simple, uniform method, illustrated by the net in Fig. 3.4-3, constructed for $i = 1, 2, 3$ and $j = 1, 2$.

If we had to use for this transformation the method used in the "canonical" encoder-decoder system built along the lines of Fig. 3.4-1, it would require sixty-four recognizer cells. But because the transformation has such a simple structure, we were able to realize it with far fewer than that number of cells. To realize a completely arbitrary stimulus-response relation, where the stimuli have up to N components (here $N = 6$), requires on the order of 2^N cells, on the average. Only a small proportion of behavior functions can be realized with appreciably smaller nets. But there is reason to believe that this small proportion contains almost all of the functions that are ever likely to interest us.

> The simplicity of the construction of Fig. 3.4-1 will be outweighed by its dreadful inefficiency, for most interesting applications. The general method uses what a mathematician would call a "reduction to canonical form"—in which every problem is transformed into the same "general" form. This form is conceptually simple but likely to be impracticably cumbersome.

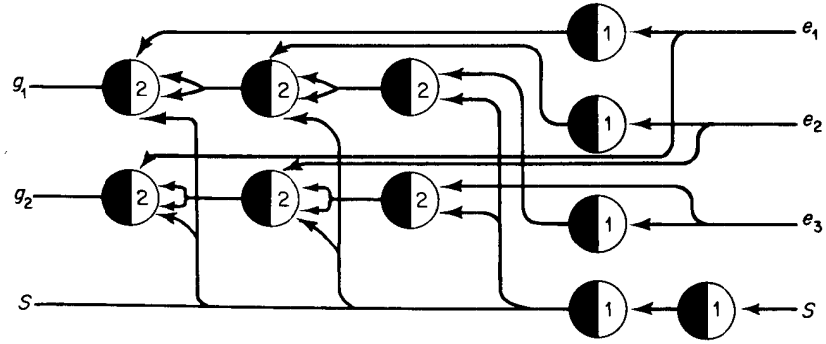


Fig. 3.4-3. Interchange of space and time.

> *The technique does not exploit the structural features of individual problems. But "interesting" functions usually have features that permit enormous reductions in the sizes of the nets required to realize them.* It is difficult to imagine how this statement, which turns around the subjective term "interesting," could be proven in some acceptable mathematical form, but we believe that it expresses a profoundly important truth.[5]

3.4.2 Nets without cycles

The machines (nets) of sections 3.4 and 3.4.1 are all *cycle-free*; that is, they contain no circular signal paths. We have seen that, even with this restriction, we can realize any stimulus-response behavior function that is duration-bounded in the manner indicated in our discussion. These are in fact the only behavior patterns that can be realized by nets without cycles.

> One should not conclude that the behavior patterns so realized are necessarily uninteresting or unimportant. We must admit that, at least in principle, one could so build a machine which would behave in every respect like some particular man, for the lifetime of that man. That is, one could (in principle) arrange things so that for each sequence of stimuli the "man" might ever receive, his proper response would be elicited. But in so far as practical matters are concerned, it is useless and even misleading to point this out since (1) the sheer bulk of such a machine would be unrealizable in our universe even with unimaginable improvements in technology and (2) one would not know what to build into such a monstrous structure without a complete, extensive theory of how the man operates. And if we had the latter, we could build a smaller, genuinely intelligent machine.

3.5 THE EQUIVALENCE OF NEURAL NETS WITH FINITE-STATE MACHINES IN GENERAL

It is evident that each neural network of the kind we have been considering is a finite-state machine. At any moment, the total state of the net is given by the firing pattern of its cells. The state-transition function $Q(t + 1) = G(Q(t), S(t))$ is determined by the connection structure of the net, and the output function $R(t + 1) = F(Q(t), S(t))$ is determined by which fibers are designated as carrying output signals.

It is interesting and even surprising that there is a converse to this.

THEOREM 3.5

Every finite-state machine is equivalent to, and can be "simulated" by, some neural net. That is, given any finite-state machine M, we can build a certain neural net N^M which, regarded as a black-box machine, will behave precisely like M!

The construction of the equivalent net-machine N^M is surprisingly straightforward, once we agree on how the inputs and outputs are to be represented. Suppose that the inputs and outputs to M are $S_1, \ldots, S_m$ and $R_1, \ldots, R_n$ respectively, and that the states of M are $Q_1, \ldots, Q_p$. Our net N^M will then have input fibers $s_1, \ldots, s_m$ and output fibers $r_1, \ldots, r_n$, and it will contain m cells, $C_{1j}, \ldots, C_{mj}$, for each state Q_j of the machine M. The cells C_{ij} are to be arranged in a two-dimensional $(m \times p)$ array, as depicted in Fig. 3.5-1.

Each of the cells C_{ij} has threshold 2. We exploit the ability of such cells to detect coincidences. In fact the cell C_{ij} will fire (at time $t + 1$) precisely when it receives an input from S_i (at time t) *and* when the simulated machine M is in state Q_j. Thus the firing of C_{ij} is to be equivalent to the pair of events (Q_j, S_i). To make the net equivalent to the machine M we now have only to arrange things so that it produces the proper output $F(Q_i, S_j)$ and goes into the appropriate internal state $G(Q_i, S_j)$. To see how this is done, consider the vertical columns of the diagram.

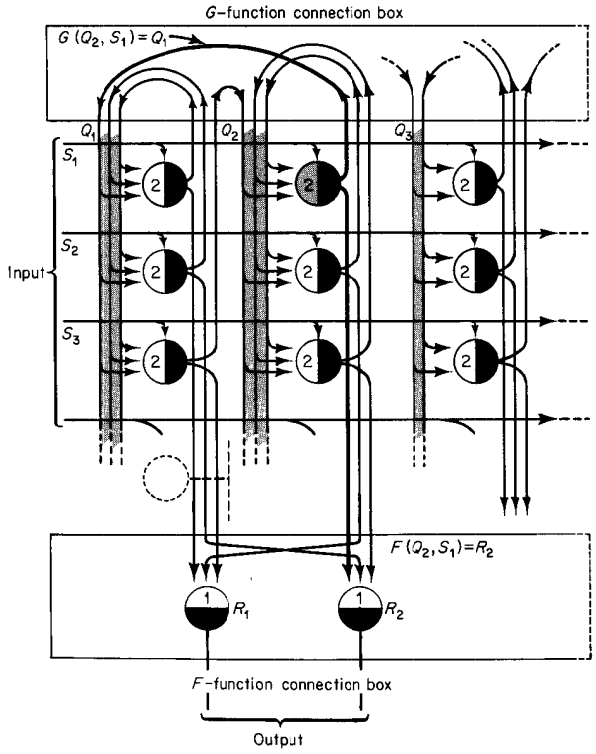


Fig. 3.5-1. The canonical finite-state neural net.

At any moment t there will be, among all the descending fibers in all the shaded columns, precisely *one* active fiber. Suppose that this fiber is in the jth column. Then the net N^M is simulating the state Q_j of M. (It doesn't matter which fiber of the column is active, since all have the same connections.) Suppose also that at each moment precisely one horizontal (input) fiber is active, say, the ith row fiber. This corresponds to some input signal S_i. Then precisely one of the array cells will fire at time $t + 1$; this will be cell C_{ij}.

The fiber from C_{ij} divides into two branches. The descending branch goes to the "F-function connection box" and there leads to the cell which represents the appropriate *output* $F(Q_j, S_i)$. Thus the function F is "wired" into this part of the net, and it need concern us no further.

The ascending branch of the output from C_{ij} is responsible for the machine's change of state. This fiber goes up through the "G-function connection box" which is so wired that this fiber enters the descending column for the appropriate new state $(G(Q_j, S_i)$. If one and only one cell C_{ij} fires at time $t + 1$, we are thus assured that at time $t + 1$ there will again be precisely one active fiber among the descending columns (and that it will be in the column corresponding to the next state of the simulated machine M). Thus, if the signals entering N^M are at each moment the same as those entering M, both machines will go through the same sequences of states and outputs! (The output of N^M will be delayed by one time unit because of the OR cells in the output connection box.) This completes the proof of the theorem.

EXAMPLE

The particular connections shown in the diagram Fig. 3.5-1 show the result of applying the general method to a serial binary adder, e.g., the one described in section 2.3. S_1 corresponds to the 00 input, S_2 to the 01 and 10 inputs and S_3 to the 11 input case. Q_1 is the no-carry state and Q_2 is the carry state. One has to get the machine started by introducing a start pulse into some descending fiber; this selects the initial state of the net. To make the machine into a conventional two-input serial adder one has to attach an input encoder of some sort, such as the one in Fig. 3.5-2.

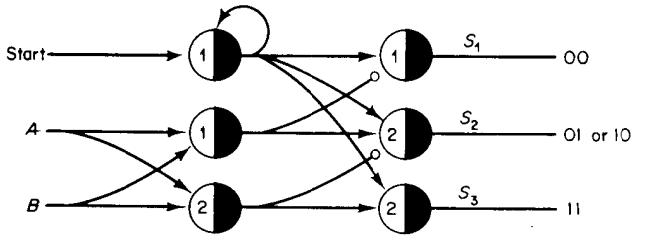


Fig. 3.5-2. Input encoder for binary adder.

The "canonical net" for finite state machines has the interesting property that it is composed essentially of nothing more than AND cells. (The reader will agree that the OR cells in the output connection box are not really involved in the basic operation of the machine.) It follows that, in some sense, these are all we need. We will look into this more carefully in section 3.6. In particular, it would seem that there is no need for inhibitory connections—for none of these appear in the diagram! The secret lies in our assumption that the input signals are in "completely decoded" form: at each moment one and only one of the input lines is excited. We now know that if this is the case we need only threshold-2 cells to simulate arbitrary machines. However, if the input signals don't have this convenient form, then, in general, it will require inhibitory connections (as in Fig. 3.5-2) to bring them into that form. But it still seems quite remarkable that only the threshold-2 cells are required once the signals are so decoded.

PROBLEM. Prove that one cannot, in general, decode signals using only AND and OR cells (without inhibition).

Efficiency

It is important to recognize that we have simulated the behavior of the given machine M by a rather inefficient mechanism. If there are I signals and J states, our construction uses $I \times J$ cells (plus the number used in encoders and decoders). If these cells could take on all possible firing configurations the machine N^M would have 2^{IJ} possible states. Only a very small fraction of these occur in the operation of our machine. See the remarks in section 3.4.1. This sort of result reduces the applicability of many powerful conceptual tools of logic and analysis to the practical design of actual machines. Those methods transform away the special features of devices or expressions by loading them down with terms which do not add to their content. This improves their "form." But it usually works against efficient design.

There is also a great deal of theory and practical knowledge concerned with the *efficient* realization of behavior patterns (switching functions) by nets of various kinds of elements—relays, transistors, vacuum tubes, and the like; but these practical matters are off the main line of our study.

3.6 UNIVERSAL SETS OF CELLS

Up to this point we have constructed our nets by freely connecting together elements of the form shown in Fig. 3.6-1, where we set no limits

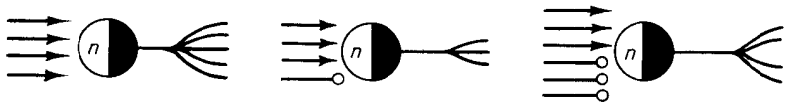


Fig. 3.6-1

on allowed values of the threshold n, nor did we put any limit on the numbers of connections to and from any one cell. It will be noticed, however, that in most of the nets exhibited, the cells used thresholds of 1 and 2. Indeed we might suspect, from the results in sections 3.3 and 3.5 that we do not need thresholds larger than this. This is true, as we will see below. But although higher thresholds are not basically necessary, they make it possible to perform computations with fewer cells and less delay. Consider the parallel decoder of Fig. 3.3-2. If there are n input fibers, this decoder uses cells with threshold up to and including n. But the decoding delay is only 1 time unit. We could construct a parallel decoder out of threshold 1 and 2 cells, but the delays have to be longer; at least $\log_2(n)$.

PROBLEM. Construct parallel decoders out of threshold-1 and -2 cells. Construct the same using only threshold-2 cells.

It is interesting that if we are willing to pay the price in delay and numbers of cells, *we can simultaneously restrict both the threshold number and the number of connections to and from each cell.* We use a technique of branching binary trees. To see the method, suppose that we wish to construct nets equivalent to the elements in Fig. 3.6-2, which are typical of the kind we have been using freely. These cells can be replaced by the left and right nets of Fig. 3.6-3 respectively. In the net to the right, we had to load the diagram with some extra OR cells so that the net will have the same delay for all signals.

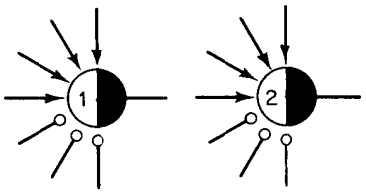


Fig. 3.6-2

The numbers of input and output connections, as well as the threshold numbers, are held to 2 in these nets. This means that we need not be concerned with problems of "loading" and noise, provided again that we do not mind the general slowing down of the operation of the system. And it means that our stockroom of parts needs to contain nothing more than copies of the three elements we have used in Fig. 3.6-3, shown individually in Fig. 3.6-4.

Hence this set of three elements forms a "universal base" for the construction of finite machines. It turns out that, at the cost of a delay, the second (AND) element can be omitted, for it can be replaced by Fig. 3.6-5 so that the first (OR) and third (AND NOT) elements alone could form a universal base. Indeed, these two elements can easily be combined to form a single device (Fig. 3.6-6) which forms, by itself, a universal base! (It is understood that one can leave some fibers unconnected, when necessary.) To use this as a base, one has to compensate for the fact that the

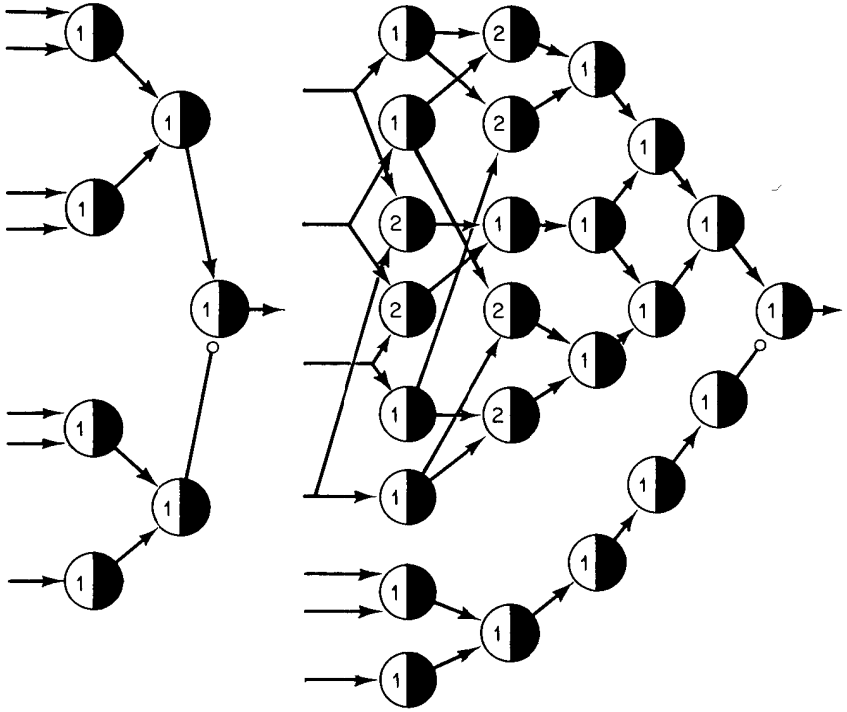


Fig. 3.6-3. OR and AND nets with limited numbers of connections on each cell.

net in Fig. 3.6-5 for forming AND from three AND NOT's takes two moments to respond. This cannot be avoided, so the element is really universal only for systems in which signals enter only on alternate moments. The same remarks hold for several of the bases discussed below.

Other simple combination bases exist. Let us permit a cell to have a

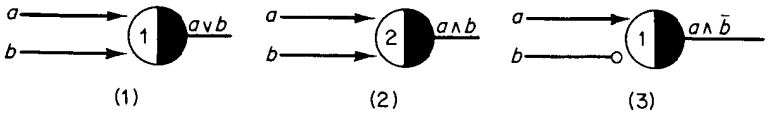


Fig. 3.6-4. (1) a OR b, (2) a AND b, (3) a AND NOT b.

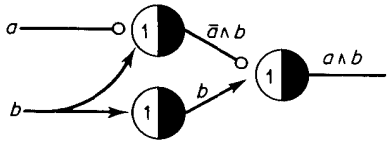


Fig. 3.6-5

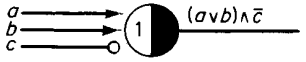


Fig. 3.6-6. OR AND NOT.

threshold of 0, so that *it will fire unless actively inhibited*. Then the element NOR in Fig. 3.6-7 is universal when combined with a simple delay element. To see this, observe (Fig. 3.6-8) that we can get all the functions above (in the alternate-moment mode). There is another two-input element, NOT BOTH, which is universal when combined with delay. It is, in effect, a minus-one threshold cell which fires unless inhibited by *both* of two inhibitory connections. (See Fig. 3.6-9.) It appears in computer circuitry under the name of NAND.

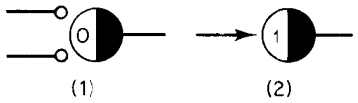


Fig. 3.6-7. (1) NOR, (2) DELAY.

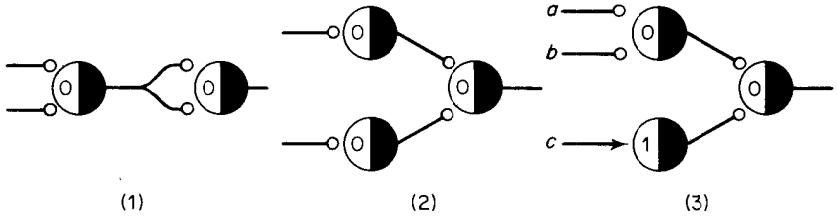


Fig. 3.6-8. (1) OR, (2) AND, (3) a OR b AND NOT c.

The MAJORITY element forms an attractive base when combined with an INVERTER (Fig. 3.6-10), for it is easy to make AND's and OR's, as shown in Fig. 3.6-11, and it is not hard to make any of the other combinations.

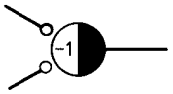


Fig. 3.6-9. NAND.

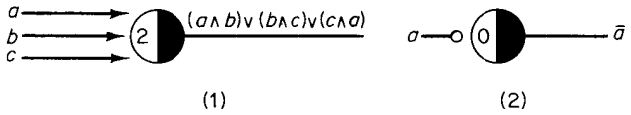


Fig. 3.6-10. (1) MAJORITY, (2) INVERTER.

There are a number of other simple combinations, some mentioned in von Neumann [1956], pp. 50–56. The reader will note that our exposition here is an expansion of von Neumann's.

There is another interesting element which forms a single-cell base (in the two-moment time scale). This is the cell (or rather, net) of Fig. 3.6-12. This cell, discussed in Minsky [1956] is interesting because it has a

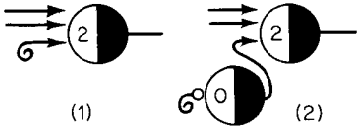


Fig. 3.6-11. (1) AND, (2) OR.

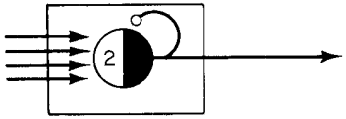


Fig. 3.6-12. Refractory cell.

"refractory period"—it cannot be fired on two successive moments—a basic property of biological neurons. It is interesting that this property can be used to replace the use of inhibitory connections. Why this is so will be explained in the next section.

3.6.1 Monotonic functions

It is easy to see that the two elements AND and OR do not by themselves make a complete universal base. (They do, as we showed in 3.5, if we allow external decoders and encoders, but we are now considering the complete problem.) They are deficient in a certain ability, which the reader will discover if he tries to use them to make a net which simulates, e.g., the "*A* and not *B*" element. The trouble is that every net composed only of AND's and OR's has a *monotonic* stimulus-response property. That is, *it is impossible to make the output smaller by making the input larger.* Suppose that a certain pattern S_1 at the input produces a certain response pattern R_1. Now suppose that we construct a new input S_2 which contains all of the impulses in S_1 (in the sense of section 3.4) plus an additional pulse. Then the output R_2, resulting from S_2 must contain all the pulses of the response R_1 to S_1, and possibly more.

To see that this is true, consider the propagation of pulses in any net composed exclusively of AND's and OR's. At each stage, the activity in response to S_2 must *include* the activity that would result from S_1 at the same moment. If S_1 causes a cell to be fired, then S_2 will at least meet the same threshold. There is no way in which the presence of the new pulse in S_2 can cause there to be *less* activity than before. If a threshold is already reached, there is no way a new pulse can prevent an AND or an OR cell from firing. The argument continues, step by step, through any network whose cells are individually monotonic (as are AND and OR).

What must one add, to AND and OR, to make a complete base? We know that admitting an inhibitor connection, e.g., in the form of the "*A* and not *B*" element, will suffice. In fact, it can be shown that one need only add *any* non-monotonic element whatever! The proof of this (Minsky [1956]) is rather complicated and will not be developed here.

It is easy to show that using only AND and OR, one can realize any monotonic function whatever. To do this, observe that one can simply construct a decoder like that in section 3.3.2 *without putting in the inhibitory connections.* Some thought should show that the inhibitory connections are there precisely for obtaining non-monotonic behavior—to permit a larger stimulus to block a response that would occur with a proper part of that stimulus. This is not needed when we are realizing a monotonic function.

It follows from the above statements that, given AND and OR, we can obtain a complete base by *adding any other element that cannot already be realized by a net composed of* AND*'s and* OR*'s.* Among these are the elements shown in Fig. 3.6-13 (the last of which is already known to be universal by itself). All of these remarks are subject to the restriction that the machines composed of the elements may have to be operated with delays that require the input signals to be sufficiently separated in time.

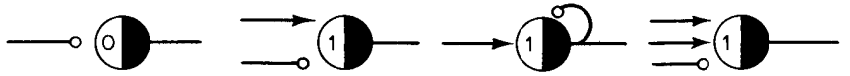


Fig. 3.6-13. Non-monotonic elements.

It follows that our "refractory" cell is universal, since it is non-monotonic, and since it evidently has AND and OR built in very directly.

3.6.2 The double-line trick

The discussion in 3.6.1 seems to say that one can do very little, given only monotonic cells like AND and OR. Yet section 3.5 and particularly the discussion of Fig. 3.5-1 seems to say that the most general finite-state machine (e.g., Fig. 3.5-1) can be assembled essentially from just these two elements. How can both statements be true? The only difference is that in section 3.5 the inputs to the machine are assumed to be decoded—at each moment there is a signal on *precisely one* of the input lines. This means that in the design of a machine using those inputs, the occurence of a signal on one line means that we can count on the non-appearance of signals on any other line. Therefore, we don't have to be concerned about inhibiting responses due to signals on those other lines, and it happens that we don't have to worry about any other inhibition problems, either.

It turns out that we don't need full decoding at the source to eliminate the need for inhibition, but we can get away with some weaker source of non-monotonic behavior, and this can be built into the signal source in various ways. Von Neumann [1956] mentions an elegant way to do this. *Let us represent signals using pairs of fibers* instead of single fibers. *In each pair of lines, we assume that one and only one is firing, at any moment.* (One line corresponds to the original fiber; the other has pulses at just those times the original fiber does not fire.) Then we can compose the functions AND, OR, and AND NOT by the nets of Fig. 3.6-14, each of which has a proper "double-line" output.

So if we use the double-line trick, we see that just AND and OR form a universal base, and incidentally that these functions can play entirely symmetrical roles! We can see further that NOR is universal alone: the left-hand net of Fig. 3.6-15 realizes the universal single-cell base OR AND NOT but is composed entirely of NOR cells:

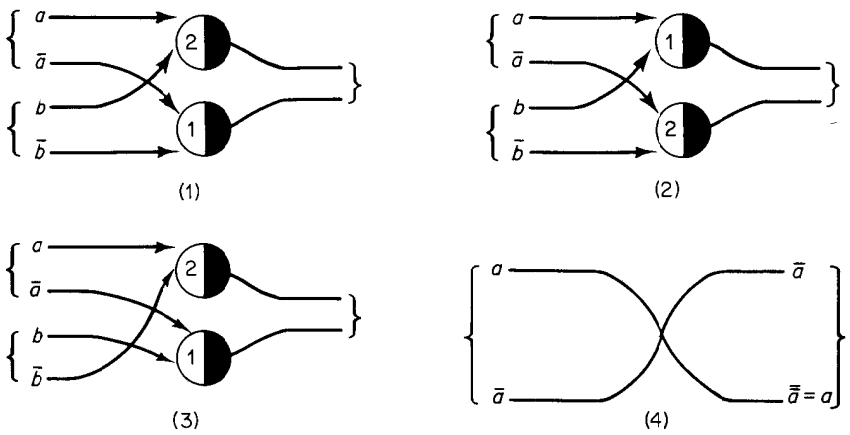


Fig. 3.6-14. The double-line trick. (1) AND, (2) OR, (3) AND NOT, (4) NOT.

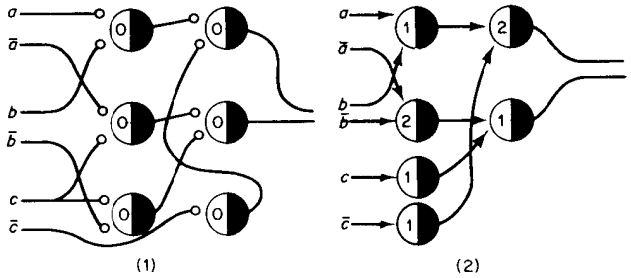


Fig. 3.6-15. (1) OR AND NOT (of NOR cells), (2) OR AND NOT (of AND's and OR's).

3.6.3 Other kinds of parts for finite-state machines

We have selected McCulloch-Pitts neurons for this exposition chiefly for their simplicity. No one uses precisely these elements in computers today, although some components in use are quite similar. By and large, the elements of computers do resemble our cells, but for reasons of packaging and wiring economy, one usually finds more in each unit package—the equivalent of about three or four of our cells. At the time of this writing, computer construction is moving in the direction of more integrated "micro-circuitry" in which much more complicated functions are realized in circuits sprayed, plated, diffused or otherwise embedded in a homogeneous matrix material—e.g., a semi-conductor plate. It isn't clear at this writing to what extent such devices will continue to be assembled from small universal bases of elements. As the control over fabrication methods improves, we can expect the more delicate "threshold-logic" kind of circuit to play a larger role.

Another basis for finite-state machines can be found in relay-contact networks. Operated synchronously, relays can be made to act as though time jumped in discrete moments. It is very easy to construct our encoders, decoders, and other logical functions as networks of relays; this is done in the paper of Shannon [1949]. Our chief reason for using "cells" instead of relays is that relays have a useful but confusing peculiarity—closed contacts conduct current in *either* direction. Hence, in relay contact networks, there is no natural notion of direction of *signal* flow, and this detracts from expository clarity. The relay computer, with its mechanically limited speed, has been obsolete since the electronic computers of the late 1940's; but its theory is again becoming important in the design of computers using cryogenic (superconductor) elements and field-effect transistor elements.

Still other network bases are found in the vacuum-tube logic of multigrid elements (currently out of fashion, but bound to be resurrected in connection with field-effect transistors and other multi-element semiconductors) and the currently popular forms of transistor-diode logic. But details concerning such matters would point away from the theoretical direction we want to take.

PROBLEM 3.6-1. (Suggested by E. F. Moore.) Design a net, using any number of two-input AND and OR cells, *and only two* NOT *cells*, that has the input-output properties of Fig. 3.6-16. Each output line is the *complement* of the corresponding input—that is, if x is 0, x' must be 1, etc. It is convenient to do the work with delay-free elements. The solution net (even with delay-free units) is quite hard to find, but it is an extremely instructive problem to work on, so keep trying! Do not look at the solution unless desperate.

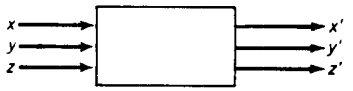


Fig. 3.6-16

To what extent can this result be applied to itself—that is, how many NOT's are needed to obtain K simultaneous complements? This leads to a whole theory in itself; see Gilbert [1954] and Markov [1958].

NOTES

1. For a superb presentation of what is known about the elements and networks that constitute real animal nervous systems, read Part I of Bullock and Horridge's "Structure and Function in the Nervous Systems of Invertebrates" [1965]. There is still no generally accepted theory of the mechanisms involved in learning in nervous systems. (See note 3.) The situation with respect to function of specialized parts of the nervous system is somewhat better; see, for example the work on the visual system described in the papers of Lettvin, Maturana, McCulloch, and Pitts [1959] and of Hubel and Wiesel [1959].

2. The models in which inhibition and excitation play symmetric roles include ours, one allows use of many inhibitory input connections from a single fiber. McCulloch, himself [1959, 1960] uses subtractive inhibition in his demonstration that one can construct neural net machines whose behavior is immune to certain kinds of fluctuations in the neurons' thresholds. The question of how to make machines that remain *reliable* under fluctuating conditions within their parts is a fascinating topic (not otherwise mentioned in this book). The results of a number of theories, notably of von Neumann [1956], Moore and Shannon [1956], and the cited work of McCulloch, show that for a variety of kinds of disturbances one can make machines as reliable as desired, at the price of introducing "redundancy"—duplication of parts—in appropriate ways.

   While talking about subtractive inhibition, we should certainly cite the earlier work of Rashevsky [1938, 1940], who first had the sense and the courage to try to make mathematical models of complicated neural nets.

   The introduction of subtractive inhibition raises some surprisingly complicated questions, even for single-neuron networks! We will not treat this subject here, either; it is called "Threshold Logic" and the reader might look at Dertouzos [1965] for a survey and at Minsky and Papert [1966] for some recent theoretical developments.

3. *Memory.* Unfortunately, there is still very little definite knowledge about, and not even any generally accepted theory of, how information is stored in nervous systems, i.e., how they *learn*. Most of the evidence leads one to believe that there are several mechanisms—at least different for *short-term* and for *long-term* memory. One form of theory would propose that short-term memory is "dynamic"—stored in the form of pulses reverberating around closed chains of neurons—while long-term memory is static—stored in the form of changes in connections, thresholds, of the microanatomy. For speculative theories of this variety, see Hebb's [1949] book on cell-assemblies. A more static model is that of Rosenblatt [1962]. Recently there have been a number of publications proposing that memory is stored, like genetic information, in the form of nucleic-acid chains, but I have not seen any of these theories worked out to include plausible read-in and read-out mechanisms.

4. The allowed "fan-out" numbers in today's computer circuitry are of the order of 6; higher when long lines or "busses" are driven by special amplifiers. In nervous systems the situation is somewhat different because the nerve fiber itself acts like a continuous amplifier that makes up for local drains, within limits. In some cases the numbers of neurons that affect, or are affected by, a given neuron is in the thousands.

5. For further discussion of this point, read the related papers by: McCarthy [1956], a paper whose importance has, I think, not been generally recognized; Solomonoff [1964], also of great philosophical importance; and Minsky [1959] for some remarks concerning these and the analogous, more precise, results of Shannon [1949]. The present book provides more or less enough mathematical background for reading these (except, perhaps, Shannon), but the reader will have also to bring along a good deal of sensitivity and insight to understand the approaches to the philosophical problem of "inductive inference" suggested in the McCarthy and the Solomonoff papers.

4 THE MEMORIES OF EVENTS IN FINITE-STATE MACHINES

4.0 INTRODUCTION

In chapter 2, we talked of machine states and classes of histories. We observed there, and in chapter 3, that there is a connection between the idea of machine "memory" and classes of histories. Now we examine the nature of this connection. First we will take a closer look to see what an "equivalence class of histories" is really like. Then we study the structure of these curious objects, using more formal methods. We find a way of describing concisely exactly what kinds of events finite-state machines can remember, what kinds of things they can recognize, and what kinds of computations they can perform.

Of course, when we say "remember," or "recognize," we are speaking about particular idealized notions of memory and recognition. These may not be in perfect harmony with our common-sense notions of such matters. The theory is explicitly only about the limitations of finite-state machines. On the other hand, there is no current scientific reason to suspect that living organisms can, *in isolation*, transcend these limits in any way important for this context. Later on we see that a creature's ability *to use the environment* for storage of records or—almost the same thing from our viewpoint—for material for growth, *does* make a difference in regard to these limitations.

The main mathematical discoveries in this chapter were made by the logician Stephen C. Kleene [1956]. In turn, this was based on the earlier McCulloch-Pitts [1943] exploration with the same general goals. There remain unsolved many important problems centered around the relations between the structures of nets, the structures of state diagrams, and the structure of memories.

The arguments in section 4.3 and section 4.4 require some mathematical "sophistication," though no specific prior knowledge on the reader's part.

Fortunately, these arguments are not required in the sequel, so the reader who gets enmired here should not hesitate to move on. While the later chapters may *look* even more complicated to browsers, they are in fact not so. Except perhaps for a few points near the end of the book, section 4.3 is the most intricate, but once understood it is really quite simple.

4.1 THE MEANING OF AN OUTPUT SIGNAL: FOUR EXAMPLES

Our first goal is to clarify the structure of the "equivalence classes of histories," mentioned in chapter 2. A few examples will illustrate what these classes are like. We discuss them first informally and then in terms of a more mathematical description.

Each example to follow is based on a McCulloch-Pitts net. For uniformity we will suppose that each has four input fibers called *a*, *b*, *c*, and *d*. For the present we assume that *at each moment precisely one and only one of these fibers is fired.* That is, *at each moment the net will receive just one of the possible signals a, b, c, or d; we will not consider the simultaneous arrival of combinations of such signals.* (They can thus be thought of as arriving from the output of some decoder network.)

In each example, the machine begins its operation with the occurrence of a pulse along a special fiber *S*. This fiber is called the "start fiber" and the pulse is called the "start pulse." For the present we will assume that *only one start pulse ever occurs*; only in section 4.4 will we consider exceptions to this. One of the ordinary input signals must also occur at the time of the start pulse.

Finally, each net has a special output fiber *R*. We will ask the same question in each case: what is the significance of a pulse along *R*. More precisely, suppose that we observe a pulse along the *R* fiber and that we do not know when the machine was started or what signals it has received. What can we say about what could have happened in the past history of the machine?

EXAMPLE 1

In our first example, fibers *b*, *c*, and *d* aren't connected to anything. (See Fig. 4.1-1.) If *R* fires at time *t*, we can deduce that *a* fired at time $t - 1$. Furthermore, since the threshold of the *R* cell is 2, we are sure also either that *S* fired at $t - 1$ or else that *R* did.

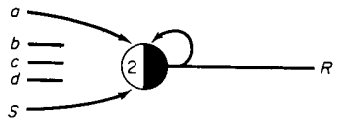


Fig. 4.1-1. Net for Example 1: $a*a$.

The third possibility—that *S and R* both fired at $t - 1$—is ruled out because before *S*'s unique firing there could have been no way for *R* ever to have got started. We deliberately ignore a last possibility—that *R* has always been firing—in an infinite past.[1]

Working backwards, we observe that *a* must also have fired at $t - 2$, $t - 3$, and indeed at every moment in the past, back to the time when *S* was fired—that is, when the machine was first started into operation. Clearly we *cannot* tell how long ago that event occurred. But we can be sure that the firing of *R* at a certain time *t* implies that *a* must have been fired at every moment from the Start to the time $t - 1$. We might say that the firing of *R* means that this net "recognizes"

any sequence composed entirely of one or more *a*'s.

We assumed that the start pulse *S* occurs only once. If this event is accompanied by an *a* signal, the *R* cell fires. So long as *a* signals continue to arrive, the *R* cell keeps firing; one may think of this as a reverberation facilitated by the *a* signals. If the *a* signal ever fails to appear, the reverberation must die out forever, since there will be no further occasion to meet the required threshold of 2 for the *R* cell. Since we assumed that just one input fiber is fired at each moment, the occurrence of any of the signals *b*, *c*, *d* cause the cessation of *R*'s firing, just because this would cause *a* to miss a pulse.

EXAMPLE 2

Our second net (Fig. 4.1-2) will recognize

any sequence that ends with an *a*.

For if *R* fires at time *t*, we conclude that *a* must have fired at time $t - 1$. And now there is simply nothing more we can say about what happened before that, for the reverberation initiated by the *S* pulse does not require any particular input signal pattern for its maintenance.

EXAMPLE 3

The third net imposes a rather more complicated constraint on the class of sequences that might have caused *R* to fire. (See Fig. 4.1-3.) We observe that the *first* (earliest) signal—that is, the signal concurrent with

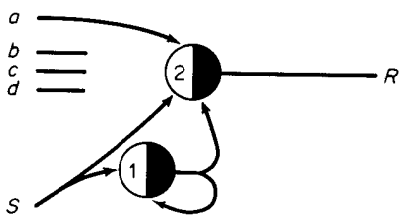


Fig. 4.1-2. Net for Example 2: $(a \vee b \vee c \vee d)*a$.

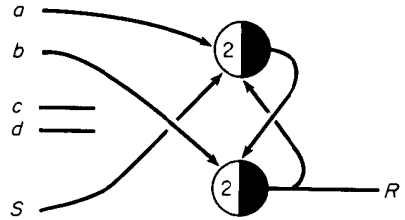


Fig. 4.1-3. Net for Example 3: $a(ba)*b$.

the S pulse—must have been an a. Otherwise the start pulse, alone, would have failed to fire any cell, and the net would remain quiet ever after. We observe also that the most recent pulse must have been b, else R could not fire. A little thought will show that the structure of the net causes R to recognize

> any sequence that begins with an a, ends with a b and in the interim repeats the sequence ba any number (including none) of times.

Alternatively, one might say that R recognizes

> any sequence that begins with ab followed by any number (including none) of repetitions of ab.

Only such a sequence can preserve the pulse reverberating within the net; this pulse, which originates from S, is required to meet the threshold R.

EXAMPLE 4

The class of histories defined by our final example is more difficult to describe. (See Fig. 4.1-4.) This net can become active only if the start

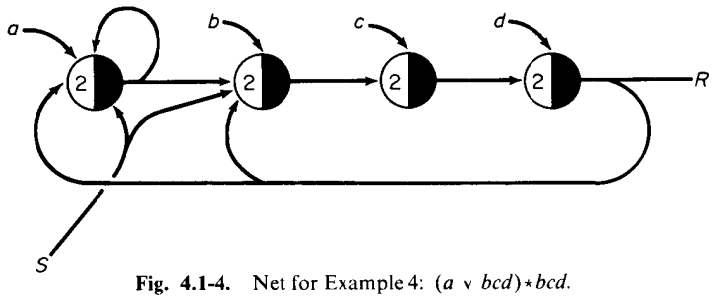


Fig. 4.1-4. Net for Example 4: $(a \vee bcd)*bcd$.

pulse on S is accompanied by a pulse on a or on b. The activity (ultimately required to fire R) can be kept alive only by (1) reverberating in the cell connected to a or (2) by cycling around the loop formed by the sequence bcd. Firing of R can occur only after the completion of bcd sequence. Some thought will then reveal that the firing of R recognizes

> any sequence, made up of any number of successive or mixed occurrences of the patterns a and bcd, which ends with an occurrence of bcd.

DISCUSSION

In each of the examples we were able to describe the class of sequences recognized by the machine, using expressions (*in boxes*) of a more or less everyday language. For nets of greater complexity such expressions would become cumbersome, ambiguous, and finally incomprehensible. Kleene's [1956] formulation describes such classes in an elegant manner, using formulas he called "regular expressions" and a number of associated notions. The definitions in the next section are not precisely those of Kleene but are more like those in Copi, Elgot, and Wright's [1958] simplified version of Kleene's theory.[2]

4.2 REGULAR EXPRESSIONS AND REGULAR SETS OF SEQUENCES

Let us paraphrase the four descriptions obtained above for the sets of sequences recognized by the nets of Figs. 4.1-1 through 4.1-4.

(1) Any number of occurrences of a followed by a.
(2) Any number of occurrences of (a or b or c or d) followed by a.
(3) a followed by any number of occurrences of ba followed by b.
(4) Any number of occurrences of (a or bcd) followed by bcd.

Now observe that these expressions (and those arising from more complicated nets) can all be assembled from a very few terms and connectives. We require, in fact, only the signal letters themselves and three connectives:

"any number of occurrences of"
"followed by"
"or"

If we abbreviate these three by '$*$', juxtaposition,[†] and '$\vee$', respectively, the above four expressions take on the compact forms:

(1) $a*a$
(2) $(a \vee b \vee c \vee d)*a$
(3) $a(ba)*b$
(4) $(a \vee bcd)*bcd$

[†]The term *juxtaposition* refers to the formation of a new expression by simply writing several old expressions consecutively without any separating punctuation. In this informal exposition we introduce parentheses only where needed to mark the limits of the '$\vee$' and '$*$' connectives.

It is understood that juxtaposition is a "stronger" bond than is '$\vee$'. That is, $a \vee bc$ means $a \vee (bc)$ and not $(a \vee b)c$. This is exactly as in school-algebra, where multiplication is denoted by juxtaposition and $a + bc$ means $a + (b \times c)$ and not $(a + b) \times c$.

We list just some of the sequences included in the sets represented by expressions 1–4 above.

(1) *a, aa, aaaa, aaaaa, ...*
(2) *a, aa, ba, ca, daa, aba, cca, bca, bcacdabbccbbdcaabcaaa, ...*
(3) *ab, abab, ababab, abababab, ababababab, ...*
(4) *bcd, abcd, bcdbcd, bcdabcd, aabcdaaaaaabcdbcdabcd, ...*

While it is perhaps clear how these expressions are meant to be read, we must state explicitly what is involved. We will define any string of signal letters, stars, v's, and parentheses to be a *regular expression* provided that it can be constructed in accord with the following rules. Each regular expression will serve to *represent* a certain set of signal sequences. Such a set is called a *regular set of sequences*. The rules below also explain how "representing" works.

DEFINITION OF THE CLASS OF REGULAR EXPRESSIONS

Any letter symbol x is, alone, a regular expression.

It *represents* precisely the set consisting of the single (one-letter) sequence x.

If E and F are regular expressions, then so is (EF).

The set of sequences *represented* by (EF) is obtained as follows: choose any sequence s_1 from the set represented by E and any sequence s_2 from the set represented by F. Then the sequence formed by attaching s_2 to the end of s_1 is a sequence in the set represented by (EF), and only such sequences are in the set.

If, E, F, ..., G are regular expressions, then so is the expression $(E \vee F \vee \ldots \vee G)$.

The set of sequences *represented* by $(E \vee F \vee \ldots \vee G)$ contains all and only those sequences which are already in any of the sets represented by E or F or ... or G. (That is, simply form the "union" of those sets.)

If E is a regular expression, then so is $E*$.

The set of sequences *represented* by $E*$ is obtained as follows: let $s_1, s_2, \ldots, s_k$ be any collection of sequences of the set represented by E. Then the sequence obtained by stringing them all in a row is a sequence of $E*$, and only such sequences are. *Warning*: We permit also the case of no sequences at all—that is, we allow k to be zero. This leads occasionally to some inconvenience in that we have to talk about the "*null sequence*"—the sequence of no signals at all.†

The regular expressions are all those defined by the above rules, and no others.‡

The *regular sets of sequences* are all sets of sequences defined by the above representation rules.

†One can think of $E*$ as: (null sequence) v E v EE v EEE v ...

‡But we allow omission of parentheses when no ambiguity can result; e.g., we write EFG for $(E(FG))$ or $((EF)G)$. Why is this permissible?

We have used what mathematicians call a "recursive definition." In this form of definition one begins with a statement that:

BASE: Certain ("primitive") objects are definitely in the class.

Then there are some rules which state that:

RECURSION: If certain kinds of objects are in the class, then so are certain other objects formed from them.

Finally, there is usually stated a restriction that:

RESTRICTION: No objects not required to be in the class by the above rules are in the class.

PROBLEM 4.2-1. The input to a net is a sequence of 0's and 1's. Represent the net—as a regular expression (easy), as a state diagram (harder), and as a McCulloch-Pitts network (usually very hard)—in each of the following cases.

An output pulse occurs when:
(1) the number of 1's in the input is divisible by 3.
(2) all 1's have been in blocks of at least 3.
(3) no 1 has occurred at a time divisible by 2 or 3.
(4) there has been an even number of blocks of 1's, each of odd length.

4.2.1 Recursive definitions and inductive proofs

The notion of recursive definition is important in its own right, and we will be concerned with it again later in the book. Therefore it is appropriate here to take time for a more detailed study of what is involved. Let us look again at the recursive definition of "regular expression," this time without so much concern about its interpretation. *Let K be the class of regular expressions.*

BASE: Any letter symbol $a, b, c, \ldots$ is an expression in K.
RECURSION: If E and F are expressions in K, then so is (EF).
If $E_1, E_2, \ldots, E_n$ are expressions in K, then so is

$$(E_1 \vee E_2 \vee \ldots \vee E_n)$$

If E is an expression in K, then so is $E*$.
RESTRICTION: Only expressions generated by the above rules are in K.

Very often, one does not state the restriction explicitly, it being understood that one is not concerned with "richer" systems, or "extensions" which may happen to contain K.

The terms "recursion" or "recursive" are used in reference to the fact that, in such a definition, the name of the class being defined recurs within the definition itself, in an essential way. In the present case we are defining a certain class K. The term K appears in the BASE part of the definition, but not in a circular way; here we only say that certain things are in K, and one does not already have to know anything about K to understand *this*. But in the RECURSION part of the definition, one says things like "...if E is in K, then...," and here it appears that one has already to know something about K—the thing being defined—to make sense of the definition! Thus there is something about a recursive definition that suggests that it might be "circular," and hence somehow unsatisfactory. Indeed, this can easily be the case. Thus, if we were to omit the BASE part of the above definition of K, the remaining definition would be quite useless, for one would have no way whatever to tell which expressions are, in fact, in K and which expressions are not.

In later chapters one of our central goals will be to study just this question of when a recursive definition really works and when it is defectively circular. For the present, it will be useful to study in some detail a few examples of recursive definitions which turn out to be satisfactory. Our first example is taken from the expressions of elementary school-algebra.

4.2.2 The set of well-formed parenthesis strings

Consider the expressions used in elementary algebra. If we examine the expressions

$(a + b)$	$)a +$
$a(b + c(d + e))$	$a(b + c(d + e)$
$(a + b)(c + d)$	$)a +)b(c + d($
$a(b + (c + d) + e)$	$a)b + (c +)d + e($

we observe that those in the first column are meaningful while those in the second column are not. One can think of this distinction as a matter of *grammar*; the expressions in the second column are somehow "ungrammatical."

Parentheses are punctuation marks which are supposed to be *paired* to mark out phrases or clauses. In the second column, it simply isn't clear what clauses, if any, are defined. If we abstract out the parenthesis structure alone, we can still make the distinction, for in each case

()	)
(())	(()
)()	))((
(())	)()(

we can see which are "well-formed" and which are not!

Here is a recursive definition which generates just the class of parenthesis structures that *are* grammatical.

RECURSIVE DEFINITION OF THE CLASS P

BASE: () is in P.	(P_0)
RECURSION:	
(1) If E is in P, so is (E).	(P_1)
(2) If E and F are in P, so is EF.	(P_2)
RESTRICTION: Nothing else is in P.	

This generates, among others, the expressions

()				
(())	()()			
((()))	(()())	(())()	()()()	()(())
(((())))	((()()))	((())())	(()()())	(()(()))
(())(())	(())()()	()()(())	()()()()	
()((()))	()(()())	()(())()		
((()))()	(()())()			

which are all the well-formed structures of up to eight symbols.

Now let us return to the question of how to decide when a recursive definition really defines something, and is not fatally circular. We will not try to give a general answer to this question. In fact, as will be seen in later chapters, there is no hope of finding a general answer. However, in particular cases we can often solve the problem. Here we can give a simple procedure by which one can tell whether a given expression is or is not in the class P. There are many ways to do this, but the one below is certainly the one that involves the least effort for many expressions.

4.2.3 The method of parenthesis-counting

Our procedure is based on the construction of a certain machine to recognize which parenthesis strings are well-formed—that is, are in the class P defined above. The machine we use is much like those considered in former chapters, except that it has an infinite number of states: (see Fig. 4.2-1).

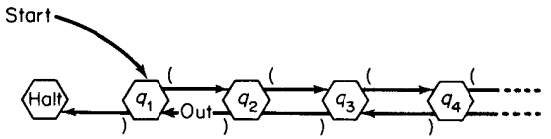


Fig. 4.2-1. The parenthesis-counting machine M.

For any particular example, only a finite number of states will be used, so we need not discuss now what is implied by having an infinite machine; this is a topic of later chapters. We will see, though, that we have not gone over to infinite machines for frivolous reasons; there is in fact no finite machine that can handle this job, at least for the whole class of expressions made up of parentheses.

Now let E be an expression made up of parentheses. Define the function $Q_i[E]$ to be *the state that M will end in if it is started in state q_i and given the sequence of symbols from E.* We will prove the following theorem.

THEOREM

An arbitrary expression E is in P if, and only if, it has the property that

$$Q_1[E] = q_1$$

Note that this is a very strong theorem; it tells us not only which expressions are in P but also which are not. The proof is instructive because it shows how one can relate the structure of a recursive definition (here, of P) to the structure of a machine (here, M).

Plan of proof: Define C to be the class of expressions with the property $Q_1[E] = q_1$. We really have to prove two things, first that

If E is in C, then E is in P

and then that

if E is in P, then E is in C.

Then it will follow that both classes must be the same.

Proof that if E is in P, then $Q_1[E] = q_1$; that is, E is in C:

We simply prove that the property holds for everything in P, recursively.

BASE: $Q_1[()] = q_1$. This is seen, simply by putting () into the machine, starting in state q_1. Hence

() is in C.

RECURSIONS: Suppose $Q_1[E] = q_1$. Then we know that *state H is never reached* if the machine is started in state q_1 and given sequence E. It follows that

$$Q_2[E] = q_2$$

by translating everything up one unit to the right. Note also that for the sequence $(E$, we have

$$Q_1[(E] = Q_2[E] = q_2$$

Hence

$$Q_1[(E)] = Q_2[E)] = Q_2[)] = q_1$$

So

If E is in C, so is (E).

Next, suppose that E and F are in C. Then in computing $Q_1[EF]$, the machine returns to state q_1 after seeing E (because $Q_1[E] = q_1$) and returns finally to state q_1 after seeing F (because $Q_1[F] = q_1$). Hence

If E and F are in C, so is EF.

Now the three statements in boxes are exactly like the three statements in the recursive definition of P (section 4.2.2). Can we conclude that the classes C and P are the same? No, but only because we have not yet proven the RESTRICTION statement. We can, however, be sure that anything in P is also in C; that is, the class P is contained in the class C, which is what we wanted to prove here. Now we have to prove that the class C is contained in the class P.

Proof that if E is in C, then E is in P:

This is the more interesting, less obvious, part. We need a new method to deal with the recursion, and we choose to use *mathematical induction on the length of expressions.* What we will do is to show that if the theorem is true for all expressions of less than a certain length, then it must be true for expressions of that length. An "induction" is a simple form of recursion. Let us define *length* $[E]$ to be the number of symbols in E.

BASE: If length $[E] = 2$ and E is in C, then E is in P. This is true, since inspection shows that the only expression which meets the conditions is (). And this is in P because of the BASE (P_0) of the definition of P.

INDUCTION: Assume that we have shown, for every expression of length *less than* k, that if it is in C then it is in P. Now consider any expression E of length k. We can separate the situation into two cases. In the first case it is possible to break E into two parts F and G such that $E = FG$ and both F and G are in C. In the second case there is no such partition of E.

CASE I. This is trivial. For if $E = FG$ and both F and G are in C, then both F and G are shorter than k (the length of E). Then by the induction hypothesis, both of them are in P. Hence (by the P_2 of the definition of P) it follows that E must be in P.

CASE II. If E is in C, then $Q_1[E] = q_1$. If there is no partition of E into two shorter expressions both in C, we can conclude that the testing machine does not re-enter state q_1 until the end of E. Therefore, it must

enter state q_2 immediately after the first symbol of E, and it must enter state q_2 on the next-to-last symbol of E (for it is only from there that it can get to state q_1 on the last symbol of E). It follows that E must have the form (F) and that $Q_2[F] = q_2$. Now observe that the machine never enters q_1 during the computation of $Q_2[F]$. Therefore, if we start the machine in state q_1, with input F, we can conclude also that $Q_1[F] = q_1$, (for the machine structure always looks the same to the right of the starting point, and we know that this computation does not encounter what is to the left of the starting state). Hence, by definition, F is in C. Then, since F is shorter than k (the length of E), F must be in P, and hence (by P_1) so is $E = (F)$. This concludes the proof that if E is in C, then E is in P and, together with the previous section, the proof that E and C are the same.

REMARKS. We started by defining the class P of well-formed parenthesis sequences by using a rather simple recursive definition. We showed that it was possible to "recognize," or "decide," which sequences belong to this set, by using a simple sort of "counting machine." The use of this machine, or "decision procedure," is very straightforward; one simply feeds the expression into the machine and observes the state of the machine at the end. There is no question about how long the machine's computation will take: its number of steps is just the expression's length.

One can imagine other recursive definitions for which it would be more difficult to find such a decision machine—one which would definitely tell which expressions are in and which are out of the defined class. As will be seen later, there are even cases in which it is impossible for any such decision machine to exist. At best, in such cases, we can find only a machine which will sometimes decide, but will other times get involved in a never-terminating computation. Anticipating these developments, we can ask here why things came out so well. The answer is that in the case of the class P, the recursive definition has one very important special property. Each time an expression is admitted on the basis of other previously admitted expressions, the new expression has greater length. Thus (E) is longer than E and EF is longer than E or F. This fact allows us to work backwards, since for any expression we need investigate only a limited class of potential ancestors. This is reflected in the fact that we were able to complete the proof by an induction on the length of expressions.

There are proof methods for making decisions about the equivalences of classes given by different recursive definitions, using methods more general than our simple length arguments. The interested reader should study the method of "Recursion Induction" described by McCarthy [1960]. If we had described our testing machine by a recursive definition instead of by using an informal state diagram description, we could have proven the theorem by an application of the Recursion Induction method.

Curiously enough, the class C of expressions is *not* itself a regular set of sequences. As will be seen, they cannot be recognized by any finite-

state machine. We have just shown that they can be recognized by a very simple infinite-state machine. When we turn to the study of Turing machines, a variety of infinite machines, we will use this class as one of our first examples.

A proof by "induction" is related to a recursive definition of a collection of "numbered" quantities or objects. Suppose that we have a set of objects, each somehow associated with a number (in the situation above, each *expression* is associated with its *length*). We want to prove some statement about all the objects. If we can prove the statement separately about the objects associated with each number, then this proves it for all the objects. The scheme called "mathematical induction" has as its goal proving that:

> BASE: The statement holds for all objects with the number 1.
> INDUCTION: If the statement holds for all objects with number n, then it holds for all objects with number $n + 1$.

This clearly establishes the statement, step-by-step, for each number.

PROBLEM: What is wrong with the following proof?

Proposition: *All marbles have the same color.*

Consider a container of marbles. We prove that any handful of marbles are the same color. BASE: If we take 1 marble, then this set of marbles certainly has only one color. INDUCTION: Take $n + 1$ marbles. Choose n of them. By inductive assumption we can suppose these are all the same color. Now replace one of these n marbles by the extra marble. Then the extra marble must be the same color as the others, in the new set of n marbles. So all marbles must be the same color.

4.3 KLEENE'S THEOREM: FINITE AUTOMATA CAN RECOGNIZE ONLY REGULAR SETS OF SEQUENCES

We now resume the study of the capabilities and limitations of finite-state machines. We consider a finite machine M which starts in a certain state Q_{init} and ask, "Which sequences of input symbols cause M to end up in a certain state Q_{fin}?" Any sequence with this property is said to be "recognized" by the machine. The set of *all* such sequences is *the set of sequences recognized by M*. (Of course, by different choices of Q_{init} and Q_{fin} the same machine can recognize a number of different sets of

sequences.) The Kleene theorem states that *the sets so recognizable by machines are precisely the regular sets of sequences.*

We divide the proof into two parts. In this section we show that any set recognized by a finite-state machine is regular; in section 4.4 we show the converse—that any regular set can be recognized by some machine.

The proof (which is essentially that of Kleene) is based on mathematical induction on the number of states of the machine. That is, we show first that if the theorem holds for all machines with n states then it holds for all machines with $n + 1$ states. We show also that it holds for all one-state machines. It follows that the theorem is proved for all machines. The proof is based on arguments about the set of paths through the machine's state diagram from Q_{init} to Q_{fin}.

It is actually easier to prove the theorem for a slightly more comprehensive class of diagrams than just those for machines; the arguments hold also for "incomplete" diagrams in which any of the arrows may be deleted. This is an important phenomenon in mathematics—that it can be easier to prove a more general theorem. The profound problem is to find the right generalization!

Consider a diagram composed of vertices (points) and arrows connecting some of the points. We require (temporarily) that there be at most one arrow in each direction between any pair of points. Let each arrow carry a different letter label. For example, consider the diagram of Fig. 4.3-1.

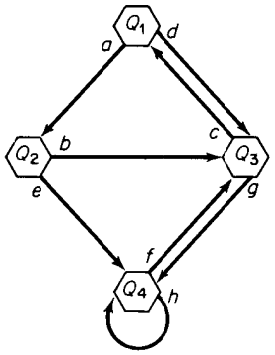


Fig. 4.3-1

We now consider paths from one vertex to another in the diagram. A *path* from x to y is a connected sequence of arrows which begins at vertex x and ends at vertex y. We place no additional restrictions on what is a path; in the example, *abcabcdcdcdgfgfghhh* represents a legitimate path from Q_1 to Q_4. Now given two vertices Q_x and Q_y (which may be the same) our task is to show that the set of all sequences from Q_x to Q_y is represented by a regular expression in the letters labelling the diagram. The argument breaks into two cases depending on whether Q_x and Q_y are different.

We denote by R_{xy} the set of all paths from Q_x to Q_y. We make an important definition:

R^z_{xy} = the set of all paths from x to y which do not enter and later leave again the vertex z. In the case that Q_z is the same as Q_x or Q_y (or both), R^z_{xy} contains paths that begin and/or end on Q_z but none that touch Q_z in the interior of the path.

Now if $Q_x = Q_y$ we can say that

$$R_{xx} = (R^x_{xx})* \tag{1}$$

and if Q_x and Q_y are different we can say that

$$R_{xy} = R_{xx}R^x_{xy} \qquad \text{(and also that } R_{xy} = R^y_{xy}R_{yy}\text{)} \tag{2}$$

Statement (1) means that any path from Q_x back to Q_x is made up of some arbitrary number (including zero) of trips between successive returns to Q_x. Statement (2) reflects the fact that *in any path from Q_x to Q_y there must be a last excursion through Q_x*; at this point the sequence from R_{xx} ends and the sequence from R^x_{xy} begins. If we can show that R^x_{xx} and R^x_{xy} are regular sets, it follows from (1) and (2) that so are R_{xx} and R_{xy}, which is what we have to prove. We will now show that all sets R^z_{xy} are regular sets.

Consider a set R^z_{xy}. Such a set is really concerned with a new diagram in which one state, Q_z, is more or less eliminated. If we can describe the set R^z_{xy} in terms of a diagram with a state deleted, we will be able to use the induction hypothesis; that the sets of paths between vertex pairs in smaller diagrams are regular. Let C_{ab} be the label for the arrow between Q_a and Q_b, if there is one. Otherwise the symbol C_{ab} is considered meaningless. A label C_{aa} is always meaningful; we will interpret it later. The key step in the argument is to observe that the set R^z_{xy} is composed of the path C_{xy} (if any) together with all meaningful sequences described by expressions of the form.

$$C_{xm}R^z_{mn}C_{ny} \tag{3}$$

in which *neither m nor n is the same as z.* Why is this true? Because any path from Q_x to Q_y which is in R^z_{xy} must go either directly from Q_x to Q_y or else through some chain of states which does not touch Q_z in its interior. There must be some state Q_m which is the first in this chain (after leaving Q_x) and there must be some state Q_n which is the last (before reaching Q_y); the remainder of this chain must be in R^z_{mn}. Observe that if each set R^z_{mn} so mentioned in (3) is regular, then so must be R^z_{xy}, for it is composed of these and the C's by 'v' and concatenation.

The final step: by the induction hypothesis the sets R^z_{mn} of (3) above must be regular. For each is a set of paths between two vertices in a graph which has fewer vertices than occurred in the original. In fact, the graph is that obtained by removing the vertex Q_z together with all its arrows! Hence, working back, the sets (3), (2), and (1) are all regular and, hence, so are R_{xx} and R_{xy}. We have finally to show (the induction base) that the

theorem holds for the one-vertex case. But the set of sequences from Q_1 to Q_1 is just $(C_{11})*$ which is regular if C_{11} is.

To apply the argument to the state diagrams of machines, we have to interpret the symbols C_{ab} properly. If Q_a and Q_b are different machine states, there may or may not be any arrows leading directly from Q_a to Q_b. If there are none, we simply omit any expressions of form (3) which contain a C_{ab} term. If there are several such arrows, labelled with letters $a_1, \ldots, a_n$, we replace the symbol C_{ab} by the expression $(a_1 \vee \ldots \vee a_n)$. We must treat the symbols C_{aa} differently. If there are arrows leading from Q_a directly to Q_a (as in the case of C_{44} of the example), we replace C_{aa} by the expression formed of the associated letters connected by v's, just as for the C_{ab}'s. (In the example, C_{44} is replaced simply by h.) But even if there are no arrows from Q_a back to Q_a, we must not eliminate entirely the corresponding expressions of form (3), because an expression like

$$C_{xx}R^{z}_{xn}C_{ny}$$

still can represent paths. In such a case we simply remove the C symbol, obtaining the expression $R^{z}_{xn}C_{ny}$, and similarly if the right-hand C has two identical subscripts. Carrying out this program, we find that all the C's and R's are eventually eliminated, yielding a regular expression using the input symbols of the machine. The base of the induction—that the set of signals recognized by a one-state machine is regular—is trivial since this set is just $(a \vee b \vee \ldots \vee z)*$ where $a, b, \ldots, z$ is the set of input symbols to the machine.

4.3.1 An example

Suppose that we wish to compute R_{11} for Fig. 4.3-1. We begin with

$$R_{11} = (R^{1}_{11})*$$

using rule (1). Then we expand R^{1}_{11}:

$$R^{1}_{11} = C_{12}R^{1}_{23}C_{31} \vee C_{13}R^{1}_{33}C_{31}$$

using rule (3); one can go only to Q_2 or Q_3 directly from Q_1. Since C_{12} is a, C_{31} is c and C_{13} is d, we can combine these to obtain

$$R_{11} = (aR^{1}_{23}c \vee dR^{1}_{33}c)*$$

Next, we expand

$$R^{1}_{23} = R^{1}_{22}R^{12}_{23}$$

using rule (2); but R^{1}_{22} is null, so simply

$$R^{1}_{23} = R^{12}_{23}$$

Next, we expand R^{12}_{23}, writing in the values of the C's as we generate them:

$$R^{12}_{23} = bR^{12}_{33} \vee eR^{12}_{44}f \vee eR^{12}_{43} \vee bR^{12}_{34}f$$

and continuing,

$$R^{12}_{33} = (R^{123}_{33})* = (gR^{123}_{44}f)* = (gh*f)*$$

$$R^{12}_{44} = (R^{124}_{44})* = (h \vee fR^{124}_{33}g)* = (h \vee fg)*$$

where the term $h = C_{44}$ is non-trivial and has to be included. Similarly,

$$R^{12}_{43} = R^{12}_{44}R^{124}_{43} = (h \vee fg)*f$$

$$R^{12}_{34} = R^{12}_{33}R^{123}_{34} = (gh*f)*gh*.$$

Backing up, this gives

$$R^{1}_{23} = b(gh*f)* \vee e(h \vee fg)*f \vee e(h \vee fg)*f \vee b(gh*f)*gh*f$$

We are not surprised to find terms duplicated, since the same subgraph of Q_3 and Q_4 has been approached in several ways. The above expression is equivalent to

$$R^{1}_{23} = b(gh*f)* \vee e(h \vee fg)*f$$

It remains to expand

$$\begin{aligned} R^{1}_{33} &= (R^{13}_{33})* \\ &= (gR^{13}_{44}f \vee gR^{13}_{43})* \\ &= (gh*f \vee gh*f)* \\ &= (gh*f)* \end{aligned}$$

Hence, finally, we get

$$R_{11} = (a(b(gh*f)* \vee e(h \vee fg)*f)c \vee d(gh*f)*c)*$$

PROBLEM 4.3-1. Explain in words the significance of the expressions in each of the steps above.

PROBLEM 4.3-2. Find the regular expression for R_{14} in the above net.

4.3.2 Remarks

There is no restriction that the letters in the machine diagram be all different. Let us relabel the diagram of Fig. 4.3-1 so that just two letters appear (Fig. 4.3-2). Simply by substituting the x's and y's in for the corresponding letters of the old diagram, we obtain the regular expression

$$R_{11} = (x(x(yy*x)* \vee y(y \vee xy)*x)x \vee y(yy*x)*x)* \qquad \text{(A)}$$

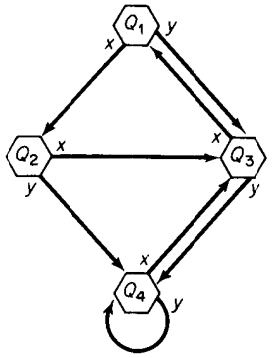


Fig. 4.3-2

for the set of paths from Q_1 back to Q_1 in the new diagram. This is the diagram of a genuine finite-state machine, since each letter occurs exactly once at each vertex.

This is not the only regular expression that can represent the set of paths in question. Each regular set has many regular expressions. There is a whole algebra for regular expressions, which allows many transformations and simplifications. (See problem 4.3-3 below.) For example, we can simplify the above expression for R_{11} all the way down to

$$((xy*x \vee y)(yy*x)*x)* \qquad \text{(B)}$$

This expression has an interpretation: To get from state Q_1 back to state Q_1, one has to go through state Q_3. There are two ways to get to state Q_3; these are $xy*x$ and y. (This is not so easy to see, in itself.) One can then delay returning to state Q_1 only by shuttling between states Q_3 and Q_4; $yy*x$ arises from each such cycle. Finally one must return to Q_1; this requires the signal x. It would be quite hard to *prove* that (A) and (B) are the same without inventing a set of formal transformation rules.

PROBLEM 4.3-3. Show that the sets represented by

$$b(ab \vee b)*a \quad \text{and} \quad bb*a(bb*a)*$$

are the same. Use this to establish the equivalence of (A) and (B) above. Realize that the statements are absolutely true, and do not depend on reasoning from any particular diagram. Invent some transformations between equivalent regular expressions and discover a correspondence between them and some transformations of state diagrams. For example, one might justify each step of $b(ab \vee b)*a = b(b \vee ab)*a = b(b*ab)*b*a = bb*a(bb*a)*$

PROBLEM 4.3-4. Interpret our analysis using relations (1), (2), and (3) of 4.3 in terms of operations which dissect state diagrams, splitting each vertex into several according to how many arrows leave it.

Kleene [1956, p. 24] gives the following results, among others. We quote:

"ALGEBRAIC TRANSFORMATIONS OF REGULAR EXPRESSIONS. We list some equalities for sets of tables. (We have scarcely begun the investigation of equivalences.)

(1) $E \vee E = E$. (2) $E \vee F = F \vee E$.
(3) $(E \vee F) \vee G = E \vee (F \vee G)$. (4) $(EF)G = E(FG)$.
(5) $(E*F)G = E*(FG)$. (6) $(E \vee F)G = EG \vee FG$.
(7) $E(F \vee G) = EF \vee EG$. (8) $E*(F \vee G) = E*F \vee E*G$.
(9) $E*F = F \vee E*EF$. (10) $E*F = F \vee EE*F$
(11) $E*F = E^s*(F \vee EF \vee E^2F \vee \ldots \vee E^{s-1}F)$ $(s > 1)$.

To prove (11) we have

$$E*F = \sum_{n=0} E^nF = \sum_{q=0} \sum_{r=0}^{s-1} E^{sq+r}F = \sum_{q=0} E^{sq} \sum_{r=0}^{s-1} E^rF.$$"

PROBLEM 4.3-5. Which of the following are true?

$$E*F = (E \vee E*)F$$
$$E*F* = (E \vee F)*(EF)*$$
$$E*F* = E*EF* \vee E*FF*$$
$$(E \vee F)* = (E* \vee F*)(F* \vee E*)*$$
$$E(FGE)*FG = EF(GEF)*G$$

PROBLEM 4.3-6. Show that given any regular set E, the set consisting of the members of E, each reversed in time, is also a regular set. (Prove by using the recursive definition of regular expression.)

4.4 KLEENE'S THEOREM (continued): ANY REGULAR SET CAN BE RECOGNIZED BY SOME FINITE-STATE MACHINE

We now prove the converse of the theorem of 4.3; we show that *there exists, for any regular set of sequences, a finite-state machine that recognizes precisely that set.* This (combined with the result of 4.3) shows that the sets recognized by machines and the sets represented by regular expressions, are co-extensive—that the notions of *regular set* and *set recognizable by some machine* are equivalent.

Our method is to show that any set representable by a regular expression can be recognized by a McCulloch-Pitts machine. Since any McCulloch net is a finite-state machine, the theorem will be proven. We could (with some difficulty) construct the state diagram of the machine directly, but we will leave this as an exercise. Anyway, since the McCulloch-Pitts net is a special case of finite-state machine, we get a

stronger result. (For, showing that any regular set can be realized by a *special* kind of finite machine is a stronger result than one which shows that any regular set can be realized as one of the *full* set of finite-state machines.)

We will break up the proof into two stages. The first stage (section 4.4.1) gives a very simple direct proof of a slightly weaker theorem, in which we use a new kind of cell. The second stage (section 4.4.2) is concerned with details revising the proof so that these new cells can be eliminated.

4.4.1. Realizing regular expressions with instant-OR cells

We will build up networks, recursively, to recognize regular sets or sequences, using the structure of the associated regular expressions. First we have to realize the single letter expressions: The nets in Fig. 4.4-1 realize the sets represented by a, b, c, ... Next we have to produce nets

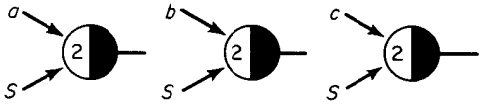


Fig. 4.4-1. Nets for single-letter signals.

for EF, $E*$, and $(E \vee F \vee \ldots \vee G)$ assuming that we have nets for the individual letters that appear in the expressions. For $(E \vee F \vee \ldots \vee G)$, we simply combine the outputs of the separate nets, using a new element —the instant-OR cell (Fig. 4.4-2).

The instant-OR cell acts like our usual threshold-1 cell, except that we assume that *it introduces no transmission delay*. We use it, in effect, so that we can tie different output fibers together. Because it does not fit in with our original requirements on finite-state machines, we will have to eliminate it before we can claim to have proven our theorem.

To realize the expression EF, we have only to tie together the nets for

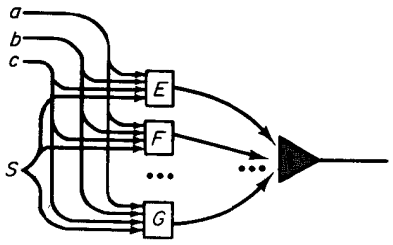


Fig. 4.4-2. Net for $(E \vee F \vee \ldots \vee G)$.

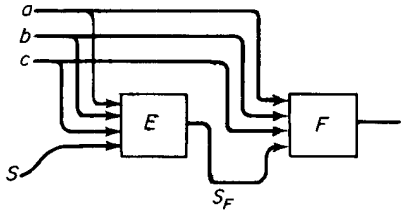
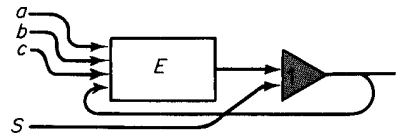


Fig. 4.4-3. Net for EF.

E and F serially as in Fig. 4.4-3. The idea here is that the output of the E net is used to supply start pulses to the net for F. Thus the F net should respond if (first) an E sequence occurs to provide a start pulse for F and (subsequently) an F sequence occurs to produce the F output. There is a complication here in that the F net may receive a history of several start pulses. But careful examination will show that this will work out perfectly and F will respond to any sequence which can be divided into an E sequence followed by an F sequence. See the remark below.

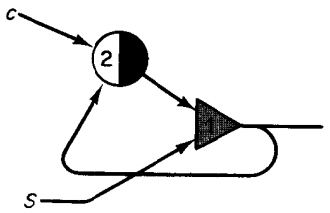
Finally, we have to realize $E*$, given E. The net of Fig. 4.4-4 does this. In this net, the output of the E net is connected back to the Start fiber (through the instant-OR cell). This means that whenever E recognizes an E sequence, it starts again looking for another one. Hence the net will recognize any sequence made up of a number of E sequences presented consecutively.



REMARK

The nets developed here may be exposed to more than one start pulse. How can we be sure they will do what we require of them? The answer is that we must include this provision within the recursive proof structure. In so doing, we find that we really must prove a theorem slightly stronger than is really required. Here is what must be done: we say that a net N *strongly recognizes* the regular set E if it has the property that N produces an output pulse at time t *if and only if the input history of* N *contains a sequence of* E *whose beginning was accompanied by a start pulse* (*and whose end was at* $t - 1$). Now, looking back over the construction of this section, we see that the nets for a, b, c, etc. do strongly recognize the single-letter sequences. The OR net (trivially) strongly recognizes the 'v' of the sets in question. And the definition of strong recognition describes precisely just what it is that the nets for EF and E* recognize.

It is interesting to observe that if we tried to work without the notion of strong recognition, it would be very hard indeed to prove directly that these nets do correctly recognize the appropriate regular sets in the ordinary (weak) sense, in which only one start pulse is ever permitted. We see here another instance of that mathematical phenomenon in which it is easier to prove a more general theorem than a weaker one.



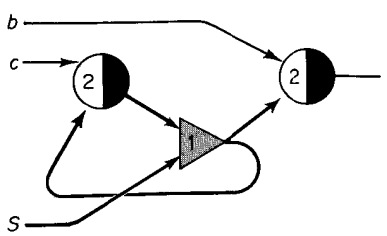


Fig. 4.4-6

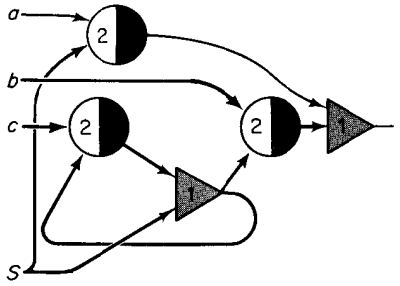


Fig. 4.4-7

EXAMPLE

We realize a net for the expression

$$(a \vee c*b)*$$

The net for $c*$ is shown in Fig. 4.4-5. Figure 4.4-6 gives the net for $c*b$; and Fig. 4.4-7, the net for $a \vee c*b$. The complete net for $(a \vee c*b)*$ is shown in Fig. 4.4-8.

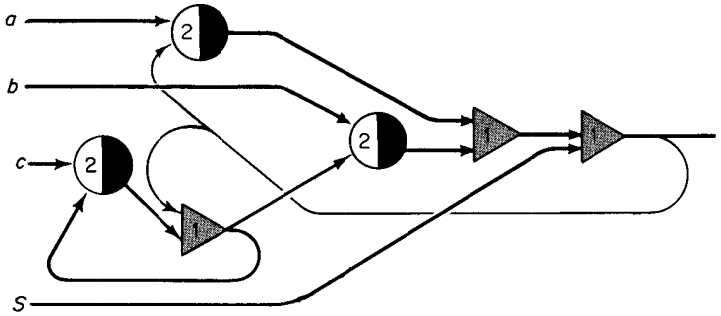


Fig. 4.4-8

4.4.2 Elimination of the instant-OR cells

Now we will eliminate the instant-OR cells from the system. To do this, we look to see how the cells of the net obtain their input pulses. Consider any threshold-2 cell in the net. *From the way the net was constructed, we know that every cell has just two input fibers.* The upper fiber is one of the single-letter input lines. (Indeed, we see from the construction method that there is precisely one such cell for each letter-occurrence in the original regular expression.)

The lower input of a threshold-2 cell can arise in three ways. It may come from

(1) the start fiber of the whole net,
(2) the output line of another 2 cell,
(3) the output line of an OR cell.

Signals arriving from OR cells are in effect transmitted directly from the inputs of those cells. An input to an OR cell is again either

(1) the start fiber,
(2) the output of a 2 cell, or
(3) the output line of another OR cell.

Suppose we choose a 2 cell and work back through the net to find all sources of pulses that may arise and instantly reach its lower input, ignoring the OR cells along the way. A list of these sources may include (1) the start fiber and (2) a certain collection of 2 cells. (We do not include OR cells in this list because they don't initiate signals, but only transmit them.) For example, in our example Fig. 4.4-8, the cell with upper input a receives lower input signals instantly from S and from the cells for a and b. The cell with upper input b receives signals directly from S, a, b, and c; the same is true for the cell with upper input c.

The upper input to a cell corresponds simply to the occurrence, in the input message, of the signal letter associated with the cell. The lower input is excited (through the instant-OR network) if any of the events in the above-mentioned list occur—in the case of the a cell, these are the firing of the S fiber, of cell a, or of cell b. So far as the a cell is concerned, we could thus replace the rest of the net as in Fig. 4.4-9. By the same reasoning, we replace the whole net by one in which the gathering-together function of the OR cells is condensed into *one* OR *cell for each 2 cell* (plus one to collect outputs) as in Fig. 4.4-10.

To see that this net behaves precisely the same as that of Fig. 4.4-8, one has only to verify that the conditions for the firing of each 2 cell are the same in both nets, due to the way we constructed the lists of firing conditions.

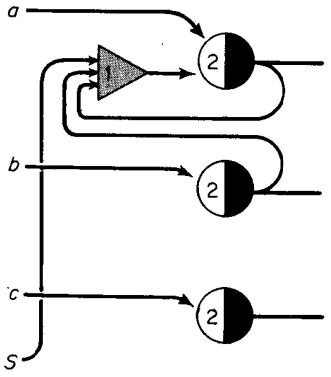


Fig. 4.4-9

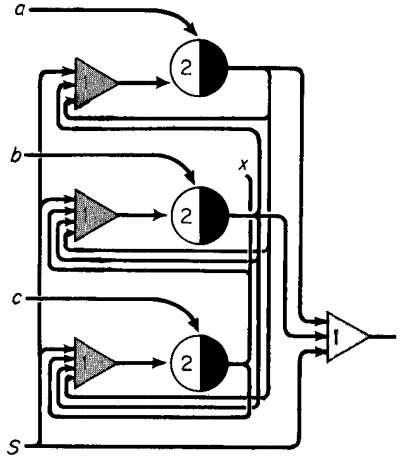
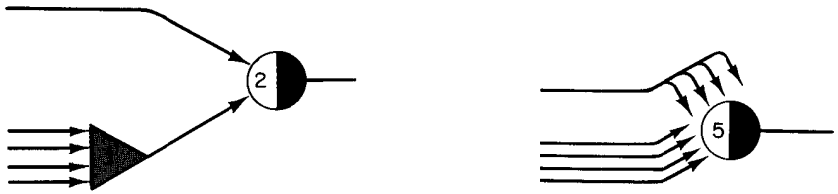
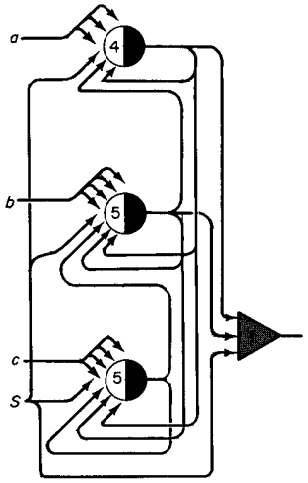


Fig. 4.4-10



The new net has no advantage in wiring simplicity or in numbers of cells, but it does have a decided advantage in orderliness. It shows that we may realize any regular expression by a net made up entirely of paired cells like that shown in Fig. 4.4-11. Now we can move to eliminate the OR cells! Observe that the network of Fig. 4.4-11 is precisely equivalent to the single McCulloch-Pitts cell in Fig. 4.4-12.



In each case, the output cell will fire if and only if the upper fiber is fired *and* one or more of the lower fibers is fired. No other combination of the fibers will produce an output. Similarly, in the general case of n lower inputs we need only set the threshold to $n + 1$, and give the upper fiber n branches. If we do this for the net of our example, we obtain the net of Fig. 4.4-13.

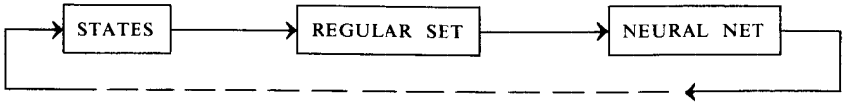
The result is that we are able to eliminate all the instant-OR cells from the net, except for the one required for the output line. We can certainly replace the terminal instant-OR cell by an ordinary McCulloch-Pitts OR cell; this introduces an extra terminal delay of one time unit. So we can say that

> any regular set can be recognized by a McCulloch-Pitts network with possibly a delay of one extra time unit.

There can be no way to eliminate the terminal OR unit, without an extra delay, if there is a fiber leading directly from the start fiber to the output, for we are not allowed to tie fibers together directly. (There is some question about what such connections could mean. Indeed, in Kleene's original discussion this was circumvented by defining '$*$' to be a binary connective so that an expression like $E*$ could not occur except within expressions of the form $E*F$.) See Problem 4.4-1.

This completes the proof that, *given any regular expression E, we can find a McCulloch-Pitts network in which a certain cell fires (with delay 1) exactly when a sequence of E has occurred.*

To complete the circuit of ideas



we have to show that any event recognized by firing a cell in a neural net can be recognized by an event in a state diagram. There is one slight snag in the argument; it may not be possible to represent the event by the occurrence of a *single* state of a state diagram. After all, the firing of a particular cell in a net does not correspond to a single state of the net, because other cells may or may not be firing concurrently. But we can represent the event by

> the occurrence of any of the states that entails firing of the output cell.

This means only that we must connect, with '$\vee$', the regular expressions of each of those states to obtain the appropriate regular expression for that cell's firing. (See problem 4.5-4); and this completes our circuit of proofs!

PROBLEM 4.4-1. If there is no direct start-output line, then the firing of the final output is contingent on the firing of any one of a certain collection of other cells of the net. Show how to add to the net a new cell which fires whenever any one of that collection fires, and at the same time (rather than one moment later). This is done by duplicating on this one cell all the connections that can excite any of the other cells, and making suitable adjustments of its fiber endings and threshold.

PROBLEM 4.4-2. Rebuild the theory using a different notion—that of regular$^\circ$ sets—in which one defines a regular$^\circ$ set by using a new interpretation of $*$: The expression E° represents *one or more* consecutive occurrences of sequences from the set represented by E. Then show that the sets of sequences defined by the regular$^\circ$ expressions differ from the sets of sequences defined by the regular expressions only in the handling of the null sequence. Indeed any regular set is identical to some regular$^\circ$ set either immediately or by addition of *null*. Thus, for the examples of section 4.2 we have the following equivalences:

(1) $a*a = a^\circ$
(2) $(a \vee b \vee c)*a = (a \vee b \vee c)^\circ a \vee a$
(3) $a(ba)*b = (ab)(ab)* = (ab)^\circ$
(4) $(a \vee bcd)*bcd = (a \vee bcd)^\circ bcd \vee bcd$

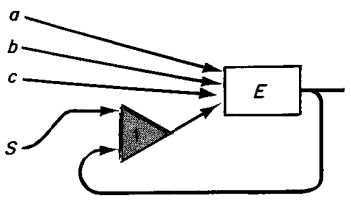


Fig. 4.4-14

and we add for further illustration

$$(ab)* = (ab)^{\circ} \vee \text{null}$$

In building up the nets for regular° sets, one can use for E° the net in Fig. 4.4-14. Then there can be no fiber leading from start to output, and the problem discussed above does not arise.

PROBLEM 4.4-3. Study the mechanism that saves these nets from becoming confused by large numbers of start pulses. Obviously, no finite machine can keep separate records of more than a limited number of start pulses. If this number is exceeded, but the machine continues to function properly, it must be because the machine either (1) has forgotten some old start pulses or (2) identifies several with the same equivalence class of histories. Thus, given the set represented by

$$(ab \vee a)* \vee (ba \vee b)*$$

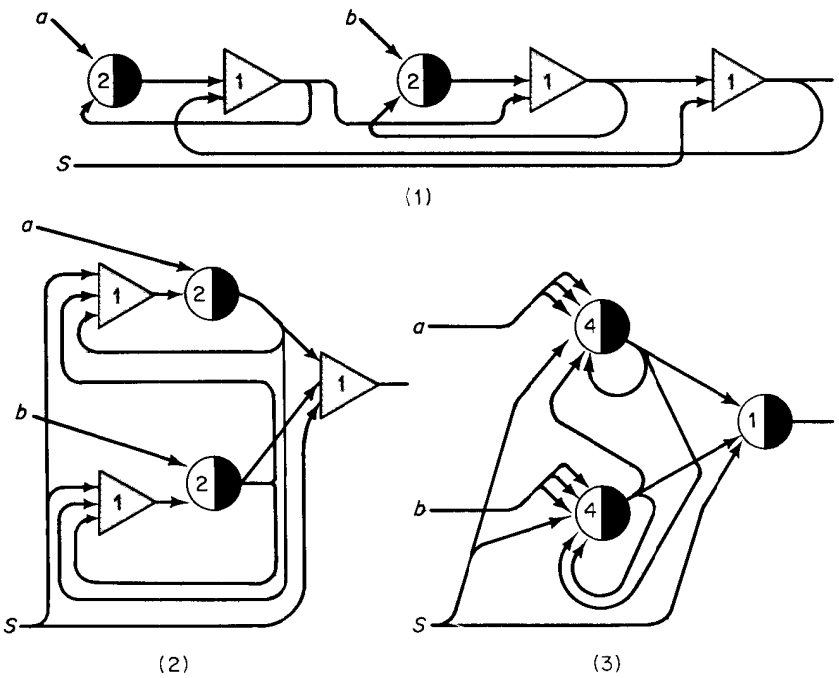


Fig. 4.4-15. Stages in the construction of $(a*b*)*$: (1) net with INSTANT-OR cells, (2) collecting the inputs, (3) replacement with McCulloch-Pitts cells.

the sequence

							ˇ		ˇ	
a	*b*	*a*	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*a*
1	2	3	4	5	6	7	8	9	10	11

is a member of that set with respect to start pulses at times 1, 3, 4, 6, 7, 8, 9, 10 and 11. But 1, 3, 4, 6, 7, 9, and 11 play one role, while 8 and 10 play another with respect to the equivalence classes of histories. The start pulses at times 2 and 5 may as well be forgotten—the machine's state should not show any trace of their having occurred. In working this out, one finds that *the (accessible) states of the network machine correspond to certain subsets of occurrences of the letters of the regular expression itself.* Using this, one could go directly from a regular expression to the state-transition table of an equivalent finite-state machine, without going through our construction of an intermediate McCulloch-Pitts network.

AN EXAMPLE

If our rules are applied to the expression $(a*b*)*$, we obtain the stages in the construction shown in Fig. 4.4-15. As is shown, when the inductive construction is finally completed, we draw together all the output fibers at the input of a final common OR cell. This is not the best possible net for recognition of the set $(a*b*)*$. In fact the net of Fig. 4.4-16 will suffice.

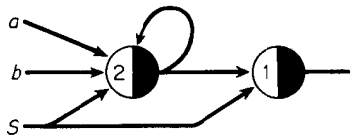


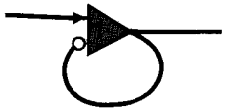
Fig. 4.4-16. Simpler net for $(a*b*)* = (a \vee b)*$.

It is interesting to note that no inhibitory connections appear in any of these nets. This is because the signals are decoded at the start—distinct stimuli occur as single pulses on distinct fibers. The situation is the same as that in section 3.4; the absence of non-monotonic elements is something of an illusion since they are really implicit in the input source decoder.

PROBLEM 4.4-4. It can be seen that these nets have the "strong recognition" property discussed in section 4.4.1—that for each regular set of sequences, the corresponding net will recognize any sequence of that set which occurs between *any* start pulse and the present time (minus 2). As noted in problem 4.4-1, the states of the net, vis-a-vis the start pulses, correspond to certain subsets of letters in the regular expression. Perhaps the best way to see this is to ask oneself: what is the meaning, with respect to the process history, of the firing of one of the cells in the net? Observe that each cell corresponds uniquely to a letter occurrence in the regular expression. Conclude that the firing pattern in the net is just what is needed to represent the essential features of the history of start pulses.

PROBLEM 4.4-5. Argue that there can really be no trouble resulting from allowing the instant-OR cells of section 4.4.1, so long as there are no in-

hibitory connections allowed in the net. Furthermore, no trouble could occur even with instantaneous inhibitory connections provided that there exist no closed signal loops without any delay. The rules for net formation given in Copi, Elgot, and Wright [1958] ingeniously prevent the construction of any logical paradoxes such as that represented by the net in Fig. 4.4-17, which uses a deadly combination of non-monotonic and delay-free elements. (Cf. section 3.1, remark 3). One cannot assign any consistent meaning to the response function of this net.



4.4.3 Remarks on regular sets

The regular sets of sequences are essentially the classes of histories that can be represented by the states of finite machines. Looking at finite machines in this way gives us some insight, but not really complete understanding. One thing we can see is that infinite sets can be obtained only through the '$*$' operation, i.e., through an obscure sort of periodicity. But one must not jump to the conclusion that infinite sets need nets with loops. Any *finite* set of sequences is (trivially) regular and realized by nets without loops. It is clear, from the definition of regular set, that one can add any finite number of terms to any regular set and still obtain a regular set. It is much more difficult to deduce *from the definition* that one can similarly *remove*, say, one regular set from another and have a regular set as the result. But this fact becomes obvious in the light of the equivalence just established, since the machine of Fig. 4.4-18 will do just that (with an extra delay). Clearly, this machine will respond to (recognize) each sequence which is in E but not in F.

Furthermore, if E and F are regular, we can see by the same kind of construction that the set '$E \wedge F$' of sequences common to both E and F (Fig. 4.4-19) and the set E' of finite sequences *not* in E (Fig. 4.4-20) are also regular. (There may be some fine points about delays in the last net.) This suggests that one might find it easier to understand finite-state machines in terms of an extended Kleene algebra including *and* ('$\wedge$') and *not* ('$'$'). For example, we can deduce that

$$(ab)' = (b \vee aa)(a \vee b)*$$

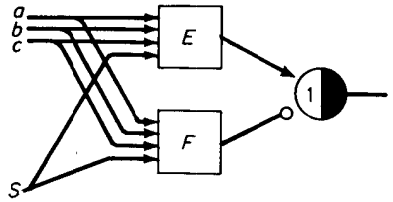


Fig. 4.4-18. E AND NOT F.

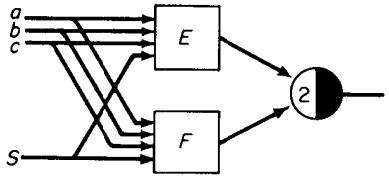


Fig. 4.4-19. E AND F.

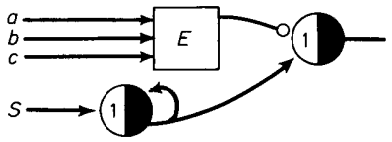


Fig. 4.4-20. NOT E.

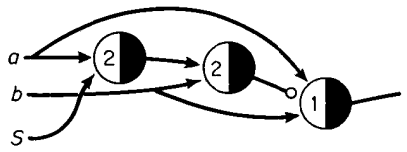


Fig. 4.4-21

Now in the Kleene algebra, this infinite set requires at least one star to represent it, but in the extended algebra it doesn't. The net of Fig. 4.4-21 represents it without any closed signal paths. Questions such as how many stars are needed to represent certain kinds of sets, are quite difficult.[3]

4.4.4 Epistemological consequences

In their classic paper, McCulloch and Pitts [1943] make some observations about the consequences, for the theory of knowledge, of the proposition that the brain is composed of basically finite-state logical elements. And even though this may not be precisely the case, the general conclusions remain valid also for any finite-state or probabilistic machine, presumably including a brain. The observations are: (1) the inclusion of disjunctive relations (the OR or 'v' connective) means that the description of a previous state cannot be completely determined from the description of the present state, and (2) the cyclic activity (from the presence of cyclic loops) makes it impossible to determine just when, in the past, the initial stimulus event occurred.

> "This ignorance, implicit in all our brains, is the counterpart of the abstraction which renders our knowledge useful."[†]

The concluding sections of their paper sketch out the possibility of a systematic investigation of the relation between nervous structure and the neurology of normal and diseased minds.

4.5 PROBLEMS

PROBLEM 4.5-1. Reconsidering problem 4.3-6, try to prove, using only reasoning about state diagrams, that if S is the set of sequences that carry some machine M from state Q_0 to Q_1, then there is a machine that so recognizes the set of the sequences of S taken in reverse. Finding the direct construction of this reversed-sequence recognizer without using arguments about regular expressions is difficult but instructive enough to be worth

[†]Op. cit., p. 131.

the effort. A solution method for a related problem is found in Shepherdson [1959].

PROBLEM 4.5-2. There is an analogy between regular expressions and algebraic operations– for example, those used by Mason [1960] for electric network theory. For example, replace $a*$ by

$$1 + a + a^2 + a^3 + \ldots = \frac{1}{1 - a}$$

and make multiplication non-commutative—that is, do not permit $xy = yx$ exchanges. Then we obtain rule (9) of problem 4.3-4 as follows:

$$\begin{aligned} E*F &\approx \frac{1}{1 - e} \cdot f \\ &= \frac{1}{1 - e} \cdot [e + 1 - e] \cdot f \\ &= \left[1 + \frac{e}{1 - e}\right] \cdot f \\ &= f + \frac{1}{1 - e} \cdot e \cdot f \approx F \vee E* EF \end{aligned}$$

Think about this and see if you can find reasons why this is so, and (if you are familiar with "signal flow graphs") what is the connection with electrical networks. Of course, there is an algebraic fact:

$$1 + a + a^2 + \ldots = 1 + a[1 + a + a^2 + \ldots] = \frac{1}{1 - a}$$

For further developments along this line, see Haring [1960], and Ott and Feinstein [1961],

PROBLEM 4.5-3. Show that

$$(a*ab \vee ba)*a* = (a \vee ab \vee ba)*$$

PROBLEM 4.5-4. Find a regular expression E for the sequences that bring the machine in Fig. 4.5-1 back to its starting state. The find a regular expression E^R for the set of sequences each of which is the reverse of a sequence in E. Now draw a state diagram to represent E^R. You will find that you can't do it if you require a single state to represent the event. Prove this.

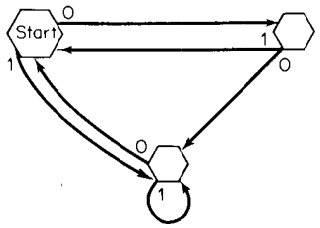


Fig. 4.5-1

PROBLEM 4.5-5. (Solutions to these can be based on the methods of section 2.6).

Which of the following sets of sequences (1–5) can be recognized by a finite state machine? All inputs are 0 or 1.

(1) The set of all sequences, 0, 1, 00, 01, 10, 11, 000, ...
(2) The numbers 1, 2, 4, 8, ..., 2^n, ... written in binary notation.
(3) The same set in unary: 1, 11, 1111, 11111111, ...
(4) The set of sequences in which the number of 0's is equal to the number of 1's.
(5) The sequences 0, 101, 11011, ..., 1^n01^n, ...

(6) If E is the set of sequences recognized by a machine M, is there another machine that recognizes any sequence *ending* with one that M recognizes?
(7) If E is the set of sequences recognized by machine M, is there a machine that recognizes any sequence *containing* one that M recognizes?
(8) If E is the set of sequences recognized by machine M, is there a machine that recognizes any two consecutive occurrences of the same sequence of E?

PROBLEM 4.5-6. Replace each of the following by regular expressions that do not use the v.

$$(a \vee b)* \qquad (a \vee bb \vee ba)* \qquad (a \vee (bb \vee ab)*)*$$

If E is any regular expression, can $E*$ always be written without any v's? Why?

NOTES

1. We thus avoid a number of fine points which account for some of the complexity of Kleene's [1956] paper. We assume that the machine is started in operation by the injection of an S pulse into an otherwise quiescent net. That is, we assume that the machine really did start at some distinct (though unknown) moment in the finite past. Otherwise, in the net of example 1, it could be that the R cell was *always* firing, has *always* received a signals, and need never have seen an S pulse. The significance of this awesome (but perfectly logical) possibility is discussed in the original McCulloch-Pitts [1943] paper.

2. Kleene [1956] defines all three operations, *or*, *star*, and *followed-by* as binary operations. So do we, for *or* and *followed-by*, but where Kleene allows only the form $E*F$, we allow $E*$ to mean concatenating any number of selections from E, including *none*. This means we have to think about the "null sequence" and some resulting complications. In Copi, Elgot, and Wright [1958], $E*$ doesn't contain the null sequence. Their proof of Kleene's theorem makes a clean separation of the "logical" properties of cells from the delay properties, and our first construction using "instant-OR" cells follows these lines; our final construction brings the result back closer to Kleene's.

3. *On the extended Kleene Algebra.*
A remarkable set of equivalence theorems has been established recently by S. Papert and R. McNaughton. Consider the class of "extended" regular

expressions obtained by adding *not* and *and* to the *or*, *star*, and *juxtaposition* of the Kleene Algebra. (This was proposed first in McNaughton and Yamada [1960].) As we showed in section 4.3, one does not obtain anything new by this, because we still lie in the domain of finite-state machines, as shown by Figs. 4.4-19 and 4.4-20. What we want to point out is that, according to Fig. 4.4-20, while we may need to introduce a loop to obtain *not*, *we do not require anything more than a loop around a single cell.* Clearly, if we confine ourselves to extended regular expressions that contain no stars, we can obtain McCulloch-Pitts nets that have only single-cell loops of this kind. It can be shown that the converse is true—that nets with only this kind of loop can be described by star-free extended regular expressions.

It turns out that machines of this class cannot "count" cyclically; for example, they cannot even achieve the behaviors in section 3.2.4 that determine whether the number of 1's in the input is even or odd. Let us give a more general definition of non-counting.

> A regular set E in a *non-counting* set if there exists a number n such that, for all strings U, V, and W, and all positive integers p, if UV^nW is in E then so is $UV^{n+p}W$.

It is easy to show that any star-free E is non-counting; it is much more difficult to show the converse, which is also true. This theorem points toward a connection between the formulations we have been considering and a rather different way of looking at machines, namely, the approach through semigroups.

The connection between the theory of automata and the theory of semigroups is very close. (We have put these remarks in the Notes so as not to put off the reader unacquainted with groups and semi-groups; see, for example Rabin and Scott [1959].) In the semigroup formulation, one thinks of a machine as having a set of states, which are transformed into one another by the input signals. In some machines, an input signal may only permute the states around—that is, no input signal drives two different states into the same state. Clearly, if this is the case, that machine is capable of some cyclic counting ability, for there must be a non-trivial loop in the state diagram. From the group-theory point of view, the existence of a permutation induced by a signal on some of the states means that there is a subgroup in the semigroup of transformations (on the states) generated by the input signals. The important theorem here is that a machine that is "permutation-free" in this sense is a star-free machine, and conversely; and it turns out that this in turn is equivalent to the class of machines that can be constructed out of the "unit automata" of Krohn and Rhodes [1963]. The advanced reader will want also to see the work of Schutzenberger [1965].

Finally, there is a connection with still a different mathematical family of concepts—the predicate calculus of quantified propositions. Consider an alphabet $X = \{x_1, \ldots, x_r\}$. For each i we define the propositional function $F_i(t)$ which asserts that the t^{th} input is x_i. Let $t_1, t_2, \ldots, t_s$ be a set of integer valued variables and let P be the class of expressions defined by:

(1) $F_i(t_j) \in P$ all i, j
(2) $t_j \leq t_k \in P$ all i, k

(3) Any propositional form in members of P is in P.
(4) If $A \in P$ and A contains the free variable t_i, then $(t_i)A$ and $(\mathsf{E}t_i)A$ are in P.

Let P_0 be the set of members of P with no free variables. Then P_0 defines the class of words on X which satisfies it.

A regular set E is in L if it can be defined as the set of words satisfying a predicate calculus expression, P_0, defined as above. The class L is the "L-languages" of McNaughton [1960] and these too, turn out to be equivalent to the star-free machines. This theory is developed in the paper of Papert and McNaughton [1966].

PART II

INFINITE MACHINES

5 COMPUTABILITY, EFFECTIVE PROCEDURES, AND ALGORITHMS. INFINITE MACHINES.

5.0 INTRODUCTION

We now turn our attention to some basic questions centered around the very notion of a mechanical process. What can a machine do? What does it mean to say that a process is mechanical? When is a procedure so completely specified that a machine can carry it out? In earlier chapters we explored the limitations of machines with finite memory. What happens when we lift this restriction? What problems can be solved by machines—by mechanical processes—with unlimited memory? Are there processes that can be precisely described yet still cannot be realized in a machine?

As we noted in chapter 1, most people have a low opinion of the intellectual potentialities of machines. It is usually felt that although machines can be very fast, or very strong, they cannot be very smart. It is well known that machines have been made to do many things that meet high human standards—to play games very well, to find solutions to mathematical problems of college-grade difficulty, to find solutions to difficult systems design problems, to classify visual patterns of appreciable complexity. But it is usually felt that this reflects no credit on the machine—that because the designer or programmer has set down every small detail of the process, the machine has only to perform (however quickly and accurately) simple clerical tasks.

It is certainly true that *programming*—the job of specifying the procedure that a computer is to carry out—amounts to determining in advance everything the computer will do. In this sense, a computer's *program* can serve as a *precise description* of the process the machine will

carry out, and in this same sense it is meaningful to say that *anything that can be done by a computer can be precisely described.*†

We often hear a kind of converse statement to the effect that "*any procedure which can be precisely described can be programmed to be performed by a computer.*" I have heard this and similar statements made on many occasions, and the proposition is usually stated to be a consequence of the work of the mathematician Alan M. Turing. But it is not usually stated exactly what it is that Turing proved; in particular it is not made clear what was his notion of "precisely described." Turing's concept of a precise description of a process is essential to the following chapters, so we have to explain it in some detail.

5.1 THE NOTION OF EFFECTIVE PROCEDURE

Our exploration of machines in Part II is based, in large part, on ideas derived from the paper of Turing [1936] on the theory of computability. This paper is significant not only for the mathematical theory which concerns us here, but also because it contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it. While it is often said that the 1936 paper did not really much affect the *practical* development of the computer, I could not agree to this in advance of a careful study of the intellectual history of the matter.

Turing's paper must be viewed against the intellectual background of a variety of ideas concerning descriptions and processes. Again we think of a collection of questions. *What processes can be described*? Surely the notion of description entails some *language*. Could any one fixed language admit description of *all* describable processes? Can there be processes which are, somehow, well defined, yet cannot be described at all? It could be argued that there might exist definite processes whose communication requires the transmission of a mental attitude, or a disposition, which cannot be captured in any finite number of words—which must remain intuitive.

†It is important to note that this does not mean that the person who writes a computer program automatically understands all the consequences of what he has done! It is perfectly possible to write instructions which launch the computer into a great search process, with many of the trial-and-error features of organic evolution, and the consequent development of structures of enormous and unexpected complexity. Many computer programs are frankly experimental—to see what will be the behavior of a specified system—not well understood in advance. All scientists know that specification does not mean immediate understanding. When we write down equations, e.g., in mathematics or physics, it is not enough to know that the solutions are thus determined. We usually want to know something about the character of the solutions—and this entails some kind of process called "solving the equations."

Such questions had concerned mathematicians for some time before the advent of computing machines. These questions are associated with the idea of an *algorithm*—an *effective procedure*†—for calculating the value of some quantity or for finding the solution of some mathematical problem.

The idea of an algorithm or effective procedure arises whenever we are presented with a set of instructions about how to behave. This happens when, in the course of working on a problem, we discover that a certain procedure, if properly carried out, will end up giving us the answer. Once we make such a discovery, the task of finding the solution is reduced from a matter of intellectual discovery to a mere matter of effort; of carrying out the discovered procedure—obeying the specified instructions.

But how does one tell, given what appears to be a set of instructions, that we really have been told exactly what to do? How can we be sure that we can henceforth effectively act, in accord with the "rules," without ever having to make any further choice or innovation of our own?

> This question is easily answered if the process is supposed to terminate in a certain finite, already known, time, because then we can just try it and see. But if the length of the process isn't known in advance, then "trying" it may not be decisive, because if the process does go on forever—then at no time will we ever be sure of the answer. Our concern here is not with the question of whether a process terminates with a correct answer, or even ever stops. Our concern is whether the next step is always clearly determined. The other questions will come up in chapter 8.

The position we will take is this: If the procedure can be carried out by some very simple machine, so that there can be no question of or need for "innovation" or "intelligence," then we can be sure that the specification is complete and that we have an "effective procedure." We expect no quarrel with this. But we will also maintain, with Turing, a sort of converse, which will seem at first quite extreme. We assert that any procedure which could "naturally" be called effective, can in fact be realized by a (simple) machine. Although this may seem extreme, the arguments below in its favor are hard to refute.

We must emphasize that this is a subjective matter, for which only argument and persuasion are appropriate; there is nothing here we can expect to *prove*. It is not necessary to accept (or even to read) the following arguments, to appreciate the mathematical development of the sequel. The reader who finds himself in strong disagreement either intellectually or (more likely) emotionally should not let that keep him from appreciation of the beautiful technical content of the theory developed further on.

†We will use the latter term in the sequel. The terms are roughly synonymous, but there are a number of shades of meaning used in different contexts, especially for "algorithm."

5.1.1 Requirements for a definition of effective procedure

In trying to give a precise mathematical definition to "effective procedure," one encounters various difficulties. (One cannot expect always to find a simple and completely satisfactory formal equivalent for a complex intuitive notion.) We will begin by saying that

> an effective procedure is a set of rules which tell us, from moment to moment, precisely how to behave.

This attempt at a definition is subject to the criticism that the *interpretation* of the rules is left to depend on some person or agent. Now a person's ability to obey instructions depends on his background and intelligence. If his intelligence is too small, he may fail to understand what we mean. If his intelligence is too large, or too alien, he may invent some consistent interpretation of the rules that was not intended. We know how often an apparently "simple" explanation turns out to have an unsuspected ambiguity.

We could avoid the problems of interpretation—of understanding—if we could specify, along with the statement of the rules, *the details of the mechanism that is to interpret them.* This would leave no ambiguity. Of course, it would be very cumbersome to have to do all this over again for each individual procedure; it is desirable to find some reasonably *uniform* family of rule-obeying mechanisms. A most convenient formulation would be one in which we set up

> (1) a *language* in which sets of behavioral rules are to be expressed, and
> (2) a *single* machine which can interpret statements in the language and thus carry out the steps of each specified process.

This suggests designing some sort of machine to accept sets of rules, expressed in some language, and to do what the rules require. Here one might expect a new difficulty. For surely one would not expect any single mechanism to be powerful enough to interpret and execute rules for *all* effective procedures. Surely one would expect to need at least a sequence of more-and-more complex machines for the execution of a sequence of more-and-more complex procedures.

Curiously, this is no problem! It turns out that we can realize our notion of an instruction-obeying machine in a form which remains constant, no matter how complex the procedure in question. That is, we can set up a rules-language and a single "universal" interpretation machine which can handle *all* effective procedures. The detailed construction is found in chapter 8; the trick is to substitute merely quantitative increases in memory size for qualitative increases in complexity of the machine.

Although we state this as fact, there remains a subjective aspect to the matter. Different people may not agree on whether a certain procedure should be called effective. Perhaps there are processes, one might suppose, which simply *cannot* be described in any formal language, but which can nevertheless be carried out, e.g., by minds. One might even argue that those important, but still mysterious, functions which make minds superior to all presently known mechanical processes must, by their intuitive nature, escape any systematic description.

Turing discusses some of these issues in his brilliant article, "Computing Machines and Intelligence" [1950], and I will not recapitulate his arguments. They amount, in my view, to a satisfactory refutation of many such objections. We will put aside such matters and turn first to the problem of making precise the intuitive idea of obeying a set of instructions. Matters are clarified by confining attention first to very concrete processes, such as the execution of mathematical computations.

5.2 TURING'S ANALYSIS OF COMPUTATION PROCESSES

In his 1936 paper, A. M. Turing defined the class of abstract machines that now bear his name. A *Turing machine* is a finite-state machine associated with a special kind of environment—its *tape*—in which it can store (and later recover) sequences of symbols. We will describe these machines in greater detail in chapter 6 and the sequel. They are very simple. At each moment the (finite-state part of the) machine gets its *input* stimulus by *reading* the symbol written at a certain point along the tape. The *response* of the machine may *change that symbol* and also move the machine a small distance either way along the tape. The result is that the stimulus for the next cycle of operation will come from a different "square" of the tape, and the machine may thus read a symbol that was written there long ago. This means that the machine has access to a kind of rudimentary exterior memory in addition to that provided within its finite-state part. And since we will place no limit on the amount of tape available, this memory has, in effect, an infinite capacity. But the restricted manner in which the machine and its tape are coupled (see chapter 6) might make one think, at first, that the possible uses of this potentially infinite memory would be really very limited.

Turing discovered, however, that he could set these machines up to make very complex computations. In chapters 6 and 7 we will work out a number of examples and see that the tape memory really does escape the limitations of the finite-state machine.

Turing goes on to defend the following proposition, now often called *Turing's thesis*:

> Any process which could naturally be called an effective procedure can be realized by a Turing machine.

This proposition, in its most general form, is usually called *Church's thesis*, after the work of Alonzo Church relating the intuitive notion of effectiveness to formal logical processes. We refer to it as Turing's thesis because of our preoccupation, in the sequel, with Turing's particular formulation of computability concepts. Part of Turing's 1936 paper is concerned with demonstrating the equivalence of his and Church's prior formulation.

When one sees how primitive the machines really are, in concept, this thesis seems incredibly rash. Yet the years have borne out Turing's view. Every procedure which mathematicians have generally agreed to be "effective" has been shown equivalent, in one way or another, to a procedure carried out by a Turing machine.

One cannot expect to *prove* Turing's thesis, since the term "naturally" relates rather to human dispositions than to any precisely defined quality of a process. Support must come from intuitive arguments, and we could hardly do better than to present some of the arguments in Turing's own words.

5.3 TURING'S ARGUMENT

The following is taken verbatim from Turing [1936, section 9]. It is one of a sequence of arguments in favor of his position. The other arguments are reviewed clearly in Kleene [1952]. The machines discussed in the quotation are essentially those described in the next chapter—today called "Turing machines." Note that the word "computer" as used in the quotation means the *person* that Turing is going to replace by a machine.

> "Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent.[†] The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.
>
> "The behaviour of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused.[‡] Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.
>
> "Let us imagine the operations performed by the computer to be split up into 'simple operations' which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always 'observed' squares.

[†]If we regard a symbol as literally printed on a square we may suppose that the square is $0 < x < 1, 0 < y < 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. If these sets are restricted to be measurable, we can define the 'distance' between two symbols as the cost of transforming one symbol into the other if the cost of moving a unit area of printer's ink unit distance is unity, and there is an infinite supply of ink at $x = 2, y = 0$. With this topology the symbols form a conditionally compact space. [Turing's note].

[‡]In the second paragraph, we cannot understand what Turing could have meant by the suggestion that "states of mind" could be "confused," if he is discussing psychological matters. If, instead, he has in mind physical states of a brain, then one would indeed expect that for sufficiently similar states there will be a chance of random transitions, e.g., because of thermal or quantum phenomena. There is a limit to the amount of information that can be recovered from any physical system of limited size. The same holds for whatever physical system is used to represent the symbols within the "squares" of fixed dimensions. [M. M.]

"Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

"In connection with 'immediate recognisability,' it may be thought that there are other kinds of squares which are immediately recognisable. In particular, squares marked by special symbols might be taken as immediately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find '... hence (applying Theorem 157767733443477) we have ...'. In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other 'immediately recognisable' squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable.

"The simple operations must therefore include:

(a) Changes of the symbol on one of the observed squares.
(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

"It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.
(B) A possible change (b) of observed squares, together with a possible change of state of mind.

"The operation actually performed is determined, as has been suggested [above] by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation.

"We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an 'm-configuration' of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares.† The move which is done, and the succeeding configuration, are determined by the scanned symbol and the m-configuration. The machines just described do not differ very essentially from computing machines as defined (previously) and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer."

5.3.1 The equivalence of many intuitive formulations

Perhaps the strongest argument in favor of Turing's thesis is the fact that, over the years, all other noteworthy attempts to give precise yet intuitively satisfactory definitions of "effective procedure" have turned out to be equivalent—to define essentially the same class of processes. In the 1936 paper Turing proves that his "computability" is equivalent to the "effective calculability" of A. Church. A very different formulation of effectiveness, described at about the same time by Emil Post, also turned out to have the same effect; we show the equivalence of Post's "canonical systems" and Turing machines in chapter 14. Another quite different formulation, that of "general recursive function" due to S. C. Kleene and others is also equivalent, as we show in chapters 10 and 11. More recently a number of other formulations have appeared, with the same result (e.g., Smullyan's "Elementary Formal Systems" [1962]). Whenever a system has been proposed which is not equivalent to these, its deficiencies or excesses have always been intuitively evident.

Why is this an argument in favor of Turing's thesis? It reassures us that different workers with different approaches probably did really have the same intuitive concept in mind—and hence leads us to suppose that there is really here an "objective" or "absolute" notion. As Rogers [1957] put it:

> "In this sense, the notion of effectively computable function is one of the few 'absolute' concepts produced by modern work in the foundations of mathematics."

Proof of the equivalence of two or more definitions always has a compelling effect when the definitions arise from different experiences and motivations.

†As we shall see, there is no loss of generality if we restrict both L and B to be unity! [M. M.]

5.4 PLAN OF PART II

This second part of the book explores the properties of our extended machines—finite-state machines with a passive but unlimited environment of "scratch paper." This exploration continues to develop the two themes of the first part of the book. We try to assess the range of possible behaviors—to characterize what is within and outside the reach of these machines. And we continue to exercise our avocation—the collection of very small "universal bases" for the assembly of structures which realize the full range of behavior.

In both these avenues we now meet a much more extensive and exciting range of phenomena. One would certainly expect the range to be large, since we have argued that our new machines can encompass all effective procedures. For our hobby, we discover some remarkable bases. The reader will find it incredible, at first sight, that some of these sets of simple operations could give rise to the full range of possible computations.

For the more serious objective—that of characterizing the effective procedures—we acquire a remarkable tool, the *universal computing machine*. It turns out (in chapter 7) that there exists a certain Turing machine which can imitate the behavior of any other Turing machine, given an adequate description of the structure of that other machine. (The description is to be written down in the environment—tape—and need not be built into the works of the universal machine.) As a result, our exploration can be reduced to the study of this single machine. While we do not entirely so restrict our attention, we do frequently call upon the existence of such a machine throughout the sequel. The universal machine also opens the road toward simple bases.

The universal machine works, as one might expect, by operating on the *description* of another machine. It *interprets* such a description, one step at a time, so that, in effect, it imitates the other's behavior. With "interpretative behavior" within the scope of our machines, we can begin to ask new kinds of questions about machines. For instance, one can ask: what happens when a machine is confronted with its own description? If this is done to the universal machine, then, as one might expect, the machine must become paralyzed by an infinite regression of interpretation cycles—it can never actually get to compute anything. At first, such phenomena seem entertaining, then annoying, and finally we are forced to conclude that they signal a portentous obstacle to our exploration. *We find that certain of the questions we naturally ask about machines cannot be answered, at least by any effective procedure for answering questions.* Indeed, the most significant results of the second part of the book are *negative* results indicating limitations not only on the effective procedures

themselves, but also on our ultimate ability ever to characterize, effectively, which procedures are in fact effective.

Chapter 8 contains the beginnings of these results. The basic method is simple and rather striking. Because the machines are capable of interpretative operation, we can imagine passing certain of our questions over to the machines themselves. Such a question, for example, is: *Which machine computations eventually terminate with a definite result, and which go on forever without any definite conclusion?* We show, by some simple technical tricks, that assuming the existence of a machine that can answer this question leads to a contradiction; hence, no such machine can exist.

The non-existence of a certain kind of machine might not be, in itself, a great disaster. But so far as our goals here are concerned, this result is indeed very serious; no Turing machine can answer this question. But then, if we accept Turing's thesis, there can be no *effective* way at all to answer it. That is, we can never aspire to a complete, systematic theory of the conditions under which a computation is sure to terminate. We may be able to decide, for one reason or another, that certain machines will work, and that certain others will not, but we will never be able to put this art on a systematic basis which will work for *all* machines! I regard this, and similar other "undecidability" results, to be among the most significant intellectual discoveries of modern times. One can reject its implications only by rejecting Turing's thesis, but there is no apparent prospect of any satisfactory replacement.

With the demonstration of these elementary undecidability results, several new lines of exploration are opened (even if the most desirable line is thus irrevocably closed). We could try to restrict our machines so that their behavior is not quite so complicated; and we do this briefly in chapter 10, in connection with the so-called "primitive-recursive functions." Another line (that we do *not* follow) is to study, as if by default, the interrelations between various kinds of unsolvability results; this forms much of the subject matter of the modern mathematical "theory of recursive functions." Our main activity will be to examine a variety of alternative formulations of effectiveness and through this to try to understand something of the sources of what we know is a hopelessly complex range of behavior. In the course of this we will come to understand better the relation between the modern computer (and its programs) on the one hand and these abstract machines (and their descriptions) on the other.

Chapter 11 is concerned with finding a middle ground between the simple but impractically inefficient Turing machine, and the more complicated but application-oriented modern computer. Our result is a new family of abstract machines—called "program machines"—which combine attractive features from both extremes. It turns out that in almost every respect program machines are superior to Turing machines for

theoretical purposes; and, by studying them, we also obtain some practical insight both into the theory of effective computation and into the basis of modern computer programming. We do not wish to arouse wild hopes that this theory, by some magical insight, will help anyone directly with practical computer programming problems. But it does provide some of the "cultural background" from which a satisfactory theory of practical computation will ultimately spring.

Chapter 9, optional and somewhat more advanced than the rest, discusses some connections between the theory of effective procedures and the theory of real numbers.

5.5 WHY STUDY INFINITE MACHINES?

Up to this point, our study of machines has been based entirely on the finite-state point of view. We have used quite a few different arguments to justify this emphasis; even Turing's arguments earlier in this chapter are so directed.† In the sequel we will deal with machines of an infinite character; our machines will have infinite tapes, or storage registers of unlimited capacity. Since no such machine could exist (in a finite universe) and (even if the universe is infinite) we can never have one, why should we study their theory?

Our answer has a paradoxical quality. We shall not give the easy answer that, just as mathematicians study infinite numbers they cannot reach, it is instructive to study the limiting, inaccessible extension of our ideas about machines. On the contrary, we take the position that this extension is actually needed to gain any really practical insight into real-life computers! It is worth some discussion to see how the infinite-machine theory could be more realistic than the finite theory, for practical purposes.

In the first place, the limitations already established for finite-state machines seem much too restrictive to take seriously. Our intuitive ideas about machines require a more comprehensive framework. We ought to be able to talk about machines which can multiply pairs of arbitrarily large numbers (shown in chapter 2 to be beyond any finite-state machine) or can verify the grammatical correctness of arbitrary expressions in the simplest of mathematical languages. (The arguments of chapter 4 showed that this cannot be done by finite-state machines.)

To be sure, we will always be confined, in real life, to machines which are finite. But I assert that it is not always the *finiteness* of the machines that limits their uses; more usually it is either (1) the *practical limitations*

†The above quotation from Turing [1936] is, to my knowledge, the first clear description of a finite-state machine in the literature.

of running-time or (2) the conceptual complexity of their structures or "programs." Thus, we know that no finite machine can enumerate all the integers; it cannot count, in any reasonable sense, past its number of distinct accessible internal states. But that is not the limitation met in practice; even the smallest modern computer has thousands of bits of accessible memory, but for one actually to count up to 2^{1000}—even operating at the wave frequency of hard cosmic rays—would take longer eons than even our most cosmological astronomers like to consider. This makes suspect the practical guidance value of inferences based solely on the finite-state limitation.

A more serious practical limitation arises when we consider machines which consume a large information storage capacity in a relatively passive way. We might want to consider, for instance, a machine which remembers all of its previous input experience.† Even here the practical limitation rarely takes the form of an outright bound on machine size; it has instead the peculiar form of relentlessly increasing economic pressure. For, given a real machine, we can always extend it a bit more by adding parts (and hence states) in the form, say, of external storage tapes. The eventual termination of such growth will depend on more or less irrelevant circumstances surrounding the project rather than on any particularly natural bound on the machine size.

For modern computers, we can usually provide enough external memory for our problems. The computation time still usually remains as the practical limitation. This is aggravated by the fact that access to external memory (e.g., magnetic tapes) may take millions of times longer than operations within the central machine.‡ Even if this were not so (and one million times 2^{1000} is still like 2^{1000}, in practice), our difficulty is less that the machine ends up in a state-diagram loop than that we cannot wait until it does.

For such reasons, it would seem profitable to study the theory of machines in which the *amount* of machinery is not itself the limitation. But it would *not* be profitable, at least from our point of view, to study machines which are really infinite either in initial endowment or in effective speed of operation. Thus, it would seem unrealistic to consider a machine which, when started, already contains the correct answers to all answerable questions in English! Nor would it seem realistic to study a machine

†There are some who believe the human brain has this capacity, but I am inclined to believe that this is due to a combination of wishful thinking and misinterpretation of striking, but exceptional, incidents.

‡There remains a fundamental problem here, even if access times for external memories remained constant with increasing size. For as the memory size grows then the length of time, on the average, for the computer to compute the *address* in memory of a datum must grow, too, albeit much more slowly.

which could test in finite time, an infinite number of cases or hypotheses (and thus tell us whether or not Fermat's Last Theorem is true by examining all cases). A compromise seems inevitable; we must consider machines which have at each moment only a finite quantity of structure, but which are capable of being extended indefinitely as time goes on—"growing machines."

The Turing machine, with its finite-state computing unit and its initially "almost-blank" tape, is perfectly suited to our purposes. We need not think of the machine's tape as infinite. We imagine instead that the machine begins with a finite tape, but that, whenever an end is encountered, another unit of tape (a "square") is attached. Thus, instead of an infinite tape, we need only an inexhaustible tape factory. Since, in fact, there is a fixed bound on the rate at which new tape can be required —one square per moment is the worst case—the factory can be maintained by investing a fixed, finite amount of money in a (perfectly) reliable bank. This picture gives a reassuringly finite picture of the new study. Accordingly, although we are done with the study of finite automata, we need not jump directly to the study of infinite automata. Instead we will work in what Burks [1959] aptly calls the domain of "growing automata." We note in passing that mathematicians do indeed study genuinely infinite automata, e.g., in the theory of the "hyperarithmetic functions" of Kleene [1952].

6 TURING MACHINES

6.0 INTRODUCTION

A Turing machine is a finite-state machine associated with an external storage or memory medium. This medium has the form of a sequence of *squares*, marked off on a linear *tape*. The machine is coupled to the tape through a *head*, which is situated, at each moment, on some square of the tape (Fig. 6.0-1). The head has three functions, all of which are exercised in each operation cycle of the finite-state machine. These functions are: *reading* the square of the tape being "scanned," *writing* on the scanned square, and *moving* the machine to an adjacent square (which becomes the scanned square in the next operation cycle).

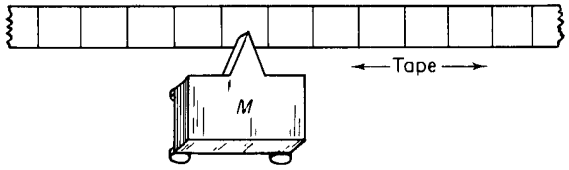


Fig. 6.0-1

It will be recalled from section 2.2 that a finite-state machine is characterized by an alphabet $(s_0, \ldots, s_m)$ of input symbols, an alphabet $(r_0, \ldots, r_n)$ of output symbols, a set $(q_0, \ldots, q_p)$ of internal states, and a pair of functions

$$Q(t + 1) = G(Q(t), S(t))$$
$$R(t + 1) = F(Q(t), S(t))$$

which describe the relation between input, internal state, and subsequent behavior.

In order to attach the external tape, it is convenient to modify this description a little. The input symbols $(s_0, \ldots, s_m)$ will remain the same, and it will be precisely these that may be inscribed on the tape, one symbol per square. The input to the machine M, at the time t, will be just that symbol printed in the square the machine is scanning at that moment. *The resulting change in state will then be determined, as before, by the function G. The output of the machine M has now the dual function* of (1) *writing on the scanned square* (perhaps changing the symbol already there) and (2) *moving the tape one way or the other.*

Thus R, the response, has *two* components. One component of the response is simply a symbol, from the same set $(s_0, \ldots, s_m)$, to be printed on the scanned square; the second component is one or the other of two symbols '0' (meaning "Move left") and '1' ("Move right"), which have the corresponding effect on the machine's position. Accordingly, it is convenient to think of the Turing machine as described by *three* functions

$$\begin{aligned} Q(t+1) &= G(Q(t), S(t)) \\ R(t+1) &= F(Q(t), S(t)) \\ D(t+1) &= D(Q(t), S(t)) \end{aligned}$$

where the new function 'D' tells which way the machine will move.

In each operation cycle the machine starts in some state q_i, reads the symbol s_j written on the square under the head, prints there the new symbol $F(q_i, s_j)$, moves left or right according to $D(q_i, s_j)$, and then enters the new state $G(q_i, s_j)$.

When a symbol is printed on the tape, the symbol previously there is erased. Of course, one can preserve it by printing the same symbol that was read, i.e., if $F(q_i, s_j)$ happens to be s_j. Because the machine can move either way along the tape, it is possible for it to return to a previously printed location to recover the information inscribed there. As we will see, this makes it possible to use the tape for the storage of arbitrarily large amounts of useful information. We will give examples shortly.

The tape is regarded as infinite in both directions. But we will make the restriction that *when the machine is started the tape must be blank, except for some finite number of squares.* With this restriction one can think of the tape as really finite at any particular time but with the provision, whenever the machine comes to an end of the finite portion, someone will attach another square.

Formal mathematical descriptions of Turing machines may be found in Turing [1936], Post [1943], Kleene [1952], Davis [1958]. There are unimportant technical differences in these formulations. For our purposes it will usually be sufficient to use pictorial state diagrams. Our immediate

purpose is to show how Turing machines, with their unlimited tape memory, can perform computations beyond the capacity of finite-state machines; it is usually easier to understand the examples in terms of diagrams than in terms of tables of functions. While it is fresh in our minds, however, let us note that the finite-state parts of our machines can be described nicely by sets of *quintuples* of the form

(old state, symbol scanned, new state, symbol written, direction of motion)

i.e.,

$$(q_i, s_j, G(q_i, s_j), F(q_i, s_j), D(q_i, s_j))$$

or

$$(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$$

i.e., as quintuples in which the third, fourth, and fifth symbols are determined by the first and second through the three functions G, F, and D mentioned above.†

Thus a certain Turing machine (section 6.1.1 below) would be described by the following six quintuples:

$$\begin{array}{lllll} (q_0, & 0, & q_0, & 0, & R) \\ (q_0, & 1, & q_1, & 0, & R) \\ (q_0, & B, & \text{HALT}, & 0, & -) \end{array} \qquad \begin{array}{lllll} (q_1, & 0, & q_1, & 0, & R) \\ (q_1, & 1, & q_0, & 0, & R) \\ (q_1, & B, & \text{HALT}, & 1, & -) \end{array}$$

or just

$$\begin{array}{lllll} (0, & 0, & 0, & 0, & 1) \\ (0, & 1, & 1, & 0, & 1) \\ (0, & B, & H, & 0, & -) \end{array} \qquad \begin{array}{lllll} (1, & 0, & 1, & 0, & 1) \\ (1, & 1, & 0, & 0, & 1) \\ (1, & B, & H, & 1, & -) \end{array}$$

where we have reserved the symbol 'H' (or 'HALT') to designate a halting state.

One more remark. When we dealt with finite-state machines and the things they could do, we had to regard the input data as coming from some *environment*, so that the description of a computation was usually not contained completely in the description of the machine and its initial state. With a Turing machine tape we have now a *closed* system, for the tape serves as environment for the finite-state machine part. *Hence we can specify a "computation" completely by giving* (1) *the initial state of the machine* and (2a) *the contents of the tape.* Of course we have also to say (2b) *which square of the tape the scanning head sees at the start.* We will usually assume the machine starts in state q_0.

†The state denoted by q_{ij} is defined to be that one of the q_i's given by the function $G(q_i, s_j)$ and similarly for s_{ij} and for d_{ij}.

6.1 SOME EXAMPLES OF TURING MACHINES

The remainder of this chapter shows some of the things Turing machines can do to the information placed on their tapes, and contrasts these processes with those obtainable from finite-state machines. (For the comparison, one may think of a finite-state machine as a specially restricted kind of Turing machine which can move in only one direction.)

6.1.1 A parity counter

We will set up a machine whose output is 1 or 0 depending on whether the number of 1's in a string of 1's and 0's is odd or even. The input string is represented on the Turing machine's tape in the form

where we have printed the sequence in question followed by a B. The machine starts (in state q_0) at the beginning of the sequence; the B is to tell the machine where the sequence ends. The machine needs two states, one for odd and one for even parity, and it changes state whenever it encounters a 1. The associated finite-state machine is represented by Table 6.1-1.

Table 6.1-1. QUINTUPLES FOR PARITY COUNTER

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
0	0	0	0	1
0	1	1	0	1
0	B	H	0	-

q_0

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
1	0	1	0	1
1	1	0	0	1
1	B	H	1	-

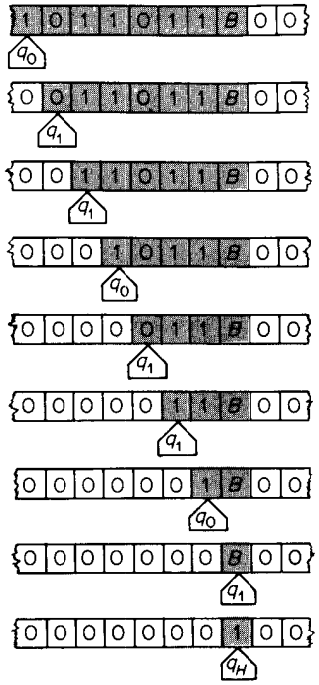
q_1

If we trace the operation of the machine we find that it goes through the configurations at the top of p. 121.

The machine ends up at the former site of the terminal B which it has replaced by the answer. The input sequence has been erased.

PROBLEM. Change the quintuples so that the sequence is not erased.

In this simple example the machine always moves to the right. In such a case there is no possibility of recording information on the tape and returning to it at a later time. Hence one could not expect it to do anything that could not also be done by an unaided finite-state machine (with sequential input) and we know already, from section 2.2, that this is true for this computation.



6.1.2 A parenthesis checker

We next give a simple example of a computation that *cannot* be performed by a finite-state machine with sequential input. The problem is to decide whether a sequence of left and right parentheses is *well-formed*, that is, whether they can be paired off from inside to outside so that each left parenthesis has a right-hand mate. Thus ((((()())())) is well-formed, while (()))) and)(and ((()))) are not. A good procedure for checking parentheses consists of searching to the right for a right parenthesis and then searching to the left for its mate and removing both. One keeps doing this until no more pairs are found. If any unmatched symbols remain, the expression is not well-formed, and conversely. Let us prepare the tape in the form

where the beginning and end of the expression are marked by A's, and use the machine of Table 6.1-2.

Table 6.1-2. QUINTUPLES FOR PARENTHESIS CHECKER

Q	S	Q'	S'	D
0	)	1	X	0
0	(	0	(	1
0	A	2	A	0
0	X	0	X	1
1	)	1	)	0
1	(	0	X	1
1	A	H	0	-
1	X	1	X	0
2	)	never occurs		
2	(	H	0	-
2	A	H	1	-
2	X	2	X	0

q_0 q_1 q_2

Tracing out the operation, we have

$A\,(\,()\,(\,(\,(\ldots$
[0]
$A\,(\,()\,(\,(\,(\ldots$
[0]
$A\,(\,()\,(\,(\,(\ldots$
[0]
$A\,(\,(X\,(\,(\,(\ldots$
[1]
$A\,(\,X\,X\,(\,(\,(\ldots$
[0]
$A\,(\,X\,X\,(\,(\,(\ldots$
[0]
$A\,(\,X\,X\,(\,(\,(\ldots$
[0]
$A\,(\,X\,X\,(\,(\,(\ldots$
[0]

and we see that one pair has been removed and the machine is searching for the next ')'. The state Q_0 is basically a right-moving state which searches (without changing symbols) until it encounters a ')'. It removes this (by X-ing it) and goes to state Q_1. State Q_1 is basically a left-moving state which searches for a matching '('. If it finds one, it X-s it and returns to state Q_0. This pairing-off continues until one of two events occurs. (1) State Q_1 may find no '(' mate. If it thus reaches an A (which would be the one to the left) the machine prints a 0 (meaning *not well-formed*) and halts. Or (2) state Q_0 may find no more ')'-s. It knows this by encountering an A (which must be the one at the right). The machine then enters state Q_2 which checks to see if any '('-s remain. If there is one, the machine again prints 0 and halts. If no left parentheses remain (which the machine finds out by reaching the left-hand A), the machine prints 1 (well-formed!) and halts.

This is a computation which cannot be done by a finite-state machine, as noted in 4.2.2. It is possible here because our Turing machine can go back over arbitrarily long intervals† to find '('-s that it had passed over earlier.

Let us note in passing an interesting property of this machine. It does not make very much use of any particular tape square and, indeed, changes its contents no more than once. We will show later (in section 14.5) that any machine is equivalent in a sense to one with this strong restriction.

DIAGRAM CONVENTIONS

In most of our machines each state will have the character of an unidirectional search: each state is usually associated with moves in a single direction. The diagrams can be made simpler and more transparent by recognizing this fact. Thus we can represent the machines of 6.1.1 and 6.1.2 by the diagrams in Fig. 6.1-1. Each arrow in the diagram represents

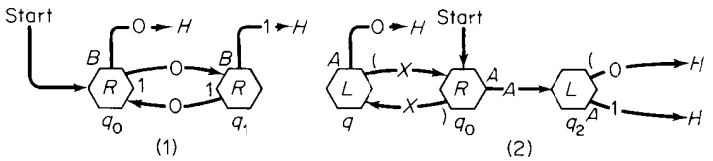


Fig. 6.1-1

some quintuple $(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$. Then q_i is the state at the tail of the arrow, s_j is the symbol at its tail, s_{ij} is written in the middle of the arrow and omitted if the same as s_j, q_{ij} is the state at the head of the arrow, and d_{ij} is the symbol written inside the hexagon for q_{ij}. If two q_{ij}'s name the same state but their d_{ij}'s are different, we cannot use this kind of diagram.‡ The most common quintuples, of the form $(q_i, s_i, q_i, s_j, d_{ij})$ are simply omitted.

PROBLEM. The reader should reconstruct Table 6.1-2 from Fig. 6.1-1(b) to be sure he understands these conventions.

6.1.3 A unary-to-binary converter

In sections 3.2.4 and 3.2.5 we described a network which converts a sequence of n pulses into a parallel pattern corresponding to the binary

†That is not the whole story, however. A remarkable theorem of Rabin and Scott [1959] shows that machines which can move both ways, but can *not* change tape symbols, are no more (or less) powerful than machines which can move only in one direction.

‡This notation is based on a suggestion of N. Rochester. It is important to recognize that it can be used only in the peculiar circumstance that *entering* each state is associated with a fixed direction. It just happens that most of our examples satisfy this condition. Problem: Show that any machine is equivalent to such a "directed-state machine."

number representation of n. Of course such a device can handle numbers only up to a certain size (depending on the number of binary scaler units it contains). We can make a Turing machine which will do this for all sequences, however large. Let the sequence be represented as a sequence of 1's on an infinite, otherwise blank tape:

The machine starts at the leftmost 1 and has the diagram shown in Fig. 6.1-2. The paired right-moving states act like a parity counter (see section 6.1.1); they also remove alternate 1's (by X-ing them). When the parity counter has passed through the data string, it enters one of the left-moving states, which enters an A or a B in the first available space to the left. When no 1's are left in the data string, the machine halts. It turns out that the A's and B's so written are the binary digits of the original number of 1's. To see this, one can verify that at the moments of entering the right-hand R state the tape has the forms shown below. (The reader should trace the machine through all the steps.)

0	1	1	1	1	1	1	1	1	1	1	0
A	*X*	1	*X*	1	*X*	1	*X*	1	*X*	1	0
B	*A*	*X*	*X*	*X*	1	*X*	*X*	*X*	1	*X*	*X*	0
A	*B*	*A*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	1	*X*	*X*	0
B	*A*	*B*	*A*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0

So the answer is $BABA \equiv 1010 = 10$ (base 2). The computation repeatedly divides by 2, writes down the remainder (that is, A or B), and repeats the process on the quotient until the quotient becomes zero.

The structure of this process is rather more complex than in the previous examples. The two parity-counter states may be thought of as a *subprocess* which is repeated over and over until some condition, detected by a "supervisory" process, is attained. We have a "loop within a loop" and can discern a rudimentary hierarchy of control.

This computation cannot be done by a finite-state machine.

PROBLEM 6.1-1. It is interesting that exactly the same result can be achieved by a binary counter scheme which requires only two states and the same alphabet! Can you find such a two-state Turing machine?

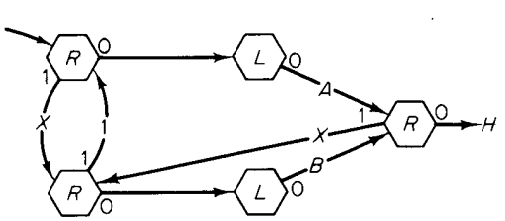


Fig. 6.1-2

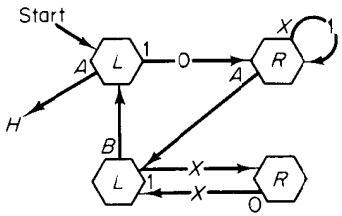
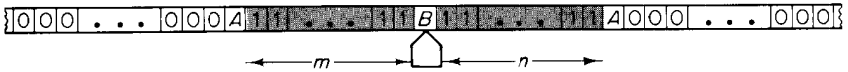


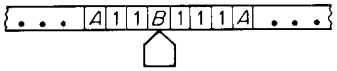
Fig. 6.1-3

6.1.4 A unary multiplier

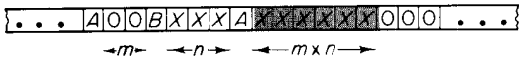
The machine in Fig. 6.1-3 will *multiply* two unary numbers m and n represented as blocks of 1's on a tape of the form



The machine starts at the separating B. If, for example, the initial tape is



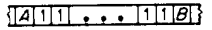
the end result will be



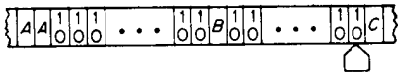
or $2 \times 3 = 6$, where the answer (in unary form) will be found as a block of X's to the right of the terminal A. Again we have a computation that cannot be done in any finite-state machine for arbitrary values of m and n.

PROBLEM 6.1-2. Construct Turing machines which:

(1) compute the square of n, where n is represented as

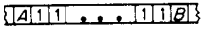


(2) compute the sum of two binary numbers represented in a form†



Assume that they have the same numbers of digits.

(3) decide whether or not the number is a prime number (very complicated).



†Where $\begin{matrix}1\\0\end{matrix}$ means the square contains 1 or 0.

PROBLEM 6.1-3. Describe the behavior of the machine in Fig. 6.1-4, when started with a blank tape. This is just an exercise in "keeping track."

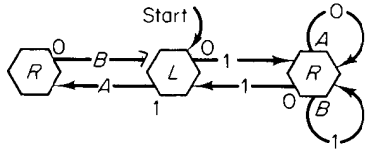


Fig. 6.1-4

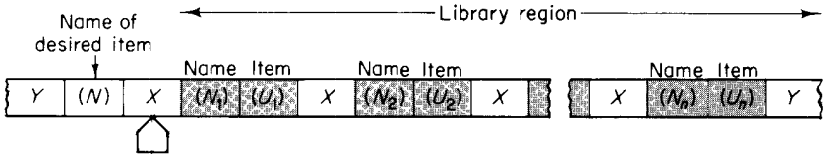
PROBLEM 6.1-4. Design Turing machines to compute (1) the product of two binary numbers m and n; (2) the exponent function m^n.

6.1.5 An addressed memory

It is not particularly apparent, from the previous examples, how a Turing machine can be constructed to use its tape as a general-purpose file for storing and retrieving information. The next example shows how one might set up an orderly file in which items could be located by name.

LOCATION AND FILE.

Suppose that one has a number of items U_i, each of which is associated with a name N_i. Let these be encoded in a binary alphabet and arranged along a tape in the form of pairs (name, item) separated by markers. (We will use X's.) Suppose that we want to locate the item whose name is N. We assume, for simplicity, that all of the names N_i have the same number of digits, and that the tape has the form:



where the parentheses represent strings of binary digits and the Y's are used as additional punctuation. The machine (Fig. 6.1-5) uses two additional symbols, A and B, for keeping track of position. It will compare

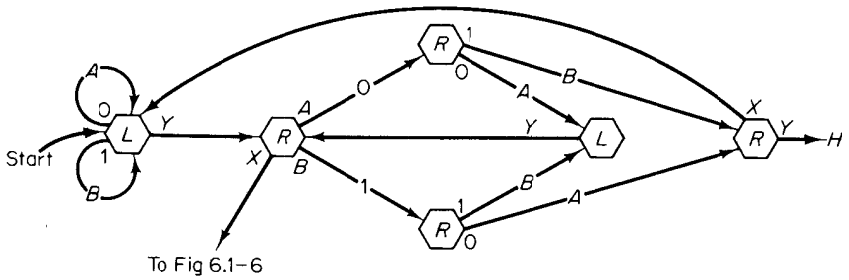
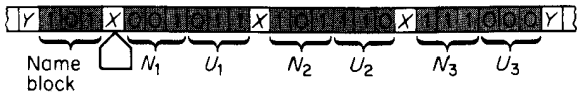
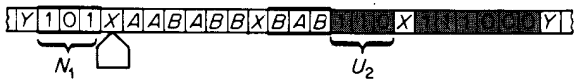


Fig. 6.1-5. Locate name.

the left-hand name with each of the names in the library, and when it finds a perfect match, it will stop at the starting position, having changed all 0's and 1's between there and the desired record $U(N)$ to A's and B's. For example, this machine would convert the tape



into



Note that all the information in the original file can be recovered by replacing the A's and B's by the corresponding 0's and 1's.†

COPYING

Having located a file, it may be desirable to move the information to another place. If we add the machine in Fig. 6.1-6 to the above, it will take the item U_i just located and *copy it into the block in which its name*

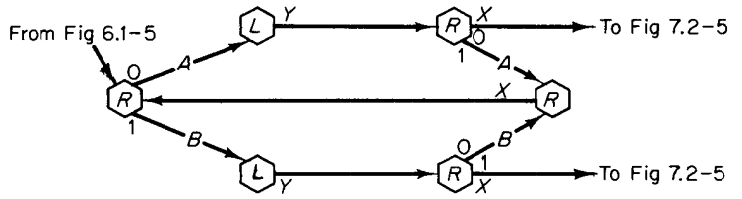
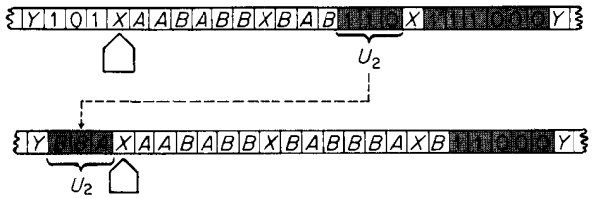


Fig. 6.1-6. Copy item.

was originally located—that is after the initial Y. Then it will stop at the starting position. We assume that the items U_i have the same length as the names.

The result is to copy the information U_i into the "address" block after the initial Y:



†The reader should trace the operation to see how the machine works. If it seems wrong, try again, being twice as careful, etc.

Again, it would be a simple matter to add states which would restore the tape to its initial 0, 1 format.

The memory file we have constructed is a variety known as an *associative memory*. Items are recovered directly by matching with a name which is paired with the item. In a conventional serial computer one retrieves items from a memory by a more indirect process, namely, through reference to their location in a fixed array of storage devices.† There are certain advantages to having the option of associative memory in a computer, but we cannot stop to discuss this here.

6.2 DISCUSSION OF TURING MACHINE EFFICIENCY

The reader who has traced out the above machines on actual tapes will have observed two complementary features of the situation. On the one hand, he is becoming convinced that one can indeed do a great many things with Turing machines that cannot be done with finite-state machines—and also that one can do surprisingly complicated things with rather simple machine structures. On the other hand, he will have observed the staggering inefficiency of Turing machines inherent in their repeated passage over long blocks of tape to perform the most elementary operation of moving a temporary marker. No one has seriously proposed the use of a Turing machine structure for any practical computation. Indeed, I do not even know of one built for demonstration purposes.

Nevertheless, despite the slowness of the machines, they do *not* necessarily make inefficient use of the tape memory capacity.

> For example, we can make a machine to decide if an n-digit binary number is prime, using less than n additional tape squares. Perhaps it should also be noted that the high speed of ordinary computers is based on their use of "random-access" memories which only appear to escape the problem of serial search along a tape. If one thinks of such machines as existing in a sort of three-dimensional tape, one can construct arguments to show that Turing machines of appropriate kinds are not fundamentally slower than other kinds of finite-state-plus-memory machines.

As we will see, it is possible to execute the most elaborate possible computation procedures with Turing machines whose fixed structures contain only dozens of parts. One can imagine an interstellar robot, for whom reliability is the prime consideration, performing its computations in such a leisurely manner, over eons of spare time.

There are modifications of the Turing machine framework, e.g., using several tapes, which bring it closer in efficiency to the conventional computing machine, but we will not look into this. Our real interest in Turing machines is in their use as a model for defining effective computability.

†The point is that, with the associative memory, one does not have to know *where* the desired item is located.

6.3 SOME RELATIONS BETWEEN DIFFERENT KINDS OF TURING MACHINES

6.3.1 Two-Symbol Turing machines

We have placed no particular restriction on the variety of symbols that may occur on the tapes of our machines, save that each machine can deal with just some finite set of symbols. It is interesting that *we can restrict our machines to the use of two symbols*, without loss of generality. (In finite-automata theory this is reminiscent of the equivalence of some McCulloch-Pitts networks, which have binary signals, to all less-restricted finite-state machines.)

To show the equivalence, one has to do little more than replace the symbols by binary numbers. Suppose that a Turing machine T uses k different symbols. Suppose also that the number k has n binary digits. Then we can assign to each symbol of T a distinctive n-binary-digit number. Now we can replace the machine T by a new machine T^* which will treat its tape, in effect, *as though its squares were grouped in blocks of length n.* On each of these blocks is written an n-digit representation of some symbol of the old machine T. For each state of the old machine T we will give T^* a collection of states, and these will be arranged somewhat like the units of the sequential binary decoder 3.3.1. In fact we construct a binary tree, n layers deep, of right-moving states. If the machine starts at the left of a symbol block, then, when it has reached the right end of that block, the position of its state in that binary tree will tell precisely what symbol (from T's old alphabet) was represented there. It is then a simple matter to attach, to each terminal state of this tree, a chain of n left-going states which write (in reverse order) the binary digits of the representation of that symbol which T would then have written there. Finally, if T were to move right, we have to adjoin n right-moving states, and likewise for motion to the left. A detailed construction is given in Shannon [1956]. See also section 14.5 for a similar construction.

6.3.2 Single and double infinite tapes

It can also be shown that Turing machines which have tapes infinite in both directions have no advantage or disadvantage over machines with singly infinite tapes. It is not worth showing this here in detail. Suffice it

to note that one can show the equivalence by *folding* the doubly infinite tapes, and then mapping them into the single tapes by using alternate squares for each half. Turing [1936] uses this method. See the solution to problem 7.4-1.

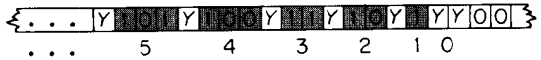
6.3.3 Multiple-tape machines

By extending the idea of using alternate squares, and adjoining enough markers and auxiliary states, one can achieve the effects of having several independent tapes (controlled by n-tuples in the finite-state machine outputs) in a single tape. One can even simulate the effects of an n-dimensional memory array. Again, we will not go into detail.

6.3.4 Two-state Turing machines

A really startling reduction was demonstrated by Shannon [1956]; any Turing machine is equivalent to a Turing machine with only *two* internal states! The equivalence is achieved by greatly enlarging the alphabet of the multistate machine. We cannot give details of this remarkable demonstration here, and the interested reader should certainly refer to the original paper. The heart of the argument is contained in the diagram of Shannon's Table I; this shows how the state information of the original machine can be preserved, as the machine moves along its tape, by a two-square shuttling back and forth, running through different alphabets.

PROBLEM 6.3-1. Design a Turing machine to enumerate the binary numbers, i.e., to produce an infinite tape; start with a blank tape and generate



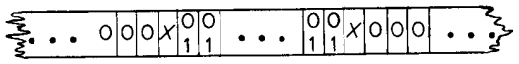
The ground rule is that once a Y is printed the machine must never again change it or move to the right of it.

PROBLEM 6.3-2. Design a Turing machine to generate all well-formed parenthesis sequences, i.e., to produce an infinite tape like

The order does not matter, but no sequence may appear more than once. Follow the same ground rule as in first problem.

Hint: Enumerate the binary numbers as in problem 6.3-1. As they are generated, interpret them as parenthesis sequences and check them for well-formedness (using 6.1.2). If details become too nasty, just sketch out the problems and how to deal with them.

PROBLEM 6.3-3. We will say that a Turing machine "accepts a finite sequence S of 0's and 1's" if, starting with



the machine eventually halts with a blank tape. Show how to make Turing machines to accept exactly the sequences of the following classes.

(1) $10, 1100, 111000, \ldots, 1^n0^n, \ldots$.

(2) The set of sequences accepted by any finite-state machine.

(3) Problem (2), but using only right-moving states in the Turing machine.

(4) Prove that a right-moving Turing machine can be replaced by a finite-state machine.

(5) Can you do problem (1) with a right-moving machine? Prove it.

(6) Can you think of a set of finite sequences that is not the set accepted by some Turing machine?

7 UNIVERSAL TURING MACHINES

7.0 USING TURING MACHINES TO COMPUTE THE VALUES OF FUNCTIONS

In chapter 6 we saw a number of ways to make Turing machines manipulate the information on their tapes. In each case we started with some information on the tape (usually a binary or unary number, but occasionally a string of symbols). The machine was started in some standard state and tape position and allowed to run until it halted. When (and if) the machine stopped, we looked at the information now on its tape and regarded this as the result of a computation.

Now in general, what is on the tape when the machine stops will depend in some complicated way on what was on the tape at the start. So we can say that the tape result of the computation is a *function* of the input. Exactly what function it is depends, of course, on what Turing machine was used. Hence we can think of a Turing machine as *defining*, or *computing*, or even as *being a function*.

What is a function? Mathematicians have several more or less equivalent ways of defining this. Perhaps the most usual definition is something like this:

> A *function* is a rule whereby, given a number (called the *argument*), one is told how to compute another number (called the *value* of the function for that argument).

For example, suppose the rule that defines a function F is "the remainder when the argument is divided by three." Then (if we consider only non-negative integers for arguments) we find that

$$F(0) = 0, \quad F(1) = 1, \quad F(2) = 2, \quad F(3) = 0, \quad F(4) = 1, \quad \text{etc.}$$

Another way mathematicians may define a function is:

> A function is a set of ordered pairs $\langle x, y \rangle$ such that there are no two pairs with the same first number, but for each x, there is always one pair with that x as its first number.

If we think of a function in this way, then the function F above *is* the set of pairs

$$\langle 0, 0 \rangle \quad \langle 1, 1 \rangle \quad \langle 2, 2 \rangle \quad \langle 3, 0 \rangle \quad \langle 4, 1 \rangle \quad \langle 5, 2 \rangle \quad \langle 6, 0 \rangle \quad \ldots$$

Is there any difference between these definitions? Not really, but there are several fine points. The second definition is terribly neat; it avoids many tricky logical points—to compute the value of the function for any argument x, one just finds the pair that starts with x and the value is the second half of the pair. No mention is made of what the *rule* really is. (There might not even be one, though that leaves the uncomfortable question of what one could use the function for, or in what sense it really exists.) The first definition ties the function down to some stated rule for computing its values, but that leaves us the question of what to do if we can think of different rules for computing the same thing! For example,

> (1) divide the number by three and the value is the remainder
> (2) add up the number's digits (base 10, of course) and divide *that* by three and take the remainder

are two rules that, as many children know, give the same values (prove it!). It would be a nuisance to think of them as two different functions, but they *are* two different rules; so we have a problem with our first definition.

In ordinary life, or even in ordinary mathematics, worrying about such a hair-splitting matter would be silly. One just says, informally, "Well, if two rules do in fact compute the same values, we will think of them as defining the same function. The different definitions, while not identical, are *equivalent*." Now we will in fact take this common-sense approach, but we have to be careful to be clear about it. This is not because we are trying just to be terribly careful and logical and so forth, but rather because we are studying the theory of rules and definitions and the like as our primary subject matter. We have to make a clear distinction between what a rule does and how it is defined because that is exactly what we are studying—not because of an obsessive, or evangelistic, or therapeutic desire to promote more clear and logical thinking generally. I say this because many of the newer mathematics texts belabor the distinctions between functions and rules and ordered pairs to the point that the student is more confused than he was before, precisely because in

the subject matter he is studying the distinctions are so unimportant that there is little or no danger of confusion to begin with.

So we will think of a function as an association between arguments and values—as a set of ordered pairs $\langle x, y \rangle$ such that there is just one pair for each x. For each function there may be many *definitions* or *rules* that tell how to find the value y, given the argument x. Two definitions or rules are *equivalent* if they define the same function. It may be very difficult, given two different rules, to tell if they are in fact equivalent. In fact it may, in a sense, be impossible, and this will form the central problem discussed in chapter 8!

7.0.1 Functions of non-negative integers

Mathematicians use the notion of function very broadly in that they allow almost any kind of thing to be an argument or value. For instance, we can define the function whose arguments are sentences in English and whose value, for an argument x, is the *subject* of x.

$$F(\text{Mary had a little lamb.}) = \text{Mary}$$

We will not stop to discuss whether this definition could ever be made really precise but will just observe that here we are thinking of a function that has a sentence, rather than a number, for an argument. Another, more common kind of function is the function whose argument is a *set* and whose value, for an argument x, is *the set of subsets of* x:

$$F([a, b, c]) = [[\,], [a], [b], [c], [a, b], [a, c], [b, c], [a, b, c]]$$

where—as most children today know—[ ] is the empty set and, including that, there are eight possible subsets of three objects. This kind of function is called a *set function* because its arguments and values are sets instead of numbers.

In most of what follows, we will deal with a very simple kind of function—namely, functions whose arguments and values are non-negative integers, i.e., are in

$$0, 1, 2, 3, 4, \ldots$$

There is nothing really special about this; the theory would be the same in intellectual content, only just more complicated, if we used negative numbers or rational numbers as well. (It would be different if we allowed *real* numbers; that is what the [optional] chapter 9 is about.) In some of our theory, particularly in the present chapter, we encounter some functions whose arguments are a little more like sentences in English. But unless otherwise noted, we will talk only about functions that concern the non-negative integers.

We are also interested in the rules for defining functions, and for the next few chapters we will consider mainly a very special kind of rule—namely, *defining functions in terms of the behavior of Turing machines.*

7.0.2 Functions defined by Turing machines

We have already noted, at the beginning of this chapter, that we can define a function in terms of the behavior of a Turing machine; the argument is written on the tape at the start, and the value is found on the tape when the machine stops. In chapter 6 we discussed how to do this for a number of arithmetic functions including

$$n^2, \quad \text{parity}(n), \quad x + y, \quad xy, \quad \textit{etc.}^\dagger$$

From now on, we will fix our attention almost entirely on functions that can be defined in this way by Turing machines. In order to be able to talk precisely about such matters, we will have to agree on some rather strict definitions. I want to reassure the reader about these in advance. The definitions below may seem very narrow and restrictive. The reader will be able to think of many other ways he could use a Turing machine to define a function, and he may feel that in the restriction to doing it our way a great deal is being lost. He should keep these objections in reserve until he has completed this chapter. Only then can one see clearly the incredible range over which strikingly different kinds of definitions, in this general area, come out finally to have exactly the same character.

7.0.3 Turing-computable functions

A function $f(x)$ will be said to be Turing-computable *if its values can be computed by some Turing machine T_f whose tape is initially blank except for some standard representation of the argument x. The value of $f(x)$ is what remains on the tape when the machine stops.*

Now there is a real problem in the above definition unless we agree on a "standard representation" for numbers. For what does $f(111)$ mean? If the numbers are unary it means $f(3)$; if binary, $f(7)$; if decimal, $f(111)$. Similarly, we have to choose a standard convention for interpreting the value result. It turns out, it doesn't much matter what we do, so long as we are consistent throughout our theory. So we will assume, unless otherwise noted, that numbers will be represented in unary notation. They will also sometimes begin and end with other special symbols (for punctuation,

†Our discussion about ordered pairs, etc., doesn't tell us what to do about functions of several arguments, e.g., $f(x, y) = x + y$. Clearly this needs something more—say, ordered triples. This slightly complicated matter is discussed in section 10.2.

so that the Turing machine can, for example, tell where a number ends and where blank tape begins).

We have just said that if the representation of numbers remains consistent throughout our theory, it doesn't much matter what it is. On the other hand, if we do alter the representation of numbers *within* the theory, we obviously *do* change the function that a Turing machine can be considered to compute. Indeed, for a given Turing machine, there might be a great many different functions associated with different ways of representing the arguments on the initial tape. In this chapter we show Turing's great discovery: that there is a Turing machine so sensitive in this respect that, by properly adjusting the input representation, we can cause it to compute *any* Turing-computable function—that is any function that can be computed by any other Turing machine whatever!

To summarize his assertion briefly, let us agree that f is a Turing-computable function, and T is a Turing machine that computes it if

> when the number x is written in unary notation on a blank tape, and T is started in state q_0 on the rightmost 1 of x, then, when the machine stops, the number $f(x)$ will appear in unary notation on the tape.

Then Turing shows how to construct a single, fixed machine U with the property that *for each and every Turing machine T, there is a string of symbols* d_T such that

> if the number x is written in unary notation on a blank tape, followed by the string d_T, and U is started in q_0 on the leftmost symbol of d_T, then when the machine stops the number $f(x)$ will appear on the tape,

where $f(x)$ is the number that would have been computed if the machine T had been started with only x on its tape.

This is what we are to prove in this chapter. Before doing so, let us see what it says. It says that no matter how complex the structure of a Turing machine T, its behavior is within the reach of the fixed machine U. It does not matter that T may be very much larger than U in its number of states, or that T may use more and different symbols than U. All U needs is to be provided with a certain string of symbols d_T, and it can compute the same function that T does. What is d_T? Obviously it is some way in which the machine T is described to U.

The existence of the machine U is a chief support of the thesis that Turing's notion of computability is an acceptable technical counterpart of our intuitive notion of effective computability. For on the one hand it is certainly hard to imagine an objection that Turing machines are too powerful. All they do is read, write, and move. On the other hand, no one has ever been able to think of anything a universal Turing machine cannot do that one could reasonably ask of a well-defined instruction-obeying process. As we will see later, this idea has stood the test of time. All other formulations that seemed otherwise satisfactory have been shown equivalent to the notion of Turing computability.

The universal machine quickly leads to some striking theorems bearing on what appears to be *the ultimate futility of attempting to obtain effective criteria for effectiveness itself*—that is, to recognize which descriptions of processes do in fact describe effective processes. We shall discuss some of these results in the next chapter.

Fortunately, the demonstration that there exist universal machines, and even their construction, is quite straightforward, and much of the work has already been done in chapter 6. The following sections complete the construction.

7.1 THE UNIVERSAL MACHINE AS AN INTERPRETIVE COMPUTER

The manner of operation of a universal Turing machine is familiar to many computer programmers in the form of "interpretive" programming systems. The idea is this. Let $f(x)$ be Turing-computable; then, by definition, there is some Turing machine T which can compute its values. That is, for each value of x, represented on the tape of T as a certain string s_x of symbols, T will eventually halt with an appropriate string $s_{f(x)}$ remaining on its tape. It is not important just what form is chosen for s_x, provided that we agree throughout the discussion to keep it fixed.

Now if the reader were given a description of T and also of s_x, he could trace out the behavior of T with input s_x and find for himself what will be the corresponding value of $f(x)$. This is what the reader has done, presumably, for some of the machines described in the previous chapter. He required, presumably, an external storage medium—paper and pencil—some time, and a precise understanding of how to interpret the description of each machine.

The universal machine will be given just the necessary materials: a description, on its tape, of T and of s_x; some working space; and the built-in capacity to interpret correctly the rules of operation as given in the description of T. Its behavior will be very simple. U will simulate the behavior of T one step at a time. It will be told by a marker M at what point on its tape T begins, and then it will keep a complete account of what T's tape looks like at each moment. It will remember what state T is supposed to be in, and it can see what T would read on the "simulated" tape. Then U will simply look at the description of T to see what T is next

supposed to do, and do it! This really involves no more than looking up, in a table of quintuples, to find out what symbol to write, which way to move, and what new state to go into. We will assume that T has a tape which is infinite only to the left, and that it is a binary (2-symbol) machine. These restrictions are inessential, but make matters much simpler.

To make the situation more concrete, let us assume that the universal machine U has its tape divided into regions as indicated in Fig. 7.1-1.

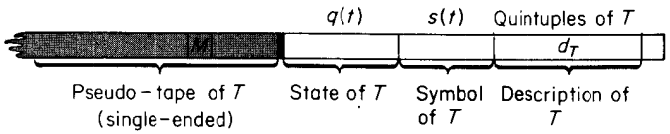


Fig. 7.1-1.

The infinite region to the left will be the current version of the tape of machine T. Somewhere in this semi-infinite region is a marker M to show where the reading head of T is currently located.

The second region contains the name of the current state of T. Next there is a space in which we record the symbol that T has just read or is just about to write, depending on the phase of the process. Finally there is a region, which will be described in section 7.2, containing the description of T.

7.2 THE MACHINE DESCRIPTIONS

The structure of a Turing machine is determined entirely by its state diagram, i.e., by the set of quintuples which describe the relation between its states, inputs, and outputs. Therefore a complete description of T is realized simply by setting down a list of its quintuples. We will set this list down on U's tape in a very straightforward form. (See Fig. 7.2-1.) *The quintuples are represented in binary notation*, separated by X's, which are used for punctuation.

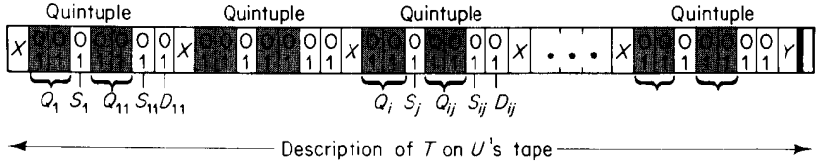


Fig. 7.2-1.

We have to use some representation like binary numbering to designate the states of the machine T. The reason is that the universal machine U which we are constructing is just one fixed machine, hence it is limited to some fixed set of tape symbols. On the contrary, the machines to be simulated, while each one of them is finite, may require arbitrarily large numbers of states; and so U cannot afford a different symbol for each T state. (We have assumed that all the T machines will be 2-symbol machines; so there is no problem about that.) Our problem is solved as follows: If T has n states, and n is a k-digit binary number, we will use blocks of k binary digits to represent the states. In Fig. 7.2-1, k was chosen to be 2. In that same figure, we need only single binary digits for each of the two-valued quantities S_j, S_{ij}, and D_{ij}.

To complete the description of T on U's tape we need to represent T's tape, its location, and its current state-symbol pair. For this we will use the form shown in Fig. 7.2-2. We assume that the region of the tape called "machine condition" contains

(1) The initial state Q_0 of T
(2) The symbol first scanned by T

(M has usurped the position of the symbol (2) on T's imitated tape.) Also we assume that U's reading head is placed over the leftmost X as shown.

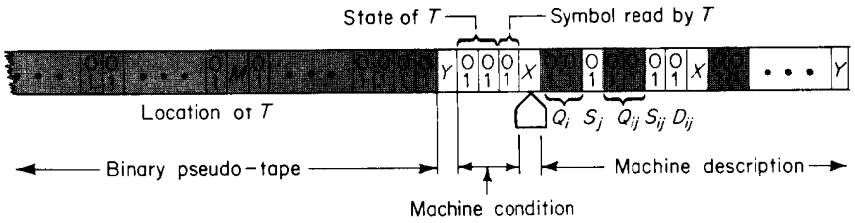


Fig. 7.2-2.

Now U's operation proceeds in a cycle with four parts:

(1) To start, the "locate" machine of Fig. 6.1-5 is put into operation. It first searches to the right to find the first state-symbol pair that matches the one represented in the "machine condition" area. On the path to the desired state-symbol pair, it changes all 0's and 1's to A's and B's. After finding the desired pair, and changing it also to A's and B's, it runs back to the leftmost X. At the end of this process, the tape looks like Fig. 7.2-3.

(2) Next the "copy" machine of Fig. 6.1-6 is put into operation. It starts on the leftmost X, and from there shifts to the right until it has gone past the last A or B and sees for the first time some 0's and 1's. These 0's

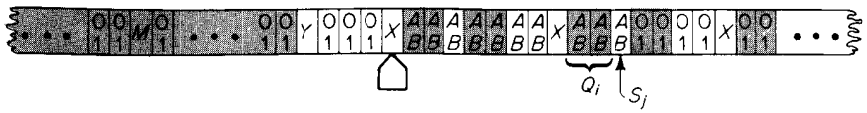


Fig. 7.2-3.

and 1's describe the new state Q_{ij} of T, the new symbol S_{ij} which must eventually be printed in location M, and the direction of motion D_{ij}. Next, the machine copies the 0's and 1's respresenting Q_{ij} and S_{ij} into the machine-condition region; it does not print D_{ij} but just remembers it. Now the tape looks like Fig. 7.2-4.

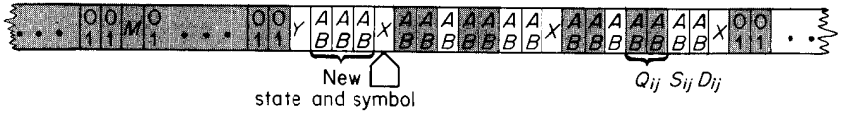


Fig. 7.2-4.

(3) In its third phase, the machine shifts to the left until it reaches M. There it erases M and prints (temporarily) in its place the direction D_{ij} (= A or B) which it has until now remembered. This information was retained in the choice between the two exit arrows of the "copy" part of Fig. 6.1-6. Then the machine shifts to the right and changes all A's and B's on the tape to 0's and 1's, except for the A or B in M's old location that now represents D_{ij}. Finally, it moves to the immediate left of the leftmost X. It erases the S_{ij} which is there, remembers it, and prints 'S' in its place. (S is a special letter used only for this marking job.) This is all done by the machinery of Fig. 7.2-5. At this point the tape looks like Fig. 7.2-6.

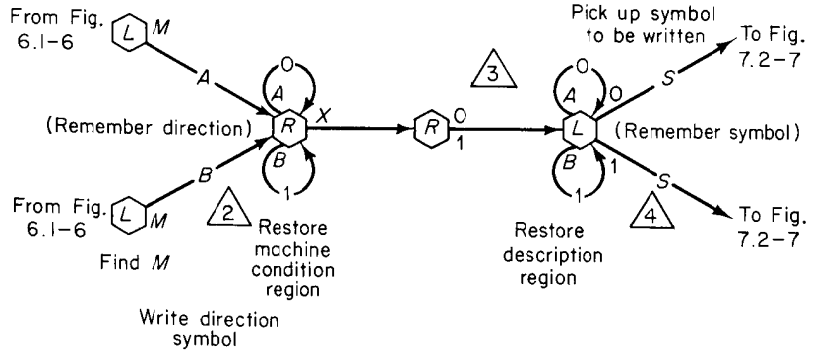


Fig. 7.2-5.

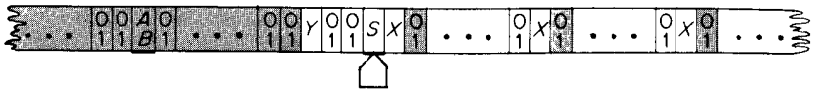


Fig. 7.2-6.

(4) Now in its final phase, the work of the machine T is about to be completed. The machine shifts to the left until it encounters for the first time an A or B. This A or B represents the direction D_{ij} that T should next move. The machine prints S_{ij} = 0 or 1 in place of the A or B, and shifts one square left or right according to whether D_{ij} was A or B. Then it reads this new tape square, remembers whether it is 0 or 1, and prints an M in its place. Finally, the machine shifts to the right until it reaches the S. It replaces that S by the remembered symbol, using A for 0 and

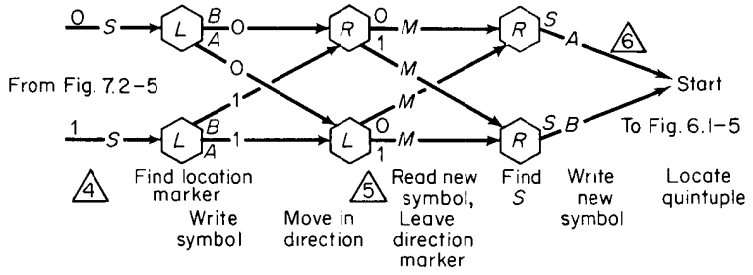


Fig. 7.2-7. Perform pseudo-tape operations.

B for 1. All this is done by the machinery of Fig. 7.2-7. At this point, the tape looks like Fig. 7.2-8. Now the machine starts all over again.

What happened as a result of this cycle of operations? The net effect of what the machine did was to start with a particular state-symbol pair Q_i, S_i, search for the corresponding quintuple [Q_i, S_i, Q_{ij}, S_{ij}, D_{ij}] and carry out the instructions dictated by it. Then it printed the new state-symbol pair in place of the old one. Thus the machine carried out the instructions of exactly one quintuple, and is prepared to start again on the next step of T's computation.

This completes the state diagram and coding instructions for our universal Turing machine, and thus completes the demonstration that

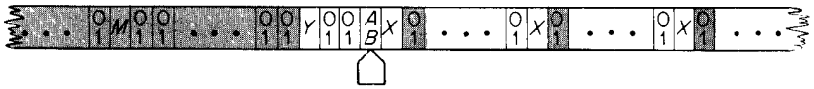
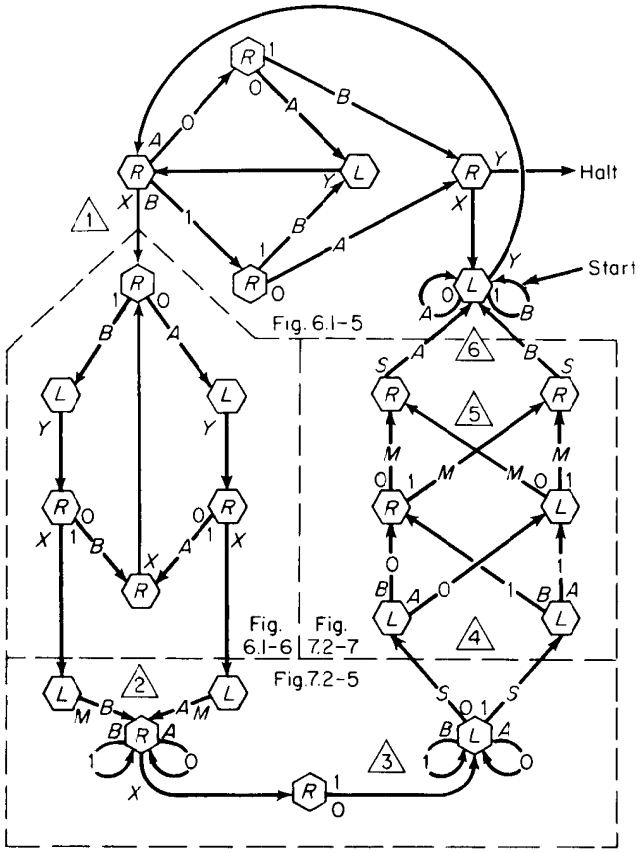


Fig. 7.2-8



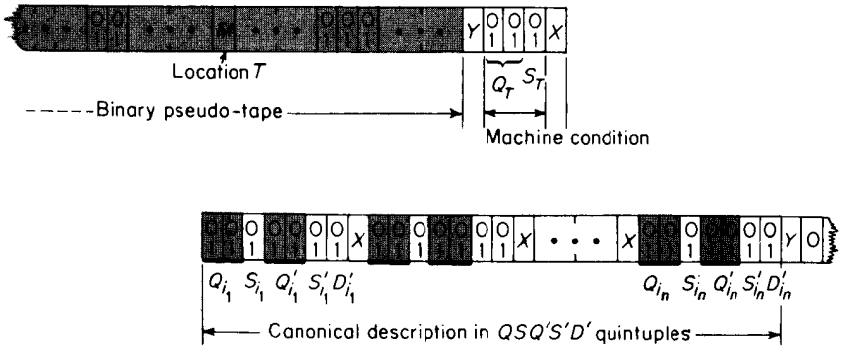


Fig. 7.2-9. The universal machine.

such a machine can be constructed! The complete diagram is shown in Fig. 7.2-9. To review its operation, in section 7.3 we will trace it through an operation cycle of a particular simulated machine, presenting the tape condition at important phases in the process.

7.3 AN EXAMPLE

We will use, as a simple but non-trivial example, a 2-state, 2-symbol machine T which happens to be a sort of binary counter. The diagram of the machine is given in Fig. 7.3-1, and its quintuples are

$Q_0S_0\ Q_0S_0D_1$	00 001
$Q_0S_1\ Q_1S_1D_0$	01 110
$Q_1S_0\ Q_0S_1D_1$	10 011
$Q_1S_1\ Q_1S_0D_0$	11 100

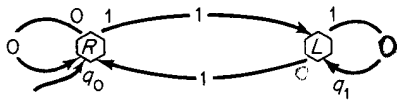
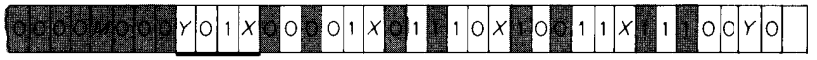


Fig. 7.3-1

When placed on a tape over a 1, this machine proceeds to "count"—that is, to write the binary numbers 1, 10, 11, 100, 101, 110, 111, 1000, etc., one over the next, and never stops. We are not interested in T itself but only in following U's simulation of it. We set up the full initial tape to describe T to U:



where the '$Y01X$' initial setting means that the machine being simulated starts at the square marked with 'M' in state Q_0, and sees there the sym-

Table 7.3-1

U	T
0 0 *M* 1 0 0 0 *Y* 1 *A* *X* etc.	0 0 0_1 1 0 0 0
0 0 1 *M* 0 0 0 *Y* 0 *B*	0 0 [1] 1_0 0 0 0
0 0 *M* 1 1 *B*	0 0 1_1 1 0 0 0
0 *M* 0 1 1 *A*	0 0_1 0 1 0 0 0
0 1 *M* 1 0 *A*	0 1 0_0 1 0 0 0
0 1 0 *M* 0 *B*	0 [1 0] 1 0 0 0
0 1 *M* 1 1 *A*	0 1 0_1 1 0 0 0
0 1 1 *M* 0 *B*	0 [1 1] 1 0 0 0
0 1 *M* 1 1 *B*	0 1 1_1 1 0 0 0
0 *M* 0 1 1 *B*	0 1_1 0 1 0 0 0
M 0 0 1 1 *A*	0_1 0 0 1 0 0 0

bol 1 = S_1. The reader will note that we have marked certain points in the diagram (Fig. 7.2-9) of U with the symbols ① ② ③ etc. Below are the tape configurations that occur at these points of the first cycle.

start	.. 0 0 0 M 0 0 0 Y 0 1 $\underleftarrow{X}$ 0 0 0 0 1 X 0 1 1 1 0 X 1 0 0 1 1 X 1 1 1 0 0 Y
①	.. 0 0 0 M 0 0 0 Y 0 1 $\underrightarrow{X}$ A A A A B X A B 1 1 0 X 1 0 0 1 1 X 1 1 1 0 0 Y
②	.. 0 0 0 $\underrightarrow{A}$ 0 0 0 Y B B X A A A A B X A B B B A X 1 0 0 1 1 X 1 1 1 0 0 Y
③	.. 0 0 0 A 0 0 0 Y 1 1 X A 1 A A B X A B B B A X $\underleftarrow{1}$ 0 0 1 1 X 1 1 1 0 0 Y
④	.. 0 0 0 A 0 0 0 Y 1 $\underleftarrow{S}$ X 0 0 0 0 1 X 0 1 1 1 0 X 1 0 0 1 1 X 1 1 1 0 0 Y
⑤	.. 0 0 $\underrightarrow{M}$ 1 0 0 0 Y 1 A X 0 0 0 0 1 X 0 1 1 1 0 X 1 0 0 1 1 X 1 1 1 0 0 Y
⑥	.. 0 0 M 1 0 0 0 Y 1 $\underleftarrow{A}$ X 0 0 0 0 1 X 0 1 1 1 0 X 1 0 0 1 1 X 1 1 1 0 0 Y

Each time U reaches stage ⑥, its machine tape has the configuration shown to the left in Table 7.3-1. To the right is what T's tape looks like at the same stage. The subscript on T's tape indicates the last symbol scanned, and is 1 if the scanning state was Q_1, and 0 if the scanning state was Q_0.

The terminal, right-hand Y marks the end of the description region. If the location phase does *not* yield any quintuple beginning with Q_iS_j, the process will encounter this Y and halt. A standard way to make Turing machines stop is to assign no quintuple to some Q_iS_j, and U recognizes this convention by halting on this Y.

PROBLEM 7.3-1. Why ought we provide a 0 (or a 1) after this Y?

7.4 REMARKS

This construction of a universal Turing machine is reasonably straightforward in the way that different parts of the process—finding the quintuple, changing state, reading, printing, and moving—are separated in parts of the state diagram. At the expense of more states and/or the use of a larger alphabet, we could have made the operation even more straightforward, but we wished to take the opportunity to illustrate a few more tricks and techniques that can be used in Turing machine building. Some of these are hidden; the reader will discover, for example, that the symbols M and S are superfluous and could be replaced by X's throughout by just changing the letters in the diagram. If one is interested in constructing a very small universal machine, one can do much better; but then it is necessary sometimes to use the same state for several different purposes, and the machines become much more difficult to understand. One also may gain by making changes in the way the machine descriptions are represented on the tape. In chapter 14 we will combine such techniques to produce a very small machine—one which uses four symbols and only

seven states—but as the reader will see, that machine, or any like it, would be unsuitable for expository purposes.

Our universal Turing machine has the property of simulating the behavior of any Turing machine with any input tape. There is no need to interpret the result narrowly, in terms of any particular set of arithmetic operations; the Universal machine can simulate any Turing machine activity whatever. Accepting Turing's thesis, we conclude that the universal machine can simulate any effective process of symbol-manipulation, be it mathematical or anything else; it is a completely general instruction-obeying mechanism.

PROBLEM 7.4-1. Show how to get around the restriction that T, the machine being simulated, is limited to a tape infinite in only one direction.

PROBLEM 7.4-2. Compared to the machine T that it is simulating, the universal machine works very slowly—so slowly, indeed, that one would hardly expect to find in it any practical application (especially in view of how slow regular Turing machines are to begin with). And the relative speed of U, as compared to T, decreases as the length of the active part of T's tape grows, because U has to travel all the way down this length and back in each simulation cycle. We can redesign U to prevent this further decline. Sketch how this can be done.

Hint: Replace the symbol 'M' by a copy of the whole instruction region. Note that the machine U, and even the new machine just constructed, makes as efficient use of the tape as does T.

PROBLEM 7.4-3. Construct a Turing machine which, when started with a blank tape, prints its own description (as a sequence of quintuples). Chester Y. Lee [1963] calls machines which can print their own descriptions "introspective," although I would reserve this appellation for machines which print what they think is their own description.

Do not consult the solution unless desperate!

PROBLEM 7.4-4. Sketch out a reconstruction of U that uses unary rather than binary strings to represent T's states in the encoding of the quintuples.

8 LIMITATIONS OF EFFECTIVE COMPUTABILITY: SOME PROBLEMS NOT SOLVABLE BY INSTRUCTION-OBEYING MACHINES

The universal machine can do anything any other Turing machine can do, although more slowly. It would be of great value to have a precise characterization of just what kinds of behavior are possible, in as neat a form as we have for finite-state machines. (See chapter 4.) Unfortunately, this is a much more mysterious area, and we will see that there is definite evidence that it must remain so. In this chapter we will look at some of the things that we can prove Turing machines *cannot* do.

8.1 THE HALTING PROBLEM

When a Turing machine, with input tape, is started in operation it may be a very, very long time before it completes its computation and comes to a halt. For many machine-tape pairs this will *never* happen—the "computation" may go on forever. It would be useful to have a *decision procedure* which would enable us, given any machine T and tape t, to determine whether the process will ever halt.† We will show that there can be no effective procedure which can do this for us! Implicit in this argument is the commitment to Turing's thesis that Turing machine computations include all effective procedures. For we are assuming that if there is any decision procedure (the term always carries the connotation that the procedure is effective), then there must be a Turing machine which can carry out the procedure. Let us see what might be involved in such a procedure.

†The term *decision procedure*, means here a single set of instructions, given once and for all, that will enable us to solve the halting problem for *every* (machine, tape) pair.

We observe first that the universal machine gives a slight simplification of the problem. We are certainly free to place descriptions of T and t on U's tape and then set U into operation. If it is the case that T stops for t, then certainly U will eventually stop. But in the difficult case that T does not stop for t, U will never stop. Now if we had some way of looking at U and saying, "Aha! U is in a condition for which it will never stop," we would have a decision. Hence the general question "Does machine T halt on tape t?" can be *reduced* to a more special problem: "Does the *particular* machine U halt for the tape

Thus we have only to solve the "halting problem" for the tapes of one particular machine U. But we will show that there can be no effective procedure to do even this.

Another approach that seems promising, at first, is based on finding an upper bound on how long the computation can take if it is ultimately going to stop. Offhand, this would seem perfectly feasible. For T uses a known number S of symbols and a known number Q of states while t has only a known number N of non-blank squares. Surely we should be able to calculate some function of S, Q, and N such that, if the machine has not already stopped by time $f(S, Q, N)$ then it will never stop. Then all we have to do is to equip U with a way to compute this function $f(S, Q, N)$. Then U (or rather the improved U) can begin to imitate the behavior of T on t, while keeping count of the number of cycles. If the computation has not stopped by time $f(S, Q, N)$, then U can stop and announce that T's computation will never stop. This would be an effective decision procedure.

There is nothing wrong with this argument except in the innocent assumption that U can calculate the bound $f(S, Q, N)$! This bound certainly is meaningful,† but it turns out (as we shall see shortly) that no machine can compute it. This seems remarkable in view of the obvious fact that *any* function with values large enough will do.‡ But the bleak fact is that *no computable function can grow so large so fast*! In fact, since this argu-

†For there are only a finite number of Turing machines with S symbols and Q states, and for each of these only a finite number of initial tapes of length N. So there are only a finite number of computations associated with (S, Q, N) and hence only a finite number of halting computations. Then $f(S, Q, N)$ is just the length of the longest of this finite set of computations. One would ordinarily expect no difficulty in finding the largest of a finite set of numbers!

‡That is, we don't need to know the function f precisely. So far as the decision procedure is concerned, we could use any function $g(S, Q, N)$ such that $g(S, Q, N) \geq f(S, Q, N)$.

ment is otherwise correct, the proof below that there is no decision procedure for this problem will serve as our proof of the preceding assertion.

8.2 UNSOLVABILITY OF THE HALTING PROBLEM

The result of the simple but delicate argument in this section is perhaps the most important conclusion in this book. It should be contemplated until perfectly understood.

Let us suppose, *as a hypothesis to be proved self-contradictory*, that we have a machine D which will decide whether or not any Turing machine computation will ever halt, given the description d_T of machine T and its tape t. Then D has the form of Fig. 8.2-1.

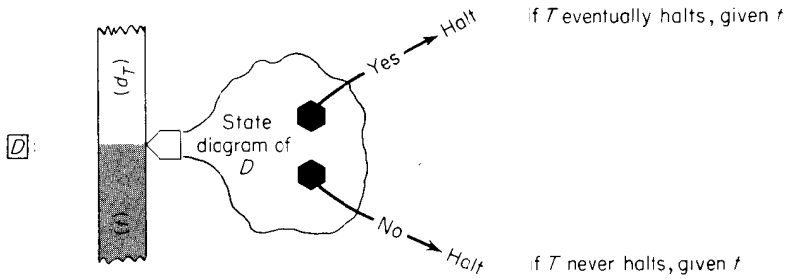


Fig. 8.2-1

If D can solve the halting problem for all description pairs (t, d_T), then it can certainly do it for the peculiar special pairs (d_T, d_T) where the tape t is a description d_T of T itself, rather than any old tape. (We need not concern ourselves with the question of why anyone would be interested in such introverted calculations; still there is nothing absurd about the notion of a man contemplating a description of his own brain.)

The two items of information previously required by D—the descriptions of T and of the tape t—are now the same, i.e., $d_T = t$. It is then easy to construct a new machine E which requires only the description d_T of T, but behaves otherwise like D. Suppose first that D, like U, is designed to use initial tapes of the form

Let E be made of D, together with some states which can copy a block of symbols. Then E can start with a tape

convert this to

and then turn matters over to D.

Whatever may be the inner structure of the hypothetical machine E, we know that its state diagram must contain two halt exits, one printing "Yes" if (d_T, d_T) ever halts, the other printing "No" if (d_T, d_T) never stops. Then E must have the form of Fig. 8-2-2. We now make a trifling

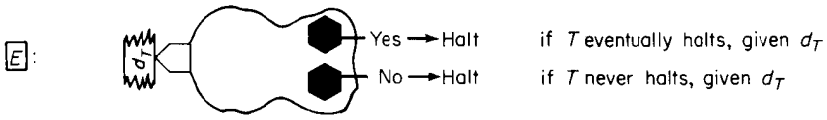


Fig. 8.2-2

change in E, by adding two new states, to obtain a new machine E^*; the new states prevent E^* from ever halting if E takes the "Yes" exit. (See Fig. 8.2-3.) E^* applied to d_T has the property that *it halts if T applied to d_T does not halt, and vice versa.* Now for the killer: *what happens if E^* is applied to the tape d_{E^*}? It halts if E^* applied to d_{E^*} does not halt, and vice versa.* This cannot be, so we must sadly conclude that such a machine as E^*, and hence E, and hence D, could not exist in the first place![1]

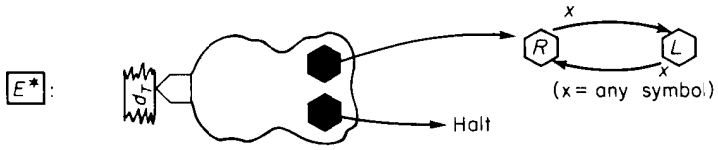


Fig. 8.2-3

We have only the deepest sympathy for those readers who have not encountered this type of simple yet mind-boggling argument before. It resembles the argument in "Russell's paradox" which forces us to discard arguments concerning "the class of all classes"—a notion whose absurdity is not otherwise particularly evident.[2] It is also related to the argument of Cantor, that one cannot "count" the real numbers. In fact the unsolvability of the halting problem is usually demonstrated by a form of Cantor's "diagonal method." We prefer the present form because it suppresses the "enumeration" required for the diagonal method. We will have to use the Cantor form of the argument in the next chapter.

8.3 SOME RELATED UNSOLVABLE DECISION PROBLEMS

Several other impossibility results are easily deduced from the basic halting problem result.

8.3.1 The halting problem

It follows immediately that we can't have a machine to answer the general question, "Does machine T halt for tape t?" because this would *include* the solution of the problem, "Does T halt for d_T?" discussed in the previous section.

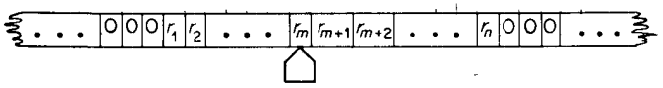
8.3.2 The printing problem

Consider the question, "Does machine T ever print the symbol S_0 when started on tape t?" This also cannot be decided, for all (T, t), by any machine. For we may take each machine T which does not ordinarily use the symbol S_0 and alter that machine so that before each of its halting exits it will first print S_0. Then the "printing problem" for the new machine is the same as the halting problem for the old machine. Since any machine that *does* use S_0 can first be converted to one that doesn't (by changing the name of S_0 to that of some unused symbol), and then altered as above, solution of the printing problem would give solution to *all* halting problems, and we know that that cannot be done.

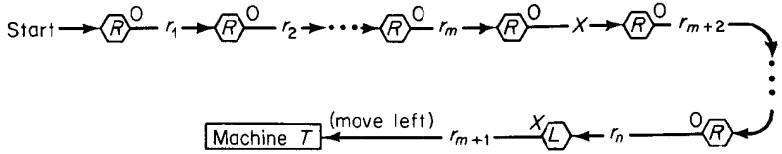
8.3.3 The blank-tape halting problem

Is it possible to build a machine which will decide, for each machine T, "*Does T halt if started on a blank tape*?" If there were a machine to solve the problem of 8.3.1 above, then that machine would solve the blank-tape problem as a special case. But the unsolvability of the problem of 8.3.1 does *not* immediately imply the unsolvability of the problem of 8.3.3 since the latter task might seem easier because of its apparently smaller domain of cases. However, we can show that the problems are equivalent by showing that for each machine-tape pair (T, t) there is a corresponding blank-tape halting problem for a certain other machine which we may call $M_{T,t}$.

The machine $M_{T,t}$ is constructed directly from the description of T and t by adding a string of new states to the diagram for T. Let us suppose that the computation T, t requires that T be started (in its initial state) at the indicated position of a tape:



The new machine $M_{T,t}$ will begin somewhere on a blank tape with the string of states



where 'X' is some letter not otherwise occurring on the input tape t. We see that $M_{T,t}$ must be equivalent, starting with a blank tape, to the machine T starting with tape t, because $M_{T,t}$ simply writes a copy of t on the tape, positions itself correctly, and then becomes identical with T from that time on.

It follows that if we could decide the halting of the blank-tape computations, i.e. solve the problem of 8.3.3, this would lead to the ability to decide about the machines $M_{T,t}$ and hence about all computations (T, t) of the problem of 8.3.1. Since this is impossible, the problem of 8.3.3 must also be unsolvable.

8.3.4 The uniform halting problem

Another interesting problem along this line is this: Can we decide for an arbitrary machine T, "Does T halt for *every* input tape?" It turns out that this problem includes the problem of 8.3.3 as a special case and is therefore unsolvable. We sketch the reduction. Let T be an arbitrary Turing machine which, as usual, is to be started in state Q_0. We construct another machine T^B by adding some states and quintuples to T. T^B starts in state q_0^*. Choose A and B to be two symbols not used by T. For each state q_i of T we adjoin the quintuples

$$(q_i, A, q_i', 0, L) \qquad \{q_i', \Phi, q_i, A, R\}$$
$$(q_i, B, q_i'', 0, R) \qquad \{q_i'', \Phi, q_i, B, L\}$$

where q_i' and q_i'' are two new states added for each q_i and Φ denotes *every* symbol (thus q_i' replaces the scanned symbol, whatever it may be, by A and then moves right to q_i). We also add three new states

$$\{q_0^*, \Phi, q_1^*, A, R,\} \qquad \{q_1^*, \Phi, q_2^*, 0, R\} \qquad \{q_2^*, \Phi, Q_0, B, L\}$$

and start the machine in q_0^*. Now we claim that the problem of whether T will halt on a blank tape is equivalent to the problem of whether T^B will halt on *every tape*; hence the problem of 8.3.4 is unsolvable.

PROBLEM 8.3.-1. Verify this last statement

PROBLEM 8.3-2. Show that the unsolvability of the problem of section 8.3.4 implies the unsolvability of the problem of section 8.3.1. (We just showed the converse.)

8.3.5 A related problem: Infinitely printed tapes

All the machines up to this point (and all that will follow) are considered to start in a given initial state with only a finite inscription on the tape. For a moment, consider machines which are given infinitely inscribed tapes. Does there exist a Turing machine D which can decide the following question: "Given a Turing machine T, is there an internal state Q and some *infinitely inscribed* tape for which T will not halt when started on that tape in state Q?" We have not discussed and will not further discuss machines with *non-finite* initial tape information, or unspecified initial states, but the reader may be interested in this easily stated but very difficult problem.[3]

PROBLEM 8.3-3. If the initial state *is* specified, the problem of whether there is a non-halting, infinitely inscribed tape is unsolvable. Show this.

8.4 THE CREATIVE CHARACTER OF THE UNSOLVABILITY ARGUMENT

Let me point out another interesting feature common to all of the problems above, though I describe it only for the halting problem. Consider the construction of E^* from E in 8.2. Now it is perfectly possible that some machine E, while not able to decide *every* question about halting, can decide *some* such questions. That is, we can suppose that when E says "Yes" then the (T, d_T) in question will halt, and that when E says "No" then the (T, d_T) in question will never halt, but that for some inputs E itself never halts and never announces a decision. There certainly do exist many such machines (Problem: Construct one.) which thus realize what we may call "partial decision procedures" in which all *announced* decisions are correct but the machine will not always commit itself. *Given a machine E for such a partial procedure, the construction of* 8.2 *gives us a particular problem which E cannot solve,* namely the halting problem of E^* for the tape d_{E^*}. Furthermore, this example is actually exhibited—there is no "non-constructive" step here (as there is in the Cantor diagonal argument). The inadequacy of any given alleged decision procedure can thus be demonstrated constructively.

Thus the problem of which situations (T, d_T) lead to halting is not only effectively unsolvable, but is associated with *an effective procedure for constructing counterexamples* for any proposed solution. Another problem of this character occurs in section 10.6, Theorem 5. Unsolvable problems of this somewhat arrogant character are studied in the theory of "creative" and "productive" sets—see Rogers [1966].

8.5 CONSEQUENCES FOR ALGORITHMS AND COMPUTER PROGRAMS: THE DEBUGGING PROBLEM

The unsolvability of the halting problem can be similarly demonstrated for any computation system (rather than just Turing machines) which can suitably manipulate data and interpret them as instructions. In particular, it is impossible to devise a uniform procedure, or computer program, which can look at any computer program and decide whether or not that program will ever terminate. This means that computation scientists cannot aspire to evolve a completely foolproof "debugging" program. (This observation holds only for programs in computers with essentially unlimited secondary storage, since otherwise the computer is a finite-state machine and then the halting problem is in fact solvable, at least in principle.) This means also that we cannot aspire to obtain a set of rules for deciding when *any* alleged "algorithm" is foolproof in the sense that it will terminate for all initial situations. Of course, we can find rules which work for large classes of important problems.

Those who object that Church's (or Turing's) thesis (see section 5.2) allows too much, usually do so on the grounds that the Turing machine formulation of computability allows computations whose lengths cannot be bounded in advance in any reasonable way. The impossibility of computing bounds (mentioned in section 8.1) that follows from the unsolvability of the halting problem is one of the obstacles that seems to stand in the way of finding a formulation of computability which is weaker yet not completely trivial.

8.6 NON-UNSOLVABILITY OF INDIVIDUAL HALTING PROBLEMS

We have shown the impossibility of an effective procedure to solve all Turing machine halting problems. But note that in no case did we show that the halting problem for any particular situation (T_0, t_0) is unsolvable, i.e., that there is no effective procedure which will tell whether or not (T_0, t_0) will ever halt. Indeed, no such result could be proven. For consider the two procedures, "Look at (T_0, t_0) and say 'Yes'," and "Look at (T_0, t_0) and say 'No'." These two trivial procedures may seem silly,

but each is certainly effective. And since (T_0, t_0) does in fact either halt or never halt, one of them must be correct! Of course the problem remains of finding which is the correct one, but it is important to observe that this is not the question that concerns us here.

Let us take a slightly different view of the matter. For a given (T_0, t_0), it could well be that no one will ever find out whether it halts or not. It could conceivably be that there is no way to find out, in some obscure sense. But it could *not* be that someone could *prove* that there is no way to find out. For that would lead to the following paradox:

Suppose that it had allegedly been proven that there is no way to find out if (T_0, t_0) halts or not. Suppose also that a great experimental project were launched, involving the construction and operation of (T_0, t_0). Sufficient funds are invested to provide for a slow, but surely never-ending supply of tape. Now there are two cases in fact—either (T_0, t_0) does eventually halt, or it never halts. In the first case, the experimental approach will ultimately succeed, and the question will be settled. This would certainly contradict any proof that the question could never be settled! Hence it must happen that (T_0, t_0) never stops. In other words, the proof that there is no way to find out can be used to prove that (T_0, t_0) never halts. Hence, we *would* be able to find out. Contradiction!

We will return to this discussion at the end of Chapter 9.

8.7 REDUCIBILITY OF ONE KIND OF UNSOLVABLE PROBLEM TO ANOTHER

In section 8.3.3 we showed the impossibility of constructing a Turing machine which can decide which Turing machines halt when started on blank tapes. This problem, of deciding which situations (T, blank) lead to a halt, seems simpler than the original problem 8.3.1 of deciding which situations (T, t) lead to a halt. But let us recall the steps of the proof. In 8.2 we showed that there is no machine which can decide all the (T, t) situations, by showing that, if there were such a machine D, there would also be a variant of it E^* which must both halt and not halt in the situation (E^*, d_{E^*})—when given its own description. We then showed that for each pair (T, t) there corresponds an easily constructed machine $M_{T,t}$ which halts on a blank tape if and only if (T, t) itself halts.

Therefore the ability to decide the special halting problems of the form (T, blank), which include all the situations ($M_{T,t}$, blank), would give us the ability to decide all of the (T, t) halting problems. We can say that in a very simple sense the apparently more difficult problem of the (T, t)'s is *reducible* to the apparently simpler problem of the (T, blank)'s.

This notion of *reducibility* has been studied very carefully in the modern theory of computability. It turns out that several technically different notions of reducibility are useful in studying the relations between different kinds of unsolvability problems and, further, that one can define infinite hierarchies of more and more difficult unsolvable problems, none of which is reducible to its predecessors. It has even been shown that there are pairs of unsolvable problems neither of which is reducible to the other. Rogers [1966] discusses these matters in detail.

8.8 PROBLEMS

PROBLEM 8.8-1. Consider the class of all 2-symbol (0 = blank, 1) machines T. Are there decision procedures for the following questions? That is, is there any machine D, not necessarily 2-symbol, that always halts with Yes or No, given T, t and answers the question.

(1) Does T ever print a 1 when started on tape t?

(2) Does T ever erase a 1 when started on tape t?

Hint: Apply finite-state machine theory to problem (1). Problem (2) is much, much harder: If you solve it you will not need to read the rest of the book!

PROBLEM 8.8-2. Is there a decision procedure that will decide:

(1) For any T, is there a tape on which T will halt?

(2) For any T, will T end with a blank tape for some input tape?

(3) For any two machines T and T', do they act the same, i.e., for every tape t is the ultimate outcome the same?

(4) For any T, does T started on t use more than N squares of tape?

NOTES

1. The argument in more detail is: if D exists then E exists, because it is easy to add a copying mechanism like that of Fig. 6.1-6. Similarly E^* exists if E does, because we can add the two loop states. Now let us put d_{E^*} on E^*'s tape. There are two possibilities:

   (1) E^* *eventually halts.* That is to say, E^* halts given d_{E^*}. Then E takes its "Yes" exit, given d_{E^*}. But this means that when E finishes its computation on d_{E^*}, it eventually enters its upper terminal state. Since E^* is the same as E up to this point, then E^*, given d_{E^*} eventually reaches *its* upper exit state and enters the never-halting loop. Therefore E^* *does not* halt, given d_{E^*}. This leaves only the second possibility:

   (2) E^* *never halts, given* d_{E^*}. But by a similar argument then E, given d_{E^*}, takes its lower exit; and so E^* (which is identical to E up to this point) takes its lower exit and *does* halt.

   Since there are no other possibilities, the existence of E^* itself is contradicted, hence that of E, hence the existence of D.

2. The "Russell paradox" is similar, where the existence of E^* is analogous, to

   C = The set of all sets **c**, for which **c** does not belong in **c**

   That is, **c** is a member of C if and only if **c** is not a member of itself. For example, the set of all sets is a set, so it is not a member of **C**. But the set of all books is not a book, so it *is* a member of **C**.
   We ask, "Does **C** belong in **C**?" If **C** *is* in **C**, then it does not belong in **C**. And if **C** is *not* in **C**, then it *does* belong in **C**! Although this may seem at first to be a sort of frivolous pun, it turns out to be a most serious indictment of ordinary, common-sense, ways of thinking about sets and relations; and in half a century of work on the foundations of logic, no technically easy way has been found to handle the problem. There are some apparently satisfactory, but technically quite difficult, ways to keep the paradox out of mathematical reasoning.

3. The problem was posed by Buchi [1962] and shown to be unsolvable by Hooper in his Ph.D. dissertation [1964].

9 THE COMPUTABLE REAL NUMBERS†

9.1 REVIEW OF THE REAL NUMBER SYSTEM

Most readers are probably not familiar with the details of the modern theory of the *real number system.* This section reviews some of the basic definitions. We assume as given the *integers* 0, 1, 2, ... The *rational numbers* are defined as quotients of integers, or more basically as equivalence classes of ordered pairs (m, n); two pairs (m_1, n_1) and (m_2, n_2) are in the same class if and only if $m_1 n_2 = m_2 n_1$.

There are two chief ways of defining *real numbers*, given the rationals. One is the method due to Dedekind; a real number is defined by a "cut" of all the rationals into two classes, such that each member of one class is less than every member of the other. Thus the real number π, whose decimal expansion begins with 3.14159265... is "defined" by two classes, one of which contains (among others)

> *3* and *3.1* and *3.14* and *3.141* and *3.1415* and *3.14159* and *3.141592*

and the other contains (among others)

> *4* and *3.2* and *3.15* and *3.142* and *3.1416* and *3.14160* and *3.141593*, etc.

Each of the numbers in the example is a rational number: 3.142 is equivalent to 3142/1000 (and also to 1571/500).

†This chapter is optional. It assumes that the reader has been exposed to the elementary theory of the real number system up to Cantor's proof that the reals are non-denumerable, etc. This background is not assumed in subsequent chapters.[1]

The same example shows how a real number can be defined by an increasing or decreasing convergent infinite sequence of rationals. And it can be shown that, if the reals are suitably defined as equivalent classes of convergent sequences of rationals, we get the same structure as that defined by the cut method. In particular one might define the "real decimal numbers" (or the "real binary numbers") in terms of infinite sequences of fractions with denominators that grow by powers of 10 (or 2) while the numerators grow one digit at a time; this may be shown to define the same structure of the real numbers. (One has to provide for the equivalence, e.g., of 1.0000... with 0.9999..., but otherwise there is no difficulty.) The definition by sequences is the method of Cauchy.

9.2 THE (TURING)-COMPUTABLE REAL NUMBERS

We define the *computable real numbers*, in a manner parallel to that of defining the (Cauchy) real numbers, as sequences of digits interpreted as decimal fractions.† But we add one key restriction. *The digits must be generated sequentially by a Turing machine.* That is, we require that in order that the real number $.a_0a_1a_2\ldots$ be a *computable real number*, there must be a Turing machine which starts with a blank tape and prints out a tape of the form

$$\ldots 000\,Xa_0Xa_1Xa_2X\ldots 000\ldots$$

We make the rule that, once an X is printed, the machine must never move to the left of, or change, that X. It may use any amount of tape to the right for its computation, but the printing of an X is an irrevocable announcement that a digit has been computed and is in the square to the left of the X.‡

How is this different from the definition of a real number? It is different because in the definition of a *computable* real number the defining sequence must be determined by a *finite* amount of information, i.e., the state diagram of the associated Turing machine. There is no such limitation for the ordinary real number. That this is a serious difference will be seen shortly.

PROBLEM 9.2-1. Show that any *rational* number can be so defined by a Turing machine which never prints anything except the digits and the X's.

†We will treat only numbers between 0 and 1. This simplifies notation because it allows us to assume the decimal point is always at the extreme left and does not otherwise affect the theory significantly.

‡We need some such rule. Without such a proviso one could never be sure that a digit is not going to be changed at some later time. If this uncertainty always remained, one could hardly accept the process as an effective definition of the number, for one would never really know for sure anything about its value.

PROBLEM 9.2-2. We could also define a computable number to be one for which there is a Turing machine which, given n on its initial tape, terminates with the nth digit of that number. Show that this definition is equivalent to that of 9.2.

9.3 THE EXISTENCE OF NON-COMPUTABLE REAL NUMBERS

We can show the existence of non-computable real numbers in two rather different ways. Our first demonstration is through appeal to the existence of unsolvable halting problems for Turing machines.

9.3.1 A particular non-computable real number R_U

Consider the behavior of our universal machine U on some tape inscribed with a finite sequence W composed of the ten symbols 0, 1, X, Y, A, B, M, N, R, and S. We know that we cannot decide, for all such tapes, whether U will eventually halt.† (This follows from section 8.3.3, because this set of tapes includes the class of all descriptions of Turing machines starting with blank tapes.)

Now let us think of these tapes as representing numbers. We identify the symbols 0, 1, X, Y, A, B, M, N, R, S with the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. *Then to each tape there corresponds a decimal number.* Thus the tape

$$0000M000Y01X00001X01110X10011X11100Y000\ldots$$

becomes the (decimal) number

$$.000301200001201110210011211 1003$$

where we make the convention that the first M will be a decimal point. Then, also, *to each decimal number there will correspond such a tape.* Hence we can now talk of the tape corresponding to the number n and vice versa.

We now define our non-computable number R_U in terms of the (countably infinite) set of Turing-machine tapes. R_U will begin with the decimal point and, going to the right, its nth digit is

1 if U *halts* on the nth tape;

0 if U *never halts* on the nth tape.

To show that the real number so defined is non-computable, we derive a contradiction from the supposition that there is a Turing machine T_{R_U}

†We assume some convention about where U starts on the tape, e.g., that it starts to the right of the sequence. The details aren't important.

which can compute it. If there were a T_{R_U} machine, then we could build from it a somewhat more complicated machine M which can decide whether U halts on an arbitrary tape. Since there can be no machine M which can do this for all tapes (see section 8.2), and since M is built in a straightforward way from T_{R_U}, it follows that T_{R_U} could not exist.

PROBLEM 9.3-1. Sketch how M can be built from T_{R_U}. Carrying out the full construction involves some complications, but is quite instructive.

9.3.2 The set of computable-real numbers is countably infinite

In the argument of 9.3.1 above, we observed that the tapes for the universal machine U could be set into a one-to-one correspondence with (some of) the integers. Now the number of such tapes is *countably* infinite, hence *there is only a countably infinite set of computable-real numbers.* But it is well known that there are an *uncountable* number of real numbers. Hence there must be some non-computable real numbers, and indeed it follows that "almost all" real numbers must be non-computable.[1]

The argument of section 9.3.1, while slightly more complex, seems to give slightly more of a result. In a way it actually tells us about a particular non-computable number. Unfortunately, but inevitably, we can't get very close to that number, because the very proof that it is non-computable tells us that we can't get a Turing machine, or any equivalent, to give us its sequence of digits. Yet one could hardly ask for a more "constructive" proof of the existence of such an intangible object and still subscribe to anything resembling Turing's thesis.[†]

9.4 THE COMPUTABLE NUMBERS, WHILE COMPUTABLE, CANNOT BE EFFECTIVELY ENUMERATED!

An interesting property of the computable real numbers is that, while like the rational numbers they form a countable, or "enumerable," set, they are unlike the rationals in that they cannot be "effectively enumerated." That is, they cannot be all arranged in a sequence which can be "represented" by a Turing machine.

[†] That is, the argument really does "define" one and only one real number. For either U halts, or it does not halt, on the Nth tape. This completely determines all the digits of our non-computable number R_U. Many mathematicians and philosophers feel, nonetheless, that there is a serious question as to whether one ought to believe that R_U's "existence" is really established.

DEFINITION

We say that an infinite sequence $c_1, c_2, \ldots, c_n, \ldots$ of real numbers is *represented* by a Turing machine T if, given a representation of any integer n, T will print out the digits of the number c_n, in the manner described in section 9.2.

If there were such a representing machine T for *all* the computable real numbers then, by a construction similar to that of M from T_{R_U} in 9.3.1, we could make a machine T' which, given m and n, would give us the mth digit c_{nm} of the nth computable number. In particular, then, we could make a machine T'' which, given m, would give us the mth digit c_{mm} of the mth computable number. Next we could modify that machine to make a machine T''' that would print out the sequence

$$\ldots 000Xc_{11}Xc_{22}X\ldots Xc_{mm}X\ldots$$

as a computable real number. Having got so far, we make one final change in our machine: The machine T'''' is like T''' except that for each m, if c_{mm} is '1', the new machine T'''' prints '2' for it; if c_{mn} is not '1', then T'''' prints '1' for it. Now consider the array of the digits of all the computable numbers $c_1, c_2, \ldots c_n, \ldots$

$$\begin{array}{lccccc}
c_1 = & (c_{11}) & c_{12} & c_{13} & c_{14} & \ldots \\
c_2 = & c_{21} & (c_{22}) & c_{23} & c_{24} & \ldots \\
c_3 = & c_{31} & c_{32} & (c_{33}) & c_{34} & \ldots \\
c_4 = & c_{41} & c_{42} & c_{43} & (c_{44}) & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}$$

where we have circled the "diagonal" digits c_{mm}. Note that the machine T'''' prints a sequence whose mth digit is *never* equal to the corresponding c_{mm}. The fact that *each diagonal entry has been changed* means that the sequence obtained by going along the new diagonal must be different (in at least one digit) from each and every row sequence. Since we have given an effective procedure for writing its digits, this sequence defines a computable number. But this means that our final Turing machine prints a computable number that did not occur in the allegedly complete sequence of computable numbers "represented" by T. This contradiction implies that no Turing machine can represent the entire set of computable numbers, hence this set cannot be effectively enumerated.

Students familiar with Cantor's theory of infinite sets should note that, although we have employed Cantor's "diagonal method," we have *not* shown that there are an *uncountable* number of computable numbers! All we have shown is that the computable numbers cannot be "counted" *by any Turing machine* (or other effective process). Their number is still only

that of the integers; they *could* be counted by an omniscient "oracle" or an infinite computer. Cantor's argument for the *uncountability* of the real numbers depends on applying the "diagonalization method" to *any enumeration whatever*, without the restriction to "effective" or "computable" enumerations.

9.5 DESCRIPTIONS AND COMPUTABLE NUMBERS†

Imagine for the moment that man has evolved a perfectly unambiguous written language through which he can communicate all his intuitive notions. The expressions of this language are finite sequences of symbols from some (fixed) alphabet. A *description* of a real number is an expression in this language which in some clearly understood way determines, selects, generates, or allows us to recognize that number and only that number. We call *describable* any number which has a description.

Warning: This intuitive notion is so vague that one may wonder whether it could have any useful precisely defined counterpart. At present this seems very unlikely, since no one has been able to point toward a way of excluding logical paradoxes, while preserving all or most of the descriptive freedom of "natural" language. Let us explore this notion, though, as if it were well defined, and see what happens.

The set of describable real numbers is surely a countably infinite set, for any description is a finite string of symbols. Hence all descriptions can be enumerated, e.g., by listing first the finite set of expressions of length 1, then those of length 2, etc. Of course not all strings will be descriptions of numbers but this doesn't matter. It follows that some real numbers are worse than non-computable; some (indeed, almost all) are non-describable—there aren't enough descriptions to go around! The computable numbers are certainly describable; hence (as argued in 9.3.2), they are countable. Now let us go on with this speculative theory.

9.5.1 Some descriptions of numbers do not describe computable numbers

Consider the non-computable number R_U described in 9.3.1. In English, it is described by:

$$\text{The } n\text{th digit of } R_U = \begin{cases} 1 & \text{if } U \text{ halts on the } n\text{th tape.} \\ 0 & \text{if } U \text{ never halts on the } n\text{th tape.} \end{cases}$$

†This section is independent of what follows and is probably not entirely sound! It is interesting to speculate on what happens if we try to build up an analysis for *description* along the lines of our analysis of effective *computation*. The subject of description has been treated in many other ways.[2]

Hence R_U is *describable* even though we can't find all its values by an effective procedure. *Remark:* We would expect no particular difficulty in describing, in super-English, what is a Turing machine.

PROBLEM. Let T be any Turing machine, and let R_T be the describable number defined by

$$\text{The } n\text{th digit of } R_T = \begin{cases} 1 & \text{if } T \text{ halts on the } n\text{th tape.} \\ 0 & \text{if } T \text{ never halts on the } n\text{th tape.} \end{cases}$$

Prove that no effective procedure can decide, for every machine T, whether R_T is computable.

It follows from this problem that no effective procedure can determine which describable numbers are computable and which are not. In fact, in general it is not possible to decide which expressions describe numbers at all.

Quite frequently, of course, we can show that *some particular description* specifies a computable number. For example, the description of e as

$$\sum_{n=0}^{\infty} \frac{1}{n!}$$

makes it possible to show that (the sequence of digits of) e is computable.

PROBLEM. Show that e is computable! This requires an analysis to find out when the representing machine can print an X.

9.5.2 Some descriptions of computable numbers aren't effective

Frequently we can describe a computable number in a manner which is completely ineffective in itself—which doesn't give the slightest hint as to how to compute that number. Consider the following non-effective description of a number N: $N = 1$ if there are an infinite number of 5's in the decimal expansion of π, and $N = 0$ if not.

Now N is either 0 or it is 1—there is no other possibility, and it must surely be only one of those. *In either case it is a computable number, for 1 is and 0 is.* But neither the author nor the reader knows which one it is, nor does anyone else! I'm sure that the vast majority of mathematicians would bet, for various reasons, that $N = 1$; but that isn't much help. It seems perfectly possible to me that no one will ever find out the answer. Perhaps time will *not* tell.

This raises some intensely puzzling questions. Up to now each of our "unsolvable problems" has been concerned with some infinite set of decisions. We have never shown that the halting problem for any *particular*

blank-tape Turing machine was unsolvable, but only that we could not hope to find any "uniform" procedure to solve *all* such problems. Some mathematicians accept these uniform unsolvability results but prefer to believe that there can be no analogue for single problems. They seem to feel that even if present methods fail to solve a problem, we will always be adding to our mathematical resources and, through such additions, can eventually solve all particular problems. (It is clear that such a complete evolution of our methods cannot occur in accord with any *effective* process, so such faith must depend on some unexplained optimistic belief in invention.)

Returning to the problem of the value of N, it is clear that this problem *cannot* be solved by brute computation of the digits of π to enormous precision. For at any time in the computation one will have only a finite number of 5's, and this provides no evidence either way. If one finds no more 5's for a very long time one may become discouraged, yet one may be just about to strike oil; while if one gets many 5's one can still draw no conclusion—the well may be about to run dry! To settle the question will require some form of mathematical *proof*—a more abstract kind of prospecting.

A little earlier we mentioned that it seems possible to us that this question will never be solved. It seems also possible, and this is quite different, that it *cannot* be settled. The first gloomy suggestion is that we will never happen to find a proof. The second is that perhaps there is no proof—within the framework of any "plausible" extension of what we recognize today as valid mathematical reasoning. A third, even more obscure possibility, is that one might *prove* that the question can't be settled. To explore carefully what such a result could mean, and what form it could take, would carry us well beyond what can be discussed in this informal framework, and no one seems to understand it terribly well, anyway.

We will discuss instead some problems whose solvability status is clear yet sometimes misunderstood:

(1) *Fermat's Last Theorem cannot be proved unsolvable.* This is like the situation in section 8.6. The question is: "Are there any integers x, y, z, and w, for which

$$x^w + y^w = z^w$$

with w larger than 2?" This question might, of course, never be solved—i.e., no one might ever find such a set of x, y, z, and w. But it cannot be proved unsolvable. For if there were such a proof, this would be inconsistent with the existence of an x, y, z, w solution; hence such a solution could not exist; hence the question would be answered "No," and this would solve the problem.

(2) *The $\pi(n)$ function must be computable, even though it may be impossible to find out how to compute it.* Define the function:

$$\pi(n) = \begin{cases} 1 & \text{if the decimal expansion of } \pi \text{ has } n \text{ 5's in a sequence.} \\ 0 & \text{if not.} \end{cases}$$

Now we observe that

$$\begin{array}{lll} \text{(i) if} & \pi(n) = 1 \quad \text{then} & \pi(n-1) = 1 \qquad (n > 0) \\ \text{(ii) if} & \pi(n) = 0 \quad \text{then} & \pi(n+1) = 0 \end{array}$$

So the values of $\pi(n)$ have either the form

$$n = 1, 2, 3, 4, \ldots, k, k+1, k+2, \ldots$$
$$\pi(n) = 1, 1, 1, 1, \ldots, 1, \quad 0, \quad 0,$$

for some k, or else $\pi(n) = 1$ for *all* n. No one knows. Perhaps we will never discover a way to find out what k is, if it has a finite value. But $\pi(n)$ is computable in any case, for (i) if there is no k, then

$$\pi(n) \equiv 1$$

which certainly is computable, and (ii) if there is a k, then

$$\pi(n) = \begin{cases} 1 & \text{if } n \leq k \\ 0 & \text{if } n > k \end{cases}$$

which function certainly is also computable. The point is for the reader to note the curious disconnection between knowing a function is "computable" and knowing how to compute it!

(3) *It is not known whether $\pi'(n)$ is computable.* Define

$$\pi'(n) = \begin{cases} 1 & \text{if the number } n \text{ occurs within the decimal expansion of } \pi. \\ 0 & \text{otherwise.} \end{cases}$$

Then, for example,

$$\pi'(141) = 1, \quad \pi'(4159) = 1, \quad \pi'(26535) = 1, \quad \text{etc.}$$

We know these because we see them in known parts of the decimal expansion of π. But we don't know that $\pi'(n)$ is computable. To find out, I suppose, would require a major advance in the mathematical theory of transcendental numbers rather than an advance in the theory of computable functions. Certainly the kind of argument, in the previous paragraph, about $\pi(n)$ simply doesn't work here. Most mathematicians would surely suppose that $\pi'(n) \equiv 1$ for all n (and hence, incidentally, that $\pi(n) \equiv 1$ for all n), but no one knows.

9.5.3 Mathematics and metamathematics

The conclusion of Section 8.6 is probably sound. We cannot expect to find consistent informal arguments proving that the type of particular—one-answer—problems considered there are unsolvable. But there is something peculiar about the argument we used to show this. We become aware of this peculiarity in two important ways. First, in other areas there are arguments of very much the same character which seem equally reasonable, yet give us absurd results; this makes us suspicious of all such arguments. This is why I said that the argument in question is only *probably* sound.

We see the peculiarity more clearly when we try to replace the informal argument by a "mathematically rigorous" demonstration, e.g., one like a proof of Euclid in which each step of the argument is "authorized" by referring to some previously-agreed-on "axiom." (Actually, Euclid's proofs must be further justified by references to "rules of inference" which specify how new theorems may be generated by using old theorems and axioms.) When we do this, we become aware that our "proof" refers explicitly to the notion of proof itself. But in what axiom system? *Is the system the proof talks about the same system that the proof is constructed in*? If not, the argument loses much of its force, for even if a problem is unsolvable (that is, if a statement is neither provable nor disprovable) in one system, there would be nothing surprising about finding that it can be settled in a larger system—e.g., one which contains *that statement* as an additional axiom.

On the other hand, if the proof systems are the same, so that statements can, in effect, talk about themselves or about their own proofs, we must be prepared to find grave difficulties. Historically, most such systems of "metamathematics"—axiom systems concerned with theorem-proving itself—have turned out to be inconsistent, producing self-contradictions. Worse, it was shown by Gödel that if, within any such system, one can *prove* that this will not happen—i.e., the system asserts its own consistency—then surely it will in fact be inconsistent. Hence one cannot believe the most persuasive argument of a mathematical system in its own defense! Gödel's methods are essentially similar to those we use in chapter 8, but somewhat complicated by the additional machinery needed to talk about theorems and proofs.

This subject is quite complicated and technical, and the reader who wishes to pursue it is referred to Rogers [1966].

The following well-known "paradox" illustrates this kind of difficulty with descriptions very nicely. Consider "*the smallest integer whose description requires more than ten words.*" Does this description define any integer? It ought to, since we know that every set of (positive) integers must have a smallest member. And why should we not be able to consider

the set of integers whose (smallest) descriptions are longer than ten words? But since the quoted expression has itself just ten words we have a paradox. The trouble comes when we attempt carefully to formalize the notion of description. There is no trouble with a straightforward representation of numbers, e.g., as strings of digits in the usual way. Then, e.g., 100 is the smallest integer not representable by less than three digits. But we get into trouble when the expressions are, as above, interpreted on the metamathematical level to talk about themselves. Since we cannot expect to be able to embed such statements in a consistent logical system, we need not be surprised at "paradoxes" like this: one expects them when the logic itself is defective.

9.6 PROBLEMS ABOUT COMPUTABLE NUMBERS

PROBLEM 9.6-1. Prove that, if x is a computable number, then so are $x/2$, x^2, $\sin x$.

PROBLEM 9.6.2. Define the *computable-complex numbers* and sketch a proof that they form an algebraically closed field—that, if $a_0, \ldots, a_n$ are computable-complex, so are the roots of the equation

$$a_0 + a_1 z + \cdots + a_n z^n = 0$$

Hint: Show that any standard computation method, e.g., Newton's method, yields an effective procedure for finding the decimal digits of the roots. Ignore the difficulties that arise for those roots that are rational, since they are computable anyway.

PROBLEM 9.6-3. Define a *real-computable function* in the following way: The function $f(x)$ is real-computable if there exists a Turing machine which, given $x = .a_1 a_2 a_3 \ldots$, together with an arbitrary integer n, in the form of an *infinite* initial tape

$$\ldots 000 \ldots 0 Y^n X a_1 X a_2 X a_3 X \ldots X a_j X \ldots$$

will always terminate with a tape of the form

$$\ldots 000 \ldots Y^{f(x;n)} X a_1 X a_2 X \ldots$$

where Y^n is a block of n Y's and $f(x; n)$ is the nth digit of $f(x)$. (If this seems to be too specialized a definition, one might just say that $f(x)$ is real-computable if there is an effective procedure which finds (for each n) the nth digit of $f(x)$, given the expansion of x on a semi-infinite tape.)

Prove that any real-computable function is continuous in the ordinary ϵ, δ sense! Hint: Show that the discontinuous function

$$f(x) = \begin{cases} .000 \ldots & \text{if } x < \frac{1}{3} = .333 \ldots \\ .100 \ldots & \text{if } x \geq \frac{1}{3} = .333 \ldots \end{cases}$$

isn't real-computable (and similarly that a real-computable function can't have a discontinuity at any point). How can the machine decide what is the first digit of $f(.333\ldots)$?

NOTES

1. A good introduction to the theory of infinite numbers, and to the real numbers, is Courant and Robbins [1961, Chap. 5].
2. See, for example, Quine [1960] and his further references. For a most interesting ambitious, and unconventional, if not entirely complete, attempt to show how everyday intuitive ideas could be incorporated in a formal mathematical system, see Freudenthal [1961].

10 THE RELATIONS BETWEEN TURING MACHINES AND RECURSIVE FUNCTIONS

10.0 INTRODUCTION

In this and the next chapter we explore a number of formulations of the notion of effective computability—and show that they all define the same class of computations, or procedures. Historically, many of these notions arose independently. The fact that they all turned out to be equivalent is one of our major reasons for confidence in arguments using Turing's thesis. And even if Turing computability should not agree perfectly with everyone's intuitive ideas about what processes are effective, this multiple discovery and equivalence makes it certain that this particular class of processes has an important significance. Its study—the branch of mathematics called *theory of recursive functions*—is one of the most elegant and self-contained mathematical developments of the twentieth century. It also provides a certain amount of practical insight in understanding the nature and limitations of computer programs.

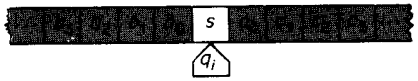
This chapter develops the notions of *primitive-recursive function* and *general-recursive function* and shows how the latter notion is at least as comprehensive as that of a Turing-machine-computable function. The mathematical method for showing this is a curious one, called "arithmetization." Arithmetization is very simple, but there is something artificial and unnatural about it. While the logic is easy to follow, one feels that the whole thing is a sort of extended joke or pun, and that there ought to be a more lucid analysis that yields the same end result but gives more insight into its nature. Later we will outline some more "natural" methods, but for now our main concern is to get certain results within the framework developed up to this point.

10.1 ARITHMETIZATION OF TURING MACHINES

We want first to show that there is a sort of equivalent to Turing machines within a rather elementary system of arithmetic—that is, to show that one can build the computable numbers and functions on simple ideas about *numbers* rather than about *machines*. We will use a simple arithmetical technique to show how a numerical formulation of the notion of effective computability yields essentially the same theory as that based on machines.

10.1.1 Representation of the Turing-machine conditions in terms of quadruples of integers

Consider a Turing machine at some time t in the course of a computation; it will be in some state $q(t)$ reading some symbol $s(t)$ on the tape. The tape will have a finite number of symbols written on it and will be otherwise blank. It is convenient to restrict ourselves to the binary Turing machines and to use '0' for the blanks and '1' for the other symbol. Then the tape will have the appearance



where we find it convenient to think of the tape as composed of two (infinite) sequences extending in either direction from the location of the reading head. Now, if we take each of these sequences to represent the digits of a binary number, then we can regard the tape as represented by three integers (s, m, n), where s is the digit under the reading head and

$$m = \sum_0^\infty b_i 2^i \qquad n = \sum_0^\infty c_i 2^i \qquad \begin{pmatrix}\text{each } b_i \text{ and } c_i \\ \text{is 1 or 0}\end{pmatrix}$$

For example, if the tape is

$$\ldots 00010111101111000\ldots$$

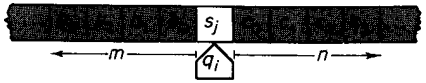
then $(s, m, n) = (0, 47, 15)$. These "infinite" sums are really finite, since only a finite number of b_i's and c_i's are different from zero. If we further represent each state q_i by the corresponding integer i, then we can represent the *complete state* of the Turing machine condition at each moment by a quadruple of integers $[q, s, m, n]$.†

Each operation cycle of the Turing machine has the effect of transforming the numbers $[q(t), s(t), m(t), n(t)]$ into a new quadruple

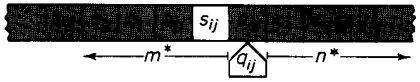
†This representation is based on one used by Wang [1957].

$[q(t+1), s(t+1), m(t+1), n(t+1)]$. Since the quadruples are complete descriptions of the machine conditions, *knowing the details of this transformation is equivalent to knowing the structure of the Turing machine involved.* In fact, the transformation is related to the Turing machine's quintuples in the following way:

Consider a quintuple $(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$ and suppose for the moment that $d_{ij} = 1$; that is, the machine is to move to the right. Then the tape



will become



The new value q^* for q is q_{ij}. The new value s^* is the old c_0. The new value m^* of m is found by noticing that all the b_i's have been moved up one binary position (doubling their values in the binary sum) and s_{ij} fills in the least significant digit; hence

$$\begin{aligned} m^* &= s_{ij} + \sum_0^\infty b_i 2^{i+1} \\ &= s_{ij} + 2\sum_0^\infty b_i 2^i \\ &= s_{ij} + 2m \end{aligned}$$

Similarly to find n^*, the new value of n, c_0 is removed and the other digits are moved down one binary position.

$$\begin{aligned} n^* &= \sum_1^\infty c_i 2_{i-1} \\ &= \tfrac{1}{2}\sum_0^\infty c_i 2^i - \tfrac{1}{2}c_0 \\ &= \tfrac{1}{2}(n - c_0) \end{aligned}$$

We can summarize all this by writing

$$\left.\begin{aligned} q^* &= q_{ij} \\ s^* &= P(n) \\ m^* &= s_{ij} + 2m \\ n^* &= H(n) \end{aligned}\right\} T_{\text{right}}$$

where we define $H(n)$ to be the largest integer in $n/2$, and $P(n)$ to be 0 if n is even, 1 if n is odd. That is,

$$\begin{aligned} P(0) &= 0 \qquad H(0) = 0 \\ P(1) &= 1 \qquad H(1) = 0 \\ P(2) &= 0 \qquad H(2) = 1 \\ P(3) &= 1 \qquad H(3) = 1 \\ P(4) &= 0 \qquad H(4) = 2, \text{ etc.} \end{aligned}$$

and, in general, $n = P(n) + 2\,H(n)$. What this shows is that the transformation involves only the simplest arithmetic ideas—addition (by unity), multiplication (by 2), and the quotient $H(n)$ and remainder $P(n)$ of division (by 2).

If the direction d_{ij} happens to be '0'—"move left"—then we need only exchange the roles of m and n in the above transformation equations.

10.1.2 The Turing transformation

Now we will complete the representation of the Turing machine in terms of arithmetic operations. We can represent the quintuple structure of a given two-symbol Turing machine by redefining the three functions (see section 6.0)

$$\begin{aligned} Q(q_i, s_j) &= q_{ij} \qquad \text{STATE} \\ R(q_i, s_j) &= s_{ij} \qquad \text{SYMBOL} \\ D(q_i, s_j) &= d_{ij} \qquad \text{DIRECTION} \end{aligned}$$

in terms of integer-valued functions of integers. They need be defined only over the finite set of value pairs

$$\begin{array}{lll} (0,0) & (1,0) & (k-1,0) \\ (0,1)' & (1,1)'\cdots' & (k-1,1) \end{array}$$

where k is the number of states of the two-symbol Turing machine. D and R take on only the two values 0 and 1, while Q can assume values between 0 and $k-1$. It will be convenient also to have the function

$$\overline{D}(q, s) = 1 - D(q, s)$$

such that $\overline{D} = 1$ for "move left" and $\overline{D} = 0$ for "move right"—just the opposite of D. We need also the division-by-2 functions, just defined:

$$H(x) = \text{quotient of } \frac{x}{2} \qquad \boxed{\text{HALF}}$$

$$P(x) = \text{remainder of } \frac{x}{2} \qquad \boxed{\text{PARITY}}$$

Suppose, finally, we have the two arithmetic functions

$$F^{+}(x, y) = x + y \qquad \boxed{\text{ADD}}$$

$$F^{\times}(x, y) = x \cdot y \qquad \boxed{\text{MULTIPLY}}$$

Now with these we can express our transformation in a uniform way, without any external statement about the value of d_{ij}:

$$\begin{aligned} q^* &= Q(q, s) \\ s^* &= P(m) \cdot \overline{D}(q, s) + P(n) \cdot D(q, s) \\ m^* &= [2m + R(q, s)] \cdot D(q, s) + H(m) \cdot \overline{D}(q, s) \\ n^* &= [2n + R(q, s)] \cdot \overline{D}(q, s) + H(n) \cdot D(q, s) \end{aligned}$$

Observe the simple-minded trick used here. If the instruction is really "move right," then $D(q, s) = 1$ and $\overline{D}(q, s) = 0$. In that case these equations are exactly the same as those of T_{right}. In the "move left" case, just the appropriate changes are made. While this works, it has a certain logical opacity (about which we complained in the introduction of this chapter). The "pun" is that '+' is used as a sort of logical "or" and '×' is used as a logical "and," somewhat as in Boolean Algebra.

We have expressed the Turing transformation entirely in terms of simple arithmetic functions. Indeed, (although it may seem a little tedious) we can write each equation explicitly as, for instance,

$$m^*(q, s, m, n) = F^{+}(F^{\times}(F^{+}(F^{\times}(2, m), R(q, s)), D(q, s)), F^{\times}(H(m), \overline{D}(2, s)))$$

This is written out, not for clarity, but to demonstrate that we really can express the transformation equations in terms of the given functions combined by several applications of the "composition" operation.†

10.1.3 Reduction to the zero and successor functions

Next we show that by the addition of more equations we can eliminate many of the basic ingredients of the above system! We do this by starting with certain very simple functions and defining the required new functions by *induction*. In fact, using induction, all we need to start with is the "zero function"

$$0(x) = 0 \qquad \boxed{\text{ZERO FUNCTION}}$$

† *Composition* is the operation of substituting function names for variables in other function names. A more formal treatment of this is given in Kleene [1952], but the details aren't important here. What is important is to observe how we make new functions by using composition operations on old, simpler, functions.

and the "successor function" (for which it is convenient to have two notations):

$$S(x) = x + 1 = x' \qquad \boxed{\text{SUCCESSOR FUNCTION}}$$

Then we can define the addition function $F^+(y, x)$ by the equations:

$$\begin{cases} F^+(0, x) = x & \boxed{\text{BASE}} \\ F^+(y', x) = (F^+(y,x))' = S(F^+(y,x)) & \boxed{\text{INDUCTION FORMULA}} \end{cases}$$

Let us be sure to understand these equations by seeing how they give the value, say, of $3 + 6 = F^+(3, 6)$. We find that

$$\begin{aligned} F^+(3,6) &= (F^+(2,6))' && \text{or} \quad S(F^+(2,6)) \\ &= ((F^+(1,6))')' && S(S(F^+(1,6))) \\ &= (((F^+(0,6))')')' && S(S(S(F^+(0,6)))) \\ &= (((6)')')' && S(S(S(6))) \\ &= ((7)')' = (8)' = 9 && S(S(7)) = S(8) = 9 \end{aligned}$$

It can be seen that the values of a function so defined are determined—that the calculation must eventually terminate—because of the constant decrease of the value of the induction variable.

We can define the multiplication function $F^\times(y,x)$ similarly:

$$\begin{cases} F^\times(0,x) = 0 \\ F^\times(y',x) = F^+(F^\times(y,x),x) \end{cases}$$

Of course, we can define any particular *constant* as a function:

$$C_4(x) \equiv S(S(S(S(0(x))))) = 0'''' = 4.$$

It is convenient to define the function $N(x)$, which is a '0'-detector:

$$\left.\begin{cases} N(0) = 1 \\ N(y') = 0(y) = 0 \end{cases}\right\} \qquad \boxed{\text{NULL FUNCTION}}$$

because then we can easily obtain $P(x)$ and $H(x)$:

$$\begin{cases} P(0) = 0 \\ P(y') = N(P(y)) \end{cases} \qquad \begin{cases} H(0) = 0 \\ H(y') = F^+(H(y), P(y)) \end{cases}$$

10.2 THE PRIMITIVE-RECURSIVE FUNCTIONS

All of our inductive definitions in the previous section have the same general form. There is one induction variable, y, and perhaps some other variables $x_2, \ldots, x_n$ regarded as parameters, that is, as numbers fixed throughout the definition. (In each case above there was either no such parameter or one.) In each inductive definition of a function, there appear two other functions which are assumed to have been already defined: *a base function* ψ and an *inductive step function* χ.†

$$\begin{cases} \phi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n) \\ \phi(y', x_2, \ldots, x_n) = \chi(\phi(y, x_2, \ldots, x_n), y, x_2, \ldots, x_n) \end{cases} \qquad \boxed{\text{PR}}$$

The important thing is that in each step the value of the right-hand side may be calculated in terms of already defined functional values. In any particular instance of this form, the parameters may be absent, either from ϕ itself, or as explicit variables in ψ and χ.

This introduction form "PR" is called *primitive-recursion*.

DEFINITION

> *Any function that can be defined in terms of* $0(x)$, $S(x)$, *composition, and primitive recursion is called a primitive-recursive function.*

We have almost completed the proof that *the functions* q^*, s^*, m^*, and n^* *are all primitive-recursive.* Only, we have not yet accounted for the "fixed" functions Q, R, D, and $\overline{D}$—that is, to show that they are primitive-recursive functions. To show how this is done we show first how to define a fixed finite function of one variable. Suppose, for example, that we want to define a function $W(x)$ so that

$$W(0) = 2, \quad W(1) = 10, \quad W(2) = 3, \quad W(3) = 7$$

and that we don't care what the other values of $W(x)$ are. Then we can define a satisfactory function W by (wastefully) defining enough other functions:

$$\begin{cases} Y(0) = 3 \\ Y(x') = 7 \end{cases} \qquad \begin{cases} X(0) = 10 \\ X(x') = Y(x) \end{cases} \qquad \begin{cases} W(0) = 2 \\ W(x') = X(x). \end{cases}$$

Then we find that $W(2) = X(1) = Y(0) = 3$ and that $W(x)$ happens to have value 7 for all values of x greater than 3. Clearly, Y, X, and W are all primitive-recursive functions. We leave to the reader to show that functions of two variables like $Q(x,y)$ and $D(x,y)$ can be defined similarly.

†We give here a very general form. ϕ is the function being defined. ψ is the base function, which appeared only as a constant or as a single variable in the examples of 10.1.3. The function χ describes some computation that depends on the induction variable y, already-defined values of the new function ϕ, and on whatever parameters $x_2, \ldots, x_n$ may be desired. For instance, in the definition of $F^\times(y, x)$ above $\psi(x) = 0$ and $\chi(a, y, x) = F^+(a, x) = a + x$.

We can conclude from all this that *the Turing transformation functions are primitive-recursive functions.* Warning: we have not yet shown, and it is false, that *any* computation is primitive recursive; we will get to this shortly.

PROBLEM. Sketch the construction of *two-variable* fixed functions like Q, R, D from primitive-recursive definitions.

PROBLEM. Give primitive-recursive definitions for $f(y,x) = x^y, f(x) = x!$

10.2.1 Turing machine computations and primitive recursion

Although we have been able to represent, by a set of four primitive-recursive functions, the effect of a *single step* of a Turing machine, we have said nothing about the ultimate effect of continued operation of the machine. We can handle this by introducing a new integer variable t to represent the number of steps the machine has taken—the number of moments it has been in operation. Next we introduce four new (five-variable) functions T_q, T_s, T_m, T_n. The function $T_q(t,q,s,m,n)$ is to give the value of q that will result if *the machine T is started in condition (q,s,m,n) and is run for t steps.*

To calculate this, we have to iterate t times the transformations q^*, s^*, m^*, n^*. Thus,

$$T_q(0,q,s,m,n) = q$$
$$T_q(1,q,s,m,n) = q^*(q,s,m,n)$$
$$T_q(2,q,s,m,n) = q^*(q^*(q,s,m,n), s^*(q,s,m,n), m^*(q,s,m,n), n^*(q,s,m,n))$$

etc.

But, in fact, we can use the primitive-recursion scheme directly to define $T_q(t,q,s,m,n)$ for all values of t, and in general

$$\begin{cases} T_q(0,q,s,m,n) = q \\ T_q(t',q,s,m,n) = T_q(t, q^*(q,s,m,n), s^*(q,s,m,n), \\ \qquad\qquad m^*(q,s,m,n), n^*(q,s,m,n)) \end{cases}$$

We can define similarly T_s, T_m, and T_n. *It follows that the iterated Turing functions T_q, etc. are also primitive-recursive functions.* These defining equations have a simple intuitive meaning. They state that the effect of $t+1$ operations, in initial situation (q,s,m,n) is the same as that of applying t such operations to (q^*, s^*, m^*, n^*)—that is of applying first one operation and then t more operations! What could be simpler?

10.3 THE PROBLEM OF RECURSION WITH SEVERAL VARIABLES

The proof in the previous paragraph is defective, because the recursive definition given for T_q was not, in fact, an example of primitive recursion (PR) as defined in section 10.2! The trouble is that the parameters q, s, m, and n do not stay fixed during the calculation. Nevertheless, it *is* possible to define T_q using only legitimate forms of PR, and it is the goal of this section to show how this can be done.

10.3.1 The information-packing functions H, T, C

Let us consider a simpler form of the problem: Suppose we have a definition

$$T(n',a,b) = T(n,\sigma(a,b),\tau(a,b))$$

where σ and τ are already-defined PR functions. Then I assert that T is also PR, but I cannot find any *direct* way to force it into the form of the PR schema. The trouble is that the PR schema seems to provide no way for two different computations to interact; in this case the dependency of $T(n',a,b)$ upon both $\sigma(a,b)$ and $\tau(a,b)$.

Now this appearance is really false, but it seems that to get around it requires some sort of machinery for dealing simultaneously with several sources of information. Suppose, for example, that we were able to find three PR functions, $\mathsf{C}(y,z)$, $\mathsf{H}(x)$, $\mathsf{T}(x)$, with the properties that, for all y and z,

$$\mathsf{H}(\mathsf{C}(y,z)) = y$$

and

$$\mathsf{T}(\mathsf{C}(y,z)) = z$$

We will indeed soon find just such a trio of functions:

$$\mathsf{H} = \text{HEAD}$$
$$\mathsf{T} = \text{TAIL}$$
$$\mathsf{C} = \text{CONSTRUCT}$$

Then we could redefine T as follows:

$$T(n,a,b) = T^*(n,\mathsf{C}(a,b))$$

and define T^* recursively as

$$\begin{cases} T^*(0,x) = T(0,\mathsf{H}(x),\mathsf{T}(x)) \\ T^*(n',x) = T^*(n,\mathsf{C}(\sigma(\mathsf{H}(x),\mathsf{T}(x)),\tau(\mathsf{H}(x),\mathsf{T}(x)))) \end{cases}$$

This may not look any better on the surface, but the next section will show that it is indeed PR. We first verify that T^* has the right properties:

$$\begin{aligned} T(n', a, b) &= T^*(n', \mathsf{C}(a, b)) \\ &= T^*(n, \mathsf{C}(\sigma(\mathsf{H}(\mathsf{C}(a, b)), \mathsf{T}(\mathsf{C}(a, b))), \tau(\mathsf{H}(\mathsf{C}(a, b)), \mathsf{T}(\mathsf{C}(a, b))))) \\ &= T^*(n, \mathsf{C}(\sigma(a, b), \tau(a, b))) \\ &= T(n, \sigma(a, b), \tau(a, b)) \end{aligned}$$

Now we want to show that T, as defined above, is a PR function. If we think of

$$\mathsf{C}(\sigma(\mathsf{H}(x), \mathsf{T}(x)), \tau(\mathsf{H}(x), \mathsf{T}(x)))$$

as a single primitive-recursive function of x, the following theorem will apply.

10.3.2 The σ^n theorem.

Consider

$$\begin{cases} \phi(0, x) = \psi(x) \\ \phi(n', x) = \phi(n, \sigma(x)) \end{cases}$$

where ψ and σ are PR. It is not obvious that ϕ is PR, from this definition, because it does not fit as an instance of the PR schema. But, let us look at its values:

$$\begin{aligned} \phi(0, x) &= \psi(x), \\ \phi(1, x) &= \phi(0, \sigma(x)) = \psi(\sigma(x)), \\ \phi(2, x) &= \phi(1, \sigma(x)) = \psi(\sigma(\sigma(x))), \\ \phi(3, x) &= \psi(\sigma(\sigma(\sigma(x)))), \\ &\cdots \end{aligned}$$

and clearly in general

$$\phi(n, x) = \psi(\sigma^n(x))$$

where $\sigma^n(x)$ means applying σ to x, n times. Now we can define $\sigma^n(x)$ recursively as

$$\begin{cases} f(0, x) = x \\ f(n', x) = \sigma(f(n, x)) \end{cases}$$

and clearly f is PR and

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

Hence

$$\phi(n, x) = \psi(f(n, x))$$

So ϕ is PR after all—i.e., because it is identical with a PR function defined another way!

PROBLEM 10.3-1. Prove by induction that $\phi(n, x) = \psi(f(n, x))$. Hint: Use induction on n, assuming each hypothesis true for all x. First prove that $f(n, \sigma(x)) = \sigma(f(n, x))$.

10.3.3 Defining the information-packing functions H, T, and C

In 10.3.1 we showed that if we could find functions $\mathsf{C}(y, z)$, $\mathsf{H}(x)$, and $\mathsf{T}(x)$ for which

$$\mathsf{H}(\mathsf{C}(y, z)) = y$$

and

$$\mathsf{T}(\mathsf{C}(y, z)) = z$$

then we could handle a function like $T(t, q, s, m, n)$ except with only two variables $T(t, a, b)$ in addition to t. Before we construct H, T, and C, let us note that, by using them more fully, we really can handle the full T problem! For we can write

$$T(t, q, s, m, n) = T^*(t, \mathsf{C}(\mathsf{C}(q, s), \mathsf{C}(m, n)))$$

horrible as this may seem, and then all the information in (q, s, m, n) is packed into one number

$$x = \mathsf{C}(\mathsf{C}(q, s), \mathsf{C}(m, n))$$

Or, alternatively,

$$x = \mathsf{C}(q, \mathsf{C}(s, \mathsf{C}(m, n)))$$

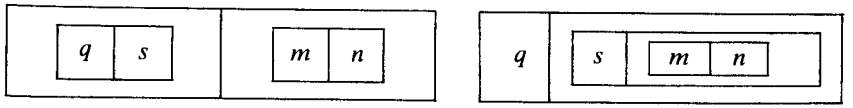
In fact, all the information can be extracted again simply by using

$$\begin{aligned} q &= \mathsf{H}(\mathsf{H}(x)) \\ s &= \mathsf{T}(\mathsf{H}(x)) \\ m &= \mathsf{H}(\mathsf{T}(x)) \\ n &= \mathsf{T}(\mathsf{T}(x)) \end{aligned}$$

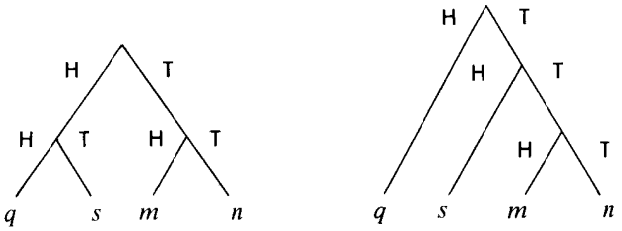
or, alternatively,

$$\begin{aligned} q &= \mathsf{H}(x) \\ s &= \mathsf{H}(\mathsf{T}(x)) \\ m &= \mathsf{H}(\mathsf{T}(\mathsf{T}(x))) \\ n &= \mathsf{T}(\mathsf{T}(\mathsf{T}(x))) \end{aligned}$$

which correspond to two different information structures:



or tree structures:



So we can define the real T by the schema

$$T^*(t', x) = T^*(t, \sigma(x))$$

where σ is a single monstrous but still PR function; then the σ^n theorem assures us that T itself, that is, $T(t, q, s, m, n)$ is primitive-recursive.

Now let us construct H, T, and C. We will adopt a simple encoding scheme by defining

$$\mathsf{C}(y, z) = 2^y + 2^{y+z+1}$$

If we think of $\mathsf{C}(y, z)$ as a binary number then it will have the simple form

$$1\ \underbrace{0\ 0\ldots0\ 0}_{z}\ 1\ \underbrace{0\ 0\ldots00}_{y}$$

where y and z are now represented by strings of zeros. Now it is easy to see that this $\mathsf{C}(y, z)$ is PR:

$$\mathsf{C}(y, z) = \exp(y) + \exp(y + z + 1)$$

where

$$\begin{cases} \exp(0) = 1 \\ \exp(n') = 2 \cdot \exp(n) \end{cases}$$

So our real problem is to find PR definitions for the H and T for which

$$\mathsf{H}(2^y + 2^{y+z+1}) = y$$
$$\mathsf{T}(2^y + 2^{y+z+1}) = z$$

Now in fact $\mathsf{H}(x)$ is the number of times x is evenly divisible by two; and again with the aid of the σ^n theorem we will be able to prove it is PR.

DIVIDING BY 2^n

We have already defined (in section 10.1) the function $H(x)$

$$H(x) = \frac{x}{2} \qquad \text{if } x \text{ is even}$$

$$H(x) = \frac{x - 1}{2} \qquad \text{if } x \text{ is odd}$$

Now we would like to make a more drastic distinction between even and odd numbers; so we define

$$J(x) = H(x) \cdot N(P(x))$$

which has the property:

$$J(x) = \frac{x}{2} \qquad \text{if } x \text{ is even}$$

$$J(x) = 0 \qquad \text{if } x \text{ is odd}$$

We also define $E(x) = N(N(x))$ which will be 0 if x is 0, and 1 otherwise. Now consider the sequence

$$E(x), \quad E(J(x)), \quad E(J(J(x))), \quad E(J(J(J(x)))), \quad \text{etc.}$$

It is easy to see that if x is divisible by 2, n times, then $E(J^n(x))$ will be 1, otherwise it will be zero. So, using the σ^n theorem if we define $D(n, x)$ to be

$$\begin{cases} D(0, x) = E(x) \\ D(n', x) = D(n, J(x)) \end{cases}$$

so that

$$D(n, x) = E(J^n(x))$$

we have in D the PR function which tells whether or not 2^n divides x. Next we define

$$\begin{cases} G(0, x) = E(x) \\ G(n', x) = D(n', x) + G(n, x) \end{cases}$$

so that

$$G(n, x) = E(x) + E(J(x)) + E(J(J(x))) + \ldots + E(J^n(x))$$

is the number of powers of 2, up to 2^n, that divide x. Finally, if we define

$$\mathsf{H}(x) = G(x, x)$$

we obtain the function we wanted—because 2 *can't* divide x more than x times![†] We have now only to get $\mathsf{T}(x)$. We will leave it to the reader to confirm that if

$$x = 2^y \cdot 2^{y+z+1}$$

[†]This last observation is really vital. We really defined $G(n, x)$ recursively to be

$$\sum_{i=1}^{n} D(i, x)$$

and thus showed, incidentally, how this "bounded summation" can be done primitive-recursively, for any function.

then

$$\mathsf{H}(x) = y$$

(as we have shown), and

$$\mathsf{T}(x) = \mathsf{H}(H^{\mathsf{H}(x)+1}(x)) = z$$

i.e., applying H to x, $\mathsf{H}(x) + 1$ times, and then applying H, and that this latter function is PR.

PROBLEM 10.3-2. Define H and T if $\mathsf{C}(y, z) = 2^y(2z + 1)$. This is actually simpler. We chose the form in section 10.3.1 so that the reader could see easy ways to generalize, e.g., to four variables directly: here we can use

$$f(a, b, c, d) = 2^a(1 + 2^{b+1}(1 + 2^{c+1}(1 + 2^{d+1})))$$

Define the appropriate information-recovery functions for this new form of C.

PROBLEM 10.3-3. In some sense, if H, T, and C exist, it must be the case that, on the whole,

$$\mathsf{C}(y, z) \geq y \cdot z$$

Discuss this. In the Solutions, for this problem we show that there is a PR function C, for any $\epsilon > 0$, for which

$$\mathsf{C}(y, z) < yz^{(1+\epsilon)}$$

so that in this sense one can get very close. Gödel, who originated these methods, used functions like

$$f(a, b, c, d) = 2^a 3^b 5^c 7^d.$$

See chapter 14 for further discussion of some methods of this sort.

PROBLEM 10.3-4. The encoding-decoding method in 10.3.3 uses large numbers and uses facts about repeated factorizations or divisions. It is possible to avoid most of that machinery, and we hasten to give an alternative. Consider using for $\mathsf{C}(y, z)$ the quadratic polynomial

$$\mathsf{C}(y, z) = \tfrac{1}{2}((y + z)^2 + (y + z) + 2x)$$

which is much smaller than the exponential encodings given above. Prove (1) that $\mathsf{C}(y, z)$ has different values for every pair of positive integers; (2) that there exist primitive-recursive inverse functions $\mathsf{H}(x)$ and $\mathsf{T}(x)$ for which

$$\mathsf{H}(f(y, z)) = y, \qquad \mathsf{T}(f(y, z)) = z$$

and that f is as efficient as possible in that it takes on all integer values.

10.4 THE (GENERAL) RECURSIVE FUNCTIONS

The primitive-recursive definition schemes have a critically important property. *Any primitive-recursive definition defines uniquely the values of a function for the entire range of its arguments*; there can be no ambiguity. (For a proof of this, if the reader cannot supply it himself, see Davis [1958, Theorem 3.3, p. 48].) In particular, for any Turing machine T, the iterated Turing functions T_q, T_s, T_m, and T_n are defined for all values of their variables, including all values of t. But curiously, this very virtue makes it difficult (in fact, impossible) to represent by primitive-recursive functions *all* the functions defined by the *computations* of Turing machines.

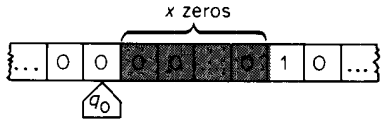
The trouble is that we have no representation, in terms of primitive recursion, of what it could mean for a Turing machine to halt. And this is much more serious than may seem at first glance.

To examine this more closely, let us make the definitions a little more precise. A Turing machine T will be said to compute a value $T(x)$ if, when started in some standard condition with some standard representation of an integer x on the tape, it eventually *halts* with (a representation of) the integer $T(x)$ on its tape. To make matters concrete, we will choose for our standard representation of the input x, the machine configuration†

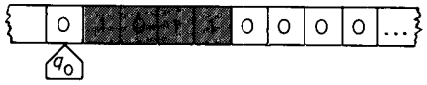
$$(q, s, m, n) = (0, 0, 0, 2^x)$$

For a standard representation of the output, we will arrange that the Turing machine halts after entering state q_1 and agree that any final con-

†That is, the machine is given the number x in the form:



The reason for this choice is a little technical: we want it to be simple, but the Turing machine has to be provided with some kind of punctuation so that it can recognize where the input data *ends*. We can't use the simpler representation (0, 0, 0, x) because then the machine will never be sure that it has yet seen the most significant digit of x and would have to keep moving forever to the right. The representation (0, 0, 0, 2^x), in effect, uses a version of unary notation instead of binary—the machine moves right (counting the 0's) until it encounters the 1—then it is certain that it has seen the whole number x. Thus the number 13 can't be represented by



in binary because the machine could never be sure there are no more 1's to the right. The reason we don't use unary notation directly is simply that we want to remain consistent with the definitions of q, s, m, and n in 10.1.1.

figuration of the form

$$(q = 1, s = \text{anything}, m = \text{anything}, n = 2^y(1 + 2 \cdot \text{anything}))$$

represents the function value $T(x) = y$.† If *such a configuration never occurs*, we say that the machine does not compute any value, or better, that $T(x)$ *is undefined*.

Now we can easily arrange for halting in an actual machine, or in an imaginary Turing machine, but we simply *cannot* do this within the framework of primitive-recursive functions. (See, for example, how the function $W(x)$ in 10.2 got defined for *all* values, whether we wanted it to or not.) What we need, precisely, for the representation of $T(x)$, is to be able to *find, given x, the value of*

$$T_n(t, 0, 0, 0, 2^x)$$

for the smallest value of t for which

$$T_q(t, 0, 0, 0, 2^x) = 1$$

For this means to run the machine until it enters state q_1—our halting state—and then read what is on its tape. It is customary to represent "the smallest value of t for which ..." by 'μ_t' so that

$$\mu_t\,[P(t) = K] \qquad \boxed{\text{MINIMIZATION OPERATOR}}$$

means "the least t for which $P(t) = K$." Using this notation, we can define the function $T(x)$ computed by Turing machine T by the expression

$$T(x) = T_n(\mu_t[T_q(t, 0, 0, 0, 2^x) = 1], 0, 0, 0, 2^x)$$

That is, first find out how many steps it will take the Turing machine to halt (i.e. to enter q_1), and then calculate the value of n after that many iterations of the Turing transformation.

DEFINITION OF GENERAL-RECURSIVE FUNCTION

> If a function can be defined by the apparatus of primitive recursion, together with use of the minimization operator μ, then it is a *general-recursive function* (or more briefly a *recursive function*).

We have just shown that the functions computable by Turing machines are general-recursive.‡ We will prove the converse in the following chapters.

†That is, y is the number of zeros before the first one to the right of the reading head.

‡One has to verify that the function $E(X) = 2^X$ is primitive-recursive:

$$\begin{cases} E(0) = 1 \\ E(X') = F^+(E(X), E(X)) \end{cases}$$

Note that the minimization operator was used only *once*, at the very end of our description of the Turing-machine computation! This is a general phenomenon; any general-recursive function can be obtained by a single application of μ to some primitive-recursive function. It will become clear why this is so as we proceed.

Are the general-recursive functions different from the primitive-recursive functions? They are, because it is impossible to define the μ operator in terms of zero, successor, composition, and primitive recursion. This will be shown in the next section.

PROBLEM 10.4-1. Prove that if $\phi(x)$ is primitive recursive, then

$$f(n) = \sum_{j=0}^{n} \phi(j) = \phi(0) + \phi(1) + \phi(2) + \ldots + \phi(n)$$

is primitive recursive. Prove also that

$$g(n) = \prod_{j=0}^{n} \phi(j) = \phi(0) \cdot \phi(1) \cdot \phi(2) \cdot \ldots \cdot \phi(n)$$

is primitive recursive.

PROBLEM 10.4-2. The "bounded minimization operator" $\mu_{(t<x)}$ is defined so that

$$\phi(x) = \mu_{(t<x)}\,[\psi(t) = X(t)]$$

is "the smallest value of t *less than* x for which $\psi(t) = X(t)$, unless there is no such t, in which case $\phi(x) = 0$." Show that $\phi(x)$ *is* primitive-recursive if $\psi(t)$ and $X(t)$ are. Solution in Kleene [1952, p. 228].

PROBLEM 10.4-3. Show that $LP(X)$ = "the largest prime factor of X" is general-recursive. Is it primitive-recursive?

10.5 TOTAL-RECURSIVE FUNCTIONS AND PARTIAL-RECURSIVE FUNCTIONS: TERMINOLOGY AND THEOREMS

In this section we explore some of the properties of the general-recursive functions. Most of the results are on the negative side—showing that there is no recursive function with such and such a property, or that some problem concerning a class of functions is effectively unsolvable. Some results are more positive, showing that while each effectively computable class of functions may have certain limitations, there are effective methods of extending such classes. In any case, one obtains a beautiful, relatively self-contained, new structure of mathematics—a theory of the relations between different classes of computable functions. We have space only to sketch parts of what is known, but fortunately even the elementary results are of great interest.

We must begin by pointing out that the definition, in 10.4, of "general-recursive function" is much more tricky and subtle than it appears. We said that a function is general-recursive if it is *defined* by a certain kind of set of equations. The trouble is that one cannot usually be sure, from looking at such a set of equations, that they "define" any function at all. When we speak of a "function" we mean, usually "a rule that yields, with each value (or n-tuple of values, if we have a function of n variables) a resulting value." But in general a system of equations doesn't always yield defined values. For example, consider the system

$$\begin{cases} E(0) &= 0 \\ E(x') &= E(x) \\ F(x) &= \mu_y[E(y) = x] \end{cases}$$

which satisfies the formal aspect of our definition. Now for $x = 0$ the value of $F(x)$ is defined and is 0. But for $x = 1$, our definition leads nowhere; the expression $\mu_y[E(y) = 1]$ means: First see if $E(0) = 1$; if this isn't so, see if $E(1) = 1$; if this isn't so, see if $E(2) = 1$; if this isn't so, etc., etc. The computation never terminates (because $E(x)$ is always zero) and no value is ever assigned to $F(1)$.

This is a characteristic phenomenon, and because of it, we must always hesitate to assume that a system of equations really defines a general-recursive function. We normally require auxiliary evidence for this, e.g., in the form of an inductive proof that, for each argument value, the computation terminates with a unique value.

Because of this we will talk usually of the *partial-recursive function* described by a set of equations involving zero, successor, primitive recursion, and minimization. When we talk about a partial-recursive function (or *partial function*, for short) $F(x)$, it is understood that there may be no value defined for some (or even any!) values of x. If $F(x)$ happens to be defined for all values of x, then we call it a *total-recursive function* or, for short, a *total function*. (In the literature "total-recursive" is a synonym of "general-recursive.") Of course any total-recursive function is also considered still to be a partial-recursive function as well.

One might suggest that matters would be simplified, if we would just confine our attention to the total-recursive functions. But we will see, shortly, that this suggestion is highly impractical!

What does the term "recursive" mean here? It refers to the *recurrence* of the name of the thing-being-defined inside its definition. Thus in

$$\begin{cases} F^+(0, x) &= x \\ F^+(y', x) &= S(F^+(y, x)) \end{cases}$$

we find the name of F^+, the thing-being-defined, inside an expression on the right-hand side of an equation, so this is a recursive definition. It

happens to satisfy the formal condition for being a primitive-recursive definition, as well.

PROBLEM. Primitive-recursive definitions *always* define total-recursive functions, so there is no problem of undefined function-values. Why is this?

The following system of equations defines a certain function $A(x)$ recursively:

$$\begin{cases} A(0, y) &= y' \\ A(x', 0) &= A(x, 1) \\ A(x', y') &= A(x, a(x', y)) \\ A(x) &= A(x, x) \end{cases}$$

These equations do not satisfy our formal conditions for either primitive- or general-recursive functions but they do in fact give a unique value for each argument x of $A(x)$. We could, in fact, have given a much weaker definition of "general-recursive," namely: "*any function defined by any recursive system of equations.*" It was proved by Kleene [1936] that our present definition comprises the same class of functions!

PROBLEM 10.5-1. Show that the Ackermann equations above really define a total-recursive function $A(x)$. Can you calculate $A(2)$? $A(3)$? $A(4)$? Can you even estimate the size of $A(10)$?

REMARK

These equations represent a sort of simultaneous induction on two variables at once. Functions defined in this way are known as "double-recursive" and it is known that some of them (including this one) are *not* primitive-recursive. In fact, it is known (see Péter [1951] or Robinson [1948]) that for each n there are n-recursive functions that are not $(n - 1)$-recursive. On the other hand, it is also known that the n-recursive functions are total. Indeed, in section 10.3.1 our proof of the σ-theorem shows how one might start to prove that the double-recursive functions are total.

10.6 EFFECTIVE ENUMERATIONS OF THE PARTIAL-RECURSIVE FUNCTIONS

For several reasons, we want to be able to think about all the partial-recursive functions as arranged in a sequence or list

$$f_1(x), f_2(x), \ldots, f_n(x), \ldots$$

and, indeed, we would like to have this list arranged in an orderly manner, so that we can talk about it in terms of effectively defined operations. The arrangment of a set of objects as a discrete, definite sequence is called

an *enumeration* of the set of objects. For our purposes, it is important that the enumeration of the partial-recursive functions be *effective*, that is, that we have an effective, computable way to find which is the nth function on the list, or at least to find its definition. Strangely enough, it will be enough for us to be sure that an enumeration exists—we won't ever need to know any details about its properties! Therefore, instead of constructing it in detail, it is sufficient to give a convincing argument that it is possible—that we could construct one if we had to!

Now the requirement that the enumeration be *effective* is equivalent to saying that given n and x, we are enabled at least to initiate the computation of $f_n(x)$, that is, the computation of the value of the nth partial-recursive function for the argument x. (The value of $f_n(x)$ may, of course, be undefined.) This in turn is to say that we need the definition of a partial-recursive function†

$$U(n, x) = f_n(x)$$

which is *universal* in the sense that it is defined exactly when $f_n(x)$ is and has the same value. If we have such a function $U(n, x)$ we will talk of n as the *index* number and of $f_n(x)$ as the nth function or the function with index n.

Why should we suppose that there even exists an effective enumeration of the partial-recursive functions? After all, they are a fantastically rich, bizarre, disorderly variety of objects; the *consequences of all possible recursive definitions*. We give two reasons, one direct and one indirect.

A direct construction involves observing that each partial-recursive function has a definition that consists of a finite set of equations each involving a finite number of symbols designating *zero*, *successor*, *primitive recursion*, and *minimization*, combined in an arrangement of a finite number of *commas*, *brackets*, and *parentheses*. The exact rules for such arrangements aren't important; what is important is that in any such situation one always can set up some sort of infinite dictionary-ordering for the composite objects, just as was done in Problem 6.3-2 for the well-formed parenthesis sequences. In the present situation, even though things seem more complicated, there is one way in which they are actually simpler. That is: since most partial-recursive functions (or rather, definitions) don't define genuine terminating computations at all, we need not be too concerned, in enumerating the definitions, that all the symbol strings generated be well-formed. In fact, we can consider any procedure that eventually produces any and all strings of symbols required for the

†This sentence makes sense only if we identify the notion of (i) "effectiveness" with (ii) "definable in terms of general-recursion." This is a philosophical rather than a mathematical hypothesis: it will be reinforced soon when we show that (ii) is mathematically identical with (iii) "computable by a Turing machine."

definitions. There is only one technical problem—that of staying inside a fixed alphabet—and that is solved by using, say, a binary-number scheme (just as in section 7.2) for the unlimited numbers of variable- and function-names that may be required. Then (as in section 9.3.1 which the reader should now read, though the chapter as a whole is optional) a simple numerical-order scheme will be an effective enumeration, provided there is a mechanism to interpret the nth description correctly. See also section 12.3 for a similar construction.

It is not, however, obvious at once that such a mechanism exists. To construct it is to write down the definition of a general-recursive function $U(n, x)$ that correctly interprets the number n and computes the appropriate function of x. To construct U directly would involve essentially the same amount of effort, *and essentially the same ideas*, as we have invested in constructing the universal Turing machine. It will be equally satisfactory for our current purpose, and more profitable in the long run, simply to prove that the computations of partial-recursive functions are equivalent in all respects to the computations of Turing machines. Therefore we will assume the existence of the recursive function $U(n, x)$, in view of the later demonstration that this is equivalent to the existence of a universal Turing machine.

So we will talk freely of the nth partial-recursive function $f_n(x)$. Warning: Do not assume anything! Some functions $f_n(x)$ may not be defined, even for a single value of x. Some differently indexed functions $f_i(x)$ and $f_j(x)$ may be the same: $f_i(x) = f_j(x)$ for all x. Indeed, some functions may appear infinitely often in the list with different explicit definitions. The only thing we are sure of is that if a function *is* partial-recursive, then it has *at least* one index in the enumeration.

We can also effectively enumerate the *primitive-recursive* functions of one argument (which are all total functions) and refer to them as $p_1(x), p_2(x), \ldots, p_n(x), \ldots$. When we say that a set of functions is effectively enumerated, the important thing is that we have a method that, given only a function's index number, tells us what are its values.

PROBLEM 10.6-1. Describe informally but in more detail, how one might construct effective enumerations of (1) The primitive-recursive functions, $p_i(x)$. (2) The partial-recursive functions, $f_i(x)$.

PROBLEM 10.6-2. Sketch the steps in the definition of a universal primitive-recursive function $V(n, x) = p_n(x)$, and show that it is total recursive.

PROBLEM 10.6-3. Sketch the steps in the definition of a universal partial-recursive function $U(n, x) = f_n(x)$. That is, try to anticipate the next few sections!

THEOREM 1

Not all total-recursive functions are primitive-recursive.

Proof: Consider the function

$$F(x) = p_x(x) + 1 = V(x, x) + 1$$

It is clear (1) that this function is total-recursive because PR functions are always defined but (2) that it cannot be primitive-recursive, because it differs for at least one value of its argument, namely $F(n) \neq p_n(n)$, from each primitive function $p_n(x)$—Cantor's diagonal argument again! The function is general recursive because the "universal primitive-recursive function" $V(n, x)$ is; hence, so is

$$F(x) = S(V(x, x))$$

THEOREM 2

There is a total-recursive function that grows faster than any primitive-recursive function!

Proof: Consider the total function $M(x)$ defined by

$$\begin{cases} S(0, b) = 0 \\ S(a', b) = S(a, b) + V(a, b) \\ T(0, a) = 0 \\ T(b', a) = S(a, b) + T(b, a) \\ M(x) = T(x, x) \end{cases}$$

In fact,

$$M(x) = \sum_{i=1}^{x-1} \sum_{j=1}^{x-1} V(i,j)$$

PROBLEM 10.6-4. Verify that $M(x)$ has the required property that, for each n,

$$M(x) > P_n(x)$$

for all sufficiently large x. (In fact, this holds whenever $x > n$.) $M(x)$ itself therefore cannot be primitive-recursive! What is not primitive-recursive about the above definition?

THEOREM 3

There is no effective way to decide, for arbitrary n, whether $f_n(0)$ is defined.

Proof: If one could, this would solve the halting problem for blank-tape Turing machines. To see this, assume that $U(n, x)$ is defined in terms of a universal Turing machine, and work back from the end of 10.3, filling in few details. It follows (with some work) that *there is no effective procedure to tell which of the partial functions are total functions.*

REMARK

It seems a bother to worry about undefined values for functions. Why not do away with them by some ruthless technique like the following? Let $f_n(x)$ be a partial-recursive function. Define $g_n(x) = f_n(x)$ when $f_n(x)$ is defined, and let $g_n(x) = 0$ when $f_n(x)$ is not defined. Now let us confine our attention only to the g's. For the g's are now defined everywhere! The trouble is that not all the g's so constructed will be *computable*, i.e., general-recursive! In fact the trouble is profound, for:

THEOREM 4

While some of the partial functions are not total because of some trivial defect in their defining equations, some are incompletably partial, because they do not agree with any total-recursive function everywhere they are defined.

Proof: Let $F(x)$ be defined to have value $f_x(x) + 1$ wherever $f_x(x)$ is defined, and to be undefined elsewhere. Now we *cannot* then argue that $F(x)$ is a new function different from all the $f_i(x)$. $F(x)$ *could* be the same as some f_n for which $f_n(n)$ is undefined.† But now suppose that $F(x)$ could be "completed," by assigning values where it is undefined, to become some total-recursive function. Then the new total-recursive function would have some position f_n in the enumeration (section 10.4) and, since f_n is a total-recursive function, $f_n(n)$ is defined. Hence $F(n)$ must have been defined in the first place, and $F(n) = f_n(n)$, because when we "completed" F, we didn't change already-defined values. But then by definition

$$F(n) = f_n(n) + 1 = f_n(n)$$

which is impossible.

It follows from this that while the partial-recursive functions can be effectively enumerated, the total-recursive functions can't! Specifically:

THEOREM 5

There is an effective procedure that, given any effective enumeration of (some) total-recursive functions, produces a new total-recursive function that was not in the given enumeration.

Proof: Let $T_n(x)$ be the enumeration. Then

$$F(x) = T_x(x) + 1$$

†Note thus how Cantor's diagonal argument would fail! The *partial*-recursive functions *can* all be effectively enumerated. In fact $F(x)$ as defined here *is* one of the f_i's, and $f_i(i)$ *is* undefined for that function!

is a total-recursive function, for $T_x(x)$ is always defined, since all the T-functions are total, and is different from each function $T_n(x)$, at least when x has the value n.

It is worth emphasizing that Theorem 5 has a positive character. It says that so far as effective enumerations are concerned, the "diagonalization process," namely *computing with both the enumeration process and the processes being enumerated*, has a curiously creative character—one gets something new. Indeed one can repeat this over and over again, getting infinitely many new functions. Most "philosophers," quick to leap at interpreting unsolvability theorems as evidence that machines have limitations that men have not, have overlooked this curious, positive, aspect of the situation.

10.7 CONDITIONAL EXPRESSIONS; THE McCARTHY FORMALISM

The arithmetization of any *practical* computation procedure by the methods of section 10.1 would be a very awkward business because of the obscure way in which the "flow of control" of the procedure has to be built into the equations. Consider the device in 10.1.2 by which the direction function $D(q, s)$ was used there to control which transformation to apply. We used the expression

$$m^* = (2m + R(q, s))D(q, s) + H(m)\bar{D}(q, s)$$

and the "branching" of the process was based on the fact that we could in effect compute both possible results and then multiply one or the other by zero. This is both unesthetic and inefficient. This defect—of poor process-description ability—is equally apparent in the construction of Turing machine state diagrams. There, too, while theoretically adequate, the description of a computation procedure is very awkward. The whole bag of arithmetic tricks of this chapter, while interesting, are unsatisfactory, both practically and esthetically.

In the languages used for digital computer programming, we find much more adequate provisions for selection of different computation "branches." But practical computer languages do not lend themselves to formal mathematical treatment—they are not designed to make it easy to prove theorems about the procedures they describe. In a paper by McCarthy [1963], we find a formalism that enhances the practical aspect of the recursive-function concept, while preserving and improving its mathematical clarity.

McCarthy introduces "conditional expressions" of the form

$$f = (\textbf{if } p_1 \textbf{ then } e_1 \textbf{ else } e_2)$$

where the e_i are expressions and p_1 is a statement (or equation) that may be true or false.

This expression means

> See if p_1 is true; if so the value of f is given by e_1.
> If p_1 is false, the value of f is given by e_2.

This conditional expression directly replaces the artificial multiplication trick. It also does more; it has also the power of the minimization operator. In fact it can give us the recursive function $T(n)$ directly from the Turing transformation equations of 10.1:

$$T(q, s, m, n) = (\textbf{if } q = 1 \textbf{ then } n \textbf{ else } T(q^*, s^*, m^*, n^*))$$

Note that this eliminates entirely the parameter t which carries the information about the length of the computation, as well as eliminating the minimization operator. The McCarthy formalism is like the general recursive (Kleene) system, in being based on some basic functions, composition, and equality, but with the conditional expression alone replacing both the primitive-recursive scheme and the minimization operator.

It is different, however, in that it has explicit provisions for self-reference, so that instead of using tricky arithmetic methods it can describe computations, enumerations, universal processes, etc. more directly. (We will not, however, give details of how it uses "quotation" for this.)

We will not develop the whole of McCarthy's system here, but, to get the flavor of it, we will develop the Turing transformation in a simplified version of it. Assume that we have available the notions of

> Zero
> Successor
> Equality (of numbers)
> Composition
> Conditional Expression

We define the *predecessor* function by

$$\text{pred}(n) = \text{pred2}(n, 0)$$
$$\text{pred2}(n, m) = (\textbf{if } m' = n \textbf{ then } m \textbf{ else } \text{pred2}(n, m'))$$

We will denote pred (n) by n^-. Note that pred (n) is not defined for $n = 0$; this will not matter if the function is used properly.

Next we define the useful functions:†

† Note that we cannot regard (**if** p **then** a **else** b) as simply a *function* of three quantities p, a, and b. For this would make us evaluate them all beforehand. But, often, the computation of b will never terminate when p is true; in this case we must evaluate a without even looking at b.

$$P(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } (\textbf{if } x = 1 \textbf{ then } 1 \textbf{ else } P(x^{--})))$$
$$H(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } (\textbf{if } x = 1 \textbf{ then } 0 \textbf{ else } H(x^{--})'))$$
$$x + y = (\textbf{if } x = 0 \textbf{ then } y \textbf{ else } x^{-} + y')$$
$$xy = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } y + x^{-} y)$$

This is enough for us to define the function computed by a Turing machine T, exactly as in section 10.1.2:

$$T(n) = T^*(0, 0, 0, 2^n)$$
$$T^*(q, s, m, n) = (\textbf{if } q = 1 \textbf{ then } n \textbf{ else } (\textbf{if } D(q, s) = 0 \textbf{ then } T^*(Q(q, s), P(m), 2m + R(q, s), H(n)) \textbf{ else } T^*(Q(q, s), P(n), H(m), 2n + R(q, s))))$$

which is both reasonably compact and reasonably expressive. Of course we have to define the state-transition functions (as in 10.2), but for this it is easy to use nested conditionals like

$$R(x, y) = (\textbf{if } x = 0 \textbf{ then } (\textbf{if } y = 0 \textbf{ then } 3 \textbf{ else } (\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } \ldots)) \textbf{ else } (\textbf{if } x = 1 \textbf{ then } (\textbf{if } y = 0 \textbf{ then } 5 \textbf{ else } (\textbf{if } y = 1 \textbf{ then } 1 \textbf{ else } \ldots)) \text{ etc.}))$$

and simply specify the function point by point.

The above formulation for T still involves the binary arithmetization of the Turing machine and its tape. We can also show *directly* that general recursion is within the scope of the present formalism by observing that, given any already defined function $f(x)$, we can obtain the result of applying the minimization operator $\mu_x[f(x) = N]$ to it by evaluating $\mu(0, N)$, where

$$\mu(x, N) \equiv (\textbf{if } f(x) = N \textbf{ then } x \textbf{ else } \mu(x', N))$$

while primitive recursion is obtained by definitions like

$$\phi(y, x) = (\textbf{if } y = 0 \textbf{ then } \psi(x) \textbf{ else } \chi(\phi(y^{-}, x), y^{-}, x))$$

EXAMPLE

We can define m/n = the largest x for which $m \geq nx$ as:

$$m/n = \text{divide}(m, n, 0, 0)$$
$$\text{divide}(m, n, x, y) = (\textbf{if } y = n \textbf{ then } \text{divide}(m, n, x', 0) \textbf{ else } (\textbf{if } m = nx + y \textbf{ then } x \textbf{ else } \text{divide}(m, n, x, y')))$$

PROBLEM 10.7-1. Define the functions:

(1) Prime(n) = (**if** n is prime **then** 1 **else** 0).

(2) gcd(m, n) = the greatest common divisor of m and n.

(3) $\phi(n)$ = the number of integers less than n which have no divisors in common with n.

10.8 DESCRIPTION OF COMPUTATIONS USING LIST-STRUCTURES†

In this section we introduce a formalism for representing functions of "lists" of integers. In the next section we use a more expressive system that can describe "branching" lists, or lists of lists, etc.

DEFINITION

An I list is a sequence of integers

$$\langle x_1, x_2, \ldots, x_n \rangle$$

The I list of *no* integers '$\langle\ \rangle$' is called NIL.

We define three functions of I lists; if A is an integer and B is the list $\langle x_1, \ldots, x_n \rangle$, then we define

$$\mathsf{C}(A, B) = \langle A, x_1, \ldots, x_n \rangle \quad \text{CONSTRUCT}$$
$$\mathsf{H}(B) = x_1 \quad \text{HEAD}$$
$$\mathsf{T}(B) = \langle x_2, \ldots, x_n \rangle \quad \text{TAIL}$$

If B is NIL then $\mathsf{H}(B)$ and $\mathsf{T}(B)$ are undefined. If B is a list of one integer $\langle x \rangle$ then $\mathsf{H}(B) = x$ and $\mathsf{T}(B) =$ NIL.

We can deduce the relations‡

$$\mathsf{H}(\mathsf{C}(A, B)) = A$$
$$\mathsf{T}(\mathsf{C}(A, B)) = B$$

so that the functions H and T can be used to disassemble an I list put together using C. Now suppose that we adjoin these functions and the concept of an I list to the notions already available (equality, composition, conditional expression). Then we can deal *directly* with the Turing machine tapes by thinking of the tape as two I lists, m and n. We then define the four functions, as "list-processing" operations:

$$q^*(q, s, m, n) = Q(q, s)$$
$$s^*(q, s, m, n) = (\textbf{if } D(q, s) = 1 \textbf{ then } \mathsf{H}(n) \textbf{ else } \mathsf{H}(m))$$
$$m^*(q, s, m, n) = (\textbf{if } D(q, s) = 1 \textbf{ then } \mathsf{C}(R(q, s), m) \textbf{ else } (\textbf{if } \mathsf{T}(m) = \text{NIL} \textbf{ then } \langle 0 \rangle \textbf{ else } \mathsf{T}(m)))$$
$$n^*(q, s, m, n) = (\textbf{if } D(q, s) = 0 \textbf{ then } \mathsf{C}(R(q, s), n) \textbf{ else } (\textbf{if } \mathsf{T}(n) = \text{NIL} \textbf{ then } \langle 0 \rangle \textbf{ else } \mathsf{T}(n)))$$

These functions express the Turing quintuples in a straightforward way. For example, $r^*(q, s, m, n)$ "reads" the first element on list n, if the machine moves right, otherwise the first element of list m. The last clause

†This section is optional.

‡Thus the system has as "primitive functions" right at the start, the $\mathsf{C}(y, z)$, $\mathsf{H}(x)$, and $\mathsf{T}(x)$ functions which we had to construct in section 10.3[1].

in the definitions of m^* and n^* is a device for making the finite lists m and n act as though they were infinite, by adding an extra zero whenever the machine reaches a point further out than it has ever reached before. For $\langle 0 \rangle$, which happens to be $\mathsf{C}(0, \text{NIL})$, represents adjoining a new blank square to a tape.

10.9 LISP†

For the sake of completeness we will now describe a little more of the full concept of list structure as developed by McCarthy. We want to be able to handle, not just lists of integers, but lists of lists, lists of lists of lists, etc. We can do this in an elegant manner, so that the basic functions $\mathsf{H}(x)$ and $\mathsf{T}(x)$ come out to be more symmetrical. To do this, we introduce a new basic notion—a binary combining operation "dot"—and make the notion of "list" a derived, subsidiary concept. We begin, then, with only the following basic notions:

Atom: The entity NIL is an atom. The integers 0, 1, 2, . . . are atoms.
Equality: We can tell if two atoms are the same.
Symbolic expressions: Any atom is a symbolic expression. If A and B are symbolic expressions, so is $\langle A \cdot B \rangle$.

Symbolic functions: If A and B are symbolic expressions then

$$\mathsf{C}(A, B) = \langle A \cdot B \rangle$$
$$\mathsf{H}(\langle A \cdot B \rangle) = A$$
$$\mathsf{T}(\langle A \cdot B \rangle) = B$$

Then we can deduce the identity

$$\mathsf{C}(\mathsf{H}(\langle A \cdot B \rangle), \mathsf{T}(\langle A \cdot B \rangle)) = \langle A \cdot B \rangle$$

Lists will not be fundamental objects but are defined as merely certain kinds of symbolic expressions; namely $\langle\ \rangle \equiv$ NIL is a list; if X is an atom then $\langle X \rangle \equiv \langle X \cdot \text{NIL} \rangle$ is a list; and more generally, if $X, Y, \ldots, Z$, etc. are atoms or *lists* then

$$\langle X, Y, \ldots, Z \rangle \equiv \langle X \cdot \langle Y \cdot \langle \ldots \langle Z \cdot \text{NIL} \rangle \ldots \rangle \rangle \rangle$$

†Optional. "LISP" (List Processor) denotes both the mathematical formalism, part of which is described here, and the practical computer programming system based on it, referenced in the notes.[1]

is a list. Thus one can make a list that is a list of lists: For example,

$$\langle \langle A, B \rangle, C, \langle D, E \rangle \rangle$$

is the list whose members are $\langle A, B \rangle$, C, and $\langle D, E \rangle$ and whose symbolic expression is

$$\langle \langle A \cdot \langle B \cdot \text{NIL} \rangle \rangle \cdot \langle C \cdot \langle \langle D \cdot \langle E \cdot \text{NIL} \rangle \rangle \cdot \text{NIL} \rangle \rangle \rangle$$

One can verify that the definitions in 10.5.1 are consistent with these definitions of H, T, C, and lists.

With this machinery we can represent the Turing transformation as follows: Let t = the list $\langle q, s, m, n \rangle$, where m and n are themselves lists representing the Turing machine tape halves. Observe that

$$q = \mathsf{H}(t) \qquad s = \mathsf{H}(\mathsf{T}(t)) \qquad m = \mathsf{H}(\mathsf{T}(\mathsf{T}(t))) \qquad n = \mathsf{H}(\mathsf{T}(\mathsf{T}(\mathsf{T}(t))))$$

which simply disassembles the list t into its four sublists. Define the function

$$\text{list}\,(a, b, c, d) = \mathsf{C}(a, \mathsf{C}(b, \mathsf{C}(c, d)))$$

which is used for reassembly of lists; since

$$\text{list}\,(A, B, C, D) = \langle A, B, C, D \rangle$$

And define

$$\text{tape}\,(n) = (\textbf{if } n = \text{NIL } \textbf{then } \mathsf{C}(0, \text{NIL}) \textbf{ else } n)$$

to make tape when necessary. Then if we define

$$T(\langle q, s, m, n \rangle) = (\text{if } D(q, s) = 1 \textbf{ then}$$
$$\text{list}\,(Q(q, s), \mathsf{H}(n), \mathsf{C}(R(q, s), m), \text{tape}\,(\mathsf{T}(n))) \textbf{ else}$$
$$\text{list}\,(Q(q, s), \mathsf{H}(m), \text{tape}\,(\mathsf{T}(m)), \mathsf{C}(R(q, s), n)))$$

and

$$T^*(t) = (\textbf{if } \mathsf{H}(t) = 1 \textbf{ then } \mathsf{H}(\mathsf{T}(\mathsf{T}(t))) \textbf{ else } T^*(T(t)))$$

then $T^*(\langle 0, 0, 0, 2^n \rangle)$ is the partial-recursive function computed by the Turing machine.

PROBLEM 10.9-1. Show that this system can be further reduced to one in which the only primitives are

Atoms: Only NIL
Equality: Only "**if** X = NIL **then** . . . **else** . . . "
Functions: $\mathsf{H}(x)$, $\mathsf{T}(x)$, $\mathsf{C}(x, y)$

and

Composition

That is, that one doesn't even need the notion of integer or successor.

NOTE

1. In the LISP programming system, the actual functions C, T, H, are called *cons* (*A*, *B*), *car* (*A*) and *cdr* (*A*) for historical reasons. For further details on this advanced computer list-processing language see McCarthy et al., *The LISP 1.5 Programmer's Manual* [1962] and *The Programming Language LISP* [1964]. See also McCarthy [1963] for some theory of the formalism. A review paper, Bobrow and Raphael [1962], gives a general discussion of languages of this kind. The use of list-structure primitives was first introduced, for practical use in complicated computer-programming situations, by Newell, Shaw, and Simon [1956]. The recognition that supplementing these primitives with composition produced an alternate formulation for recursive function theory was due to McCarthy, who also developed it into a practical "algebraic"-type computer language.

11 MODELS SIMILAR TO DIGITAL COMPUTERS

11.0 INTRODUCTION

We will now set up some models of effective computation which are more closely related to what happens in modern digital computers than the ones we have been considering. Computers have a number of attractive features. The process to be carried out by the machine is described by sequences of "instructions" arranged in meaningful blocks. Control of the process jumps from one block to another as a function of the results of computations. Partial results are stored away in a memory where they remain until required again. Each basic step—an elementary "instruction"—is itself a reasonably meaningful unit. This combination of features makes it possible to describe very complicated processes without having to manipulate absurdly long and hard-to-understand strings of symbols. As we have seen, neither the formalism of Turing machines nor that of general-recursive functions provides a satisfactory vehicle for handling the kind of procedures that one might really want to use for practical purposes. To be sure, neither does the system of the present chapter; adequate systems for this are found only in the computer programming languages in actual use. The systems developed here lie in a middle ground, where we can use the formalism for analysis and understanding of basic logical principles but still see fairly clearly how the system could be adapted for practical application.

In this chapter we develop an imaginary family of computer-like machines, and prove that they are logically equivalent to the other formulations of computability that we have examined. The new machines present a somewhat different version of what is involved in effective processes, and one that I feel gives a better intuitive picture of what such processes

can do. Of course, what seems natural and intuitive to one person may not seem so to someone else. Each of the systems has its advocates among the outstanding theorists, and each has served in the making of significant discoveries.

The first formulation of Turing-machine theory in terms of computer-like models appears in the paper of Wang [1957], which contains results that would have been much more difficult to express in the older formalisms. The development in this chapter is adapted for the arithmetic methods we have been using and incorporates ideas suggested by Wang and by John Cocke. One can find most of the basic results in Kleene [1952] but without their formulation as computer programs.

The reader unfamiliar with computers is warned that computers do not really work as shown here; in particular they don't have small numbers of "registers" each of infinite capacity. Instead they have large numbers of registers of small capacity!

11.1 PROGRAM MACHINES AND PROGRAMS

If we are to use machines as models for effective computability, we have somehow to lift the finiteness restriction. For Turing machines this was done by introducing the infinite memory tape. But there is a great deal of inconvenience about the Turing-machine computation process. One should not have to pass through everything in the record in the course of writing down or retrieving each piece of new or old data. There is something repellent about all the artifices of marking, copying, and encoding.

The modern digital computer has a different memory structure—a large number of separately-accessible, finite-capacity "registers" or memory units. Of course, any real computer memory is actually finite, though the number of such registers may be very large. Our first thought might be to make our model like this, except with an infinite set of such registers. The trouble is that this would bring us back to something along the lines of a tape-like succession of the registers; we would need to have an infinite set of *names* or symbols for the registers, and this could not be handled directly by the finite-state part of the machine.

We will suppose instead that our computer has only a *finite number of registers*, but that each of these can hold arbitrarily large numbers—that is, *each register is infinite in capacity*. There will also be a finite-state part of the machine—the central processor or *programming unit*. Since this part of the machine is finite, it cannot deal directly with arbitrarily large numbers. Therefore we will have to give it some way to deal indirectly with the full contents of the registers.

Our solution to this is simple and rather radical. The programming unit will be able to inspect any register but can discern only whether that register is empty or not—either that it contains 0 or that it contains a positive integer, but nothing else. It is quite surprising that even with this restriction, and with only a finite number of such registers, we can still realize any effective process—perform any effective computation. To show this, we will show that for any Turing machine there is an equivalent *program machine*, as we will call it.

Each *program machine* will have a certain finite number of registers, denoted by **a, b, c,** etc.† (See Fig. 11.1-11.) Each machine can perform certain *operations* which affect or detect things about the contents of registers. There will be only three or four distinct kinds of operations in any machine, and each operation will concern only one register at a time.

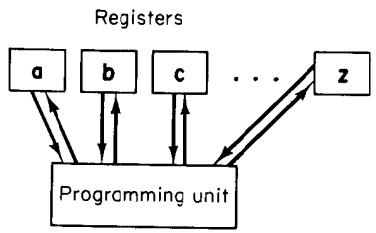


Fig. 11.1-1. Structure of a program machine.

The process carried out by the program machine is determined by its *program,* which is "built into" the programming unit of the machine.‡

Table 11.1-1

Symbol	Name	Example	Meaning
[0]	"Zero"	a^0	Set the contents of register **a** to zero. Go on to next instruction.
[′]	"Successor"	a'	Add 1 to the contents of register **a**. Go on to next instruction.
[−]	"Decrement or jump"	$a^-(n)$	If contents of **a** is not zero, decrease it by 1 and go on to next instruction. If contents of **a** is zero, jump to the nth instruction.
[H]	"Halt"	H	Stop the machine.

†For precision, we use the symbol **a** as the name of a register and the symbol a for its contents—the number stored in it. It is usually clear from the context which is meant, anyway.

‡These are *not*, then, "stored-program" computers. It is generally recognized that the greatest advances in modern computers came through the notion that programs could be kept in the same memory with "data," and that programs could operate on other programs, or on themselves, as though they were data. It is perhaps not so widely understood that one can obtain the same theoretical (though not practical) power in machines whose top level programs cannot be so modified—as in this chapter.

A *program* is a numbered sequence of *instructions*. An *instruction is a statement naming (1) an operation, (2) a register, and (3) the numbers of one or two other instructions.* We can explain these definitions best by putting down a set of operations and then exhibiting a few sample programs.

Our first machine is capable of the four operations shown in table 11.1-1. The letter a in these examples could be replaced by the name of any other register, and then the corresponding operation would affect the register so designated. We now describe a simple program machine. Suppose that a machine has three registers, **a**, **b**, and **w**, and has the program:

Instruction number	Instruction
1	b^0
2	a^- (7)
3	b'
4	b'
5	w^0
6	w^- (2)
7	H

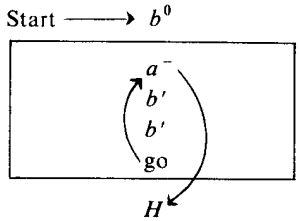
If the machine begins with instruction number 1 *and at that time there is a certain number in* **a**, *the machine will eventually halt with twice that number in* **b** (*and with* 0 *in* **a**). To see this, trace the operation, starting with 2 in **a** and anything in **b** and **w**:

Instruction number	Effect	Register contents a	b	w
1	set b to 0	[2]	0	-
2	a is not 0 so subtract 1 from it and go on	1	0	-
3	add 1 to b	1	1	-
4	add 1 to b. (b is now 2, a is 1.)	1	2	-
5	set w to 0.	1	2	0
6	w *is* 0 so go back to 2	1	2	0
2	subtract 1 from a.	0	2	0
3	add 1 to b.	0	3	0
4	add 1 to b.	0	4	0
5	set w to 0. (It was 0 already, but no harm is done.)	0	4	0
6	w is 0 so go to 2	0	4	0
2	Now a is 0 so go to 7	0	4	0
7	Halt.			

Each time around, a was reduced by 1 and b was increased by 2. When a is at last 0, the process halts, so that **b** now contains twice the original contents of **a**.

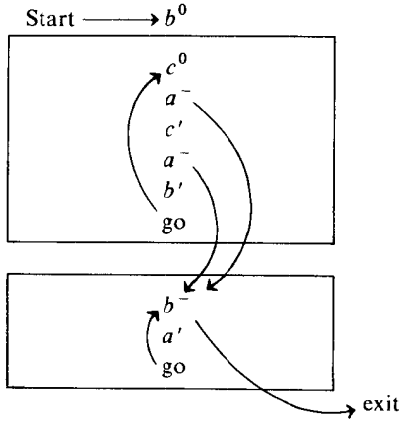
We are using the register **w** in a wasteful manner—it serves only as a device for getting the program to go back to an earlier instruction. We could have simply defined a special operation for this, but we wanted to keep the number of different operations to a minimum. From now on, we will use the instruction [go], e.g., 'go(n),' for this purpose—understanding that it could be replaced by a w^0, w^- (n) sequence.

Evidently the instruction numbers—1, 2, 3, . . .—have no intrinsic significance; they serve only to indicate the relative position of instructions in the sequence. We will omit them from now on, and indicate the "jump" or "transfer" of the process by arrows (just as we omitted state names in the diagrams of chapters 2–5). Our program is now represented by the diagram



The process usually goes from each instruction to the one below, but when the exceptional condition is met, the process follows the arrow. Here this happens only when a register is empty and a [−] operation is executed.

The following program will be very useful to us, for it computes the two functions $H(a)$ and $P(a)$ of chapter 10.



The first part of the program puts $H(a)$ in **b**, and puts $P(a)$ in **c**. Note how c is alternately set to 0 and 1, and how b is increased each time the program succeeds in subtracting 2 from a. The last three instructions simply transfer the contents of **b** into **a** so that we end up with $H(a)$ in register **a**. We indicate by boxes the basic repetitive "loops" of the program—the parts that really do the work.

11.2 PROGRAM FOR A TURING MACHINE

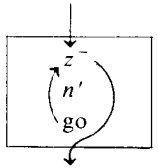
We now take an important step. We show that, given any Turing machine T, we can construct a program machine M_T which is equivalent to T. The machine M_T will have four registers named **s, m, n,** and **z.** The registers **s, m,** and **n** correspond to the numbers s, m, and n which, in chapter 10, describe the tape of the machine T. The machine M_T will transform these three quantities in precisely the same manner as they were transformed by the Turing transformation $T^*(q, s, m, n)$. To each state q_i of T there will correspond a block of program, and the event of T entering state q_i will be represented by M_T executing the corresponding program. Each state q_i is associated with two† quintuples $(q_i, 0, q_{i0}, s_{i0}, d_{i0})$ and $(q_i, 1, q_{i1}, s_{i1}, d_{i1})$.

Figure 11.2-1 is the complete program corresponding to state q_i of the Turing machine. We illustrate the case in which d_{i0} happens to be "left" and d_{i1} happens to be "right." The diagram has a comment explaining the function of each block of program. When the machine is started, at the top of the appropriate state program, one must have the proper initial value of s, m, and n in registers **s**, **m**, and **n**. The **z** register is used only for temporary storage during the calculation of the Turing transformation; we assume **w** and **z** contain zero at the start.

11.3 THE NOTIONS OF PROGRAMMING LANGUAGES AND COMPILERS

Observe that the program of Fig. 11.2-1 is composed essentially of copies of the programs of our three previous examples. We use them, in effect, as "subroutines"—meaningful units of program larger than the basic machine operations.

If we wish, we can be more formal about this use of subroutines. We could define formally the expressions that we used informally as comments in the figure, so that $\boxed{n \leftarrow z}$, for example, would be understood to mean precisely



and similarly for the other comments. (The details of this could become very complex.) Then we could write our program as follows, in a rather more expressive language:

†We assume T is a two-symbol machine, to be consistent with the formulation in chapter 10.

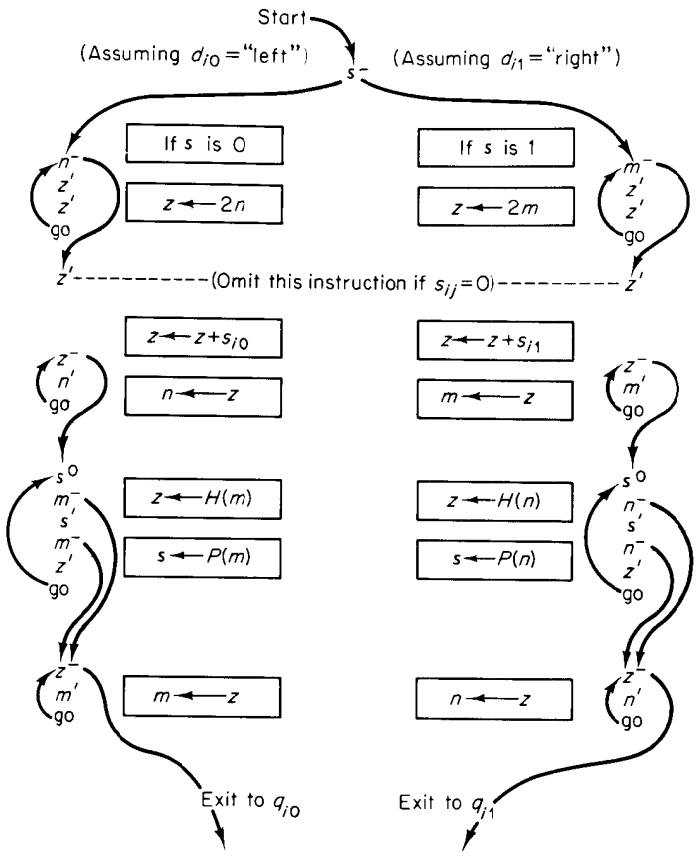


Fig. 11.2-1. Program for a Turing-machine state q_i.

Start: if $s = 0$ go to 1, else go to 2.

(1)	(2)
$z \leftarrow 2n$	$z \leftarrow 2m$
$z \leftarrow z + s_{i0}$	$z \leftarrow z + s_{i1}$
$n \leftarrow z$	$m \leftarrow z$
$z \leftarrow H(m)$	$z \leftarrow H(n)$
$s \leftarrow P(m)$	$s \leftarrow P(n)$
$m \leftarrow z$	$n \leftarrow z$
Go to q_{i0}	Go to q_{i1}

One can rather easily imagine another machine (which we won't develop) which could take this "program" and produce as output the program of Fig. 11.2-1. Such a machine would be called a "compiler" for the translation from the more sophisticated language into the simple operation language of the basic program machine. As we are not attempting a detailed exposition of what is involved in modern computer programming, there are many loose ends in this brief sketch, but it should serve to show something of what is involved. Today, most computers have basic operations not really very much more complicated than our [0], [′], [−], and the like, and most programs are written in other languages and then translated by compilers. The compilers are not actually separate machines but are usually programs that run inside the same computer that will finally execute the resulting basic machine-language program.

11.4 A SIMPLE UNIVERSAL BASE FOR A PROGRAM COMPUTER

If we choose T to be a universal Turing machine, the method of section 11.2 gives us a program computer equivalent to a universal Turing machine. Therefore, in some sense, any computation can be carried out by a computer which has only operations of the three types

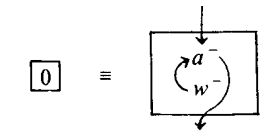
[0] set a register to 0

[′] add 1 to a register

[−] subtract 1 or transfer if already 0

If we begin with a certain register **w** already empty, we can even do without the operation [0]. For if the contents of **w** is zero, we can "clear"

any other register **a** by executing the loop program

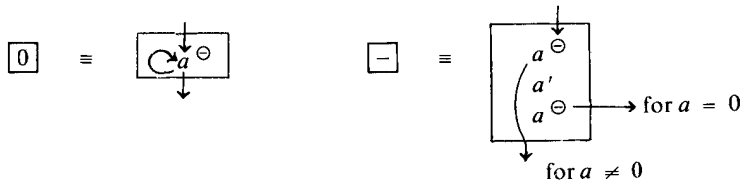


so that in this sense we need only operation types [′] and [−].

Instead of [−], we could use a slightly different operation type

[⊖]	"Decrement and jump"	$a^{\ominus}(n)$	If contents of **a** is not zero, subtract 1 and go to the nth instruction. If contents of **a** is zero, just go on to the next instruction.

To see that [′] and [⊖] form a sufficient base, observe that with them alone we can write programs (subroutines) which have the effect of [0] and [−].



How many registers does the universal program computer need? We can see from 11.1 that we need only the five registers **s**, **m**, **n**, **w**, and **z**. A good deal of thought about the way the machine works will show that the register **s** can be eliminated by encoding the information it contains into the program structure.

PROBLEM. Show how the program can be rewritten to eliminate the register **s**

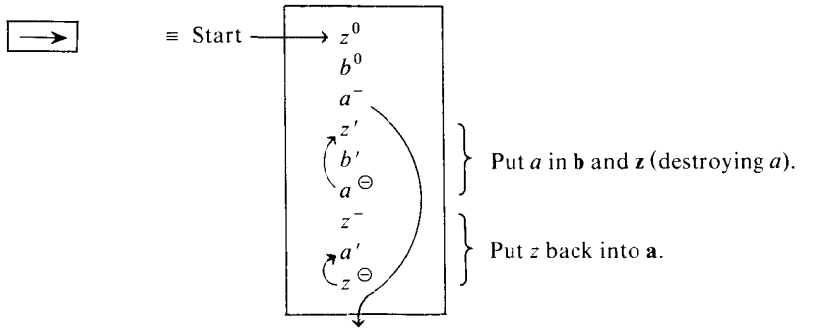
The register **w** is a nearly total waste, and could be eliminated at the cost of a simple "jump" operation, e.g., [go]—"Jump to the nth instruction." We can conclude that *there exists a universal program computer*

with a few simple operation types and with only three registers.[†] This is quite a remarkable fact, and we add it to our collection of the varieties of simple systems which are capable of unlimited effective computation. Later on (in chapter 14) we will show that actually only *two* registers are really necessary; this involves some non-trivial analysis!

11.5 THE EQUIVALENCE OF PROGRAM MACHINES WITH GENERAL-RECURSIVE FUNCTIONS

We will now show that any general-recursive function can be computed by a program machine using the operations defined in 11.4. To do this, it is convenient to define two new instruction types; and to make sure nothing new is involved here, we will define them as programs (subroutines) made up of the previously defined operations. They are:

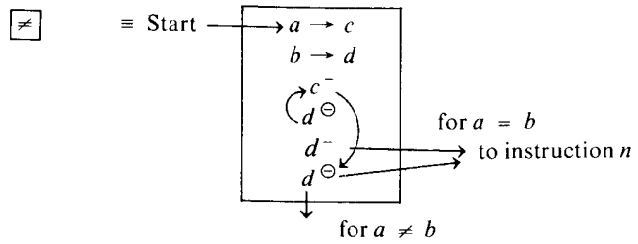
[→]	"Copy"	$a \to b$	Make the contents of **b** the same as the contents of **a** without changing a.



and:

[≠]	"Jump unless equal"	$a \neq b(n)$	If the contents of **a** and **b** differ, jump to nth instruction. If they agree, go on to next instruction.

[†]Most of today's widely available general-purpose scientific computers have at least three "index registers," and have also operations like [0], ['], [−], and [⊖] for changing them. Programmers of these machines will be amused to learn that these computers are thus universal even without using their main memory for data storage! Of course this is true only in principle, since the registers of real machines are far from infinite, and the main memory is needed for program storage.



We have defined [≠] in terms of [−], [⊖], and [→]. As an exercise one might work back to write a program for [≠] directly in terms of ['] and [−].

Next we show how to program the primitive-recursion scheme. Suppose that we know already how to calculate the values of two functions

$$\psi(x) \quad \text{and} \quad \chi(z, y, x)$$

That is, suppose we already have two blocks of program that, when started with values of z, y, and x in certain registers, will end up with the proper values of ψ and χ in other registers. We wish to write a program which will end up with the proper value, in a new register ϕ, of the function defined by the primitive recursion scheme:

$$\begin{cases} \phi(0, x) = \psi(x) \\ \phi(y', x) = \chi(\phi(y, x), y, x) \end{cases}$$

Now one ordinarily evaluates such a function by beginning with

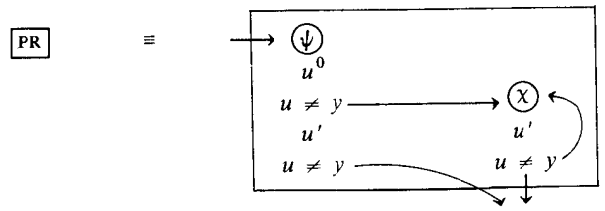
$$\phi(0, x) = \psi(x)$$

One then proceeds to calculate successively

$$\begin{aligned} \phi(1, x) &= \chi(\phi(0, x), 0, x) \\ \phi(2, x) &= \chi(\phi(1, x), 1, x) \\ \phi(3, x) &= \chi(\phi(2, x), 2, x) \\ &\vdots \quad\quad \vdots \end{aligned}$$

until one obtains the value of $\phi(y, x)$. To write a program to do this, suppose that we begin with the values of x and y in registers **x** and **y**. Let Ⓨ be a program which calculates the value of $\psi(x)$, by making use of register **x**, and leaves the value in register ϕ. Similarly, let Ⓧ be a program which uses registers ϕ, **u**, and **x**, and leaves the value of $\chi(\phi, u, x)$ in register ϕ. Note carefully the names of the registers involved. Then the

program



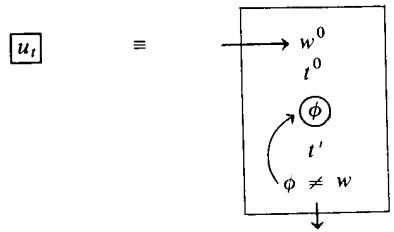
computes the value of $\phi(y, x)$, as the reader will verify, by proceeding through precisely the steps given above. We have written the program in a slightly awkward fashion to show that only operation types [0], [′], and [≠] need be involved here.

Observe that the computation proceeds forward, with u running from 0 to y directly, rather than backwards as suggested in 10.1.3. This makes it unnecessary to keep a record of the number of postponed operations.

The operation $a \neq b(n)$ gives us primitive recursion because it allows the machine to repeat the iteration until the yth step. In primitive recursion the number, y, of iterations is known at the start of the computation. But the same resources also give us full general recursion. To see this, consider the minimization operator—"the least t for which $\phi(t) = 0$,"

$$\mu_t[\phi(t) = 0]$$

and suppose that $\phi(t)$ is primitive-recursive or, more generally, that we already have a program which computes ϕ. Let Φ be a program which takes a number t from register **t** and leaves the value $\phi(t)$ in register ϕ. Then, "the least t for which $\phi(t) = 0$" is computed by the program



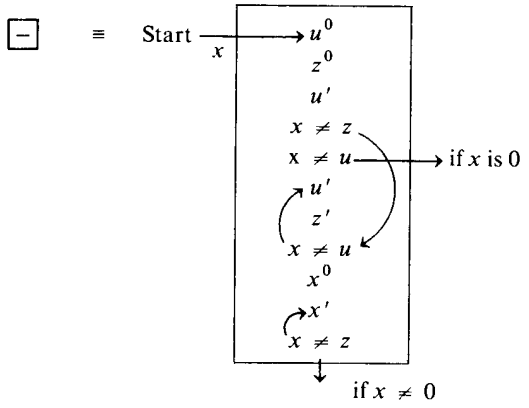
and this is all we need to show that we can compute any general-recursive function.† It follows from all this that *if a function is general-recursive then it can be computed by a program computer using only operation types* [′] *and* [−].

†For $\mu_t[\phi(t) = k]$, simply arrange for k to be in w.

We now have a welter of scattered results concerning the relations between different notions of computability. We know (from section 10.3) that, if a function is Turing-computable, then it is general-recursive. We have just demonstrated in this section that, if it is general-recursive, it is program-computer computable. It follows (and has already been shown in 11.3) that, if it is Turing-computable, then it is program-computer computable. It remains to be shown that, if a function is program-computer computable, then it is Turing-computable; this will close the loop and prove that *all* the notions are equivalent. This will drop out of the results of chapter 14, although I think the reader should be able to construct a direct proof at this point. (See the diagram in Fig. 11.8-1 at the end of this chapter.)

11.6 REPLACEMENT OF THE PREDECESSOR BY SUCCESSOR AND EQUALITY

It was hinted in 11.5 that the set of operations [0], [′], and [≠] might be sufficient for any computation. We know that [0], [′], and [−] are sufficient. The program below shows that with [0], [′], and [≠] we can obtain [−], so that these three also form a sufficient set.

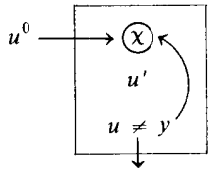


11.7 PRIMITIVE AND GENERAL RECURSION BASED ON REPETITION

In 11.6 we saw that we could obtain all primitive-recursive functions by using only the "zero," "successor," and "equality" operations. We

also obtained the general-recursive functions. This means we cannot distinguish between these lesser and greater classes of functions merely in terms of the computer operations involved in the computation. We have to look, not just at the instructions, but also at how they are strung together, and it is there that we will find the distinction.

The outstanding feature that distinguishes primitive recursion is that one knows in advance (by the value of y) *how many times the iteration* (of a primitive-recursive scheme) *will have to be done.* For general recursion, with the μ operator, one doesn't usually know this until after the work is done and the computation terminated.† We see this foreknowledge in the program of 11.5 for primitive recursion. In the non-trivial case (for which $y \neq 0$) the computation enters the loop



Since the terminating condition is that $u = y$, since y does not change, and since u begins at 0, we know that the procedure ⓧ will be applied precisely y times. Since we know this at the start, we could have expressed it more directly by defining a new operation type.

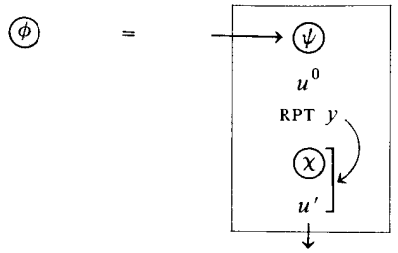
"Repeat"	RPT a:$[m, n]$	Repeat the sequence of instructions, from m to n, a times, where a is the number in register **a**. If $a = 0$, go directly to instruction $n + 1$.

It is understood that *the number of repetitions is fixed at the start* of this operation, so that it will not matter if the contents of **a** is changed in executing the RPT operation.‡ We will represent diagrammatically the range of a RPT operation by a bracket connected to the instruction, so that instruction numbers won't be necessary. Now we can rewrite the

†Otherwise it would suffice to have the bounded minimization operator, but we know (by problem 10.4-2) this won't do. And in the general case we can't have a decision procedure to tell whether the iteration will *ever* stop, let alone how long it will take.

‡But see the remark at the end of this section.

primitive-recursive program of section 11.5 as

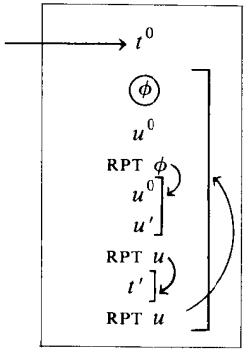


The reader can verify that this does exactly what it should and leaves the value of the defined function in register ϕ under the same conditions on ψ and χ as in 11.5. *Thus we can obtain any primitive-recursive function using only the operations* 0, ′, *and* RPT, **where the range of the RPT does not include itself.**

We emphasized the last clause, for if the RPT can include itself, we get full general recursion. To show this, we write a program to realize the minimization operator

$$\mu_t[\phi(t) = 0]$$

This is computed by the following tricky program:



The program begins by setting $t = 0$ and computing $\phi = \phi(0)$. The trick is in the next part: a register **u** is set to zero and then the program RPT ϕ: $[u^0, u']$ is applied to it. If ϕ is zero, this will be skipped, with no effect on **u** so that we will have $u = 0$. Then, because $u = 0$ the next two RPT's will be skipped and the computation finished. But if ϕ is not zero, the RPT ϕ:

$[u^0,u']$ program will end up with $u = 1$, no matter how large the value of ϕ. Then t will be increased by exactly 1 (by the RPT u:$[t']$ program) and finally the whole process will repeat because of the final RPT u program. Thus t will be increased until finally we find a t for which $\phi(t) = 0$. Result: *Any general recursive function can be computed by a program computer using only operations* [0], [′], *and* [RPT] *if we permit a* RPT *operation to lie in its own range.* In fact there need be only one such RPT operation; it (necessarily) occurs at the end of its range, and it is required to repeat only 0 to 1 times.

It should be mentioned that we have overstepped our bounds now, concerning what the RPT has to do, because in general a RPT operation could not be an instruction in the finite-state part of the machine. The reason is that the program computer has to have some way to keep track of how many RPT's remain to be done, and this might exhaust any particular amount of storage allowed in the *finite* part of the computer. RPT operations require infinite registers of their own, in general, and they must be treated differently from the other kinds of operations we have considered. Under the tight restrictions of the example above, one could get away with a finite register associated with each RPT. But if u is allowed to be greater than 1, in a self-containing RPT u range, this won't do, in general.

PROBLEM 11.7-1. Many other combinations of operation types [0], [′], [−], [⊖], [→], [≠], and [RPT] form universal bases. Find some of these bases. Which combinations of three operations are not universal bases? Invent some other operations which form universal bases with some of ours.

PROBLEM 11.7-2. Recursively defined functions can grow large very quickly. Consider the sequence of functions $P_0(a), P_1(a), \ldots$ defined by RPT and successor operation programs:

$P_0(a)$ = a'

$P_1(a)$ = RPT a
a']↙

$P_2(a)$ = RPT a
RPT a
a']↙]↙

$P_3(a)$ = RPT a
RPT a
RPT a
a']↙]↙]↙

etc.

Show that $P_0(a) = a + 1, P_1(a) = 2a, P_2(a) = a \cdot 2^a$. Can you express $P_3(a)$ in ordinary functional form? First verify, for example, that

$$P_3(2) = 2 \cdot 2^2 \cdot 2^{(2 \cdot 2^2)} = 2^{11} = 2{,}048.$$

Which is larger: $P_4(5)$ or $P_5(4)$?

PROBLEM 11.7-3. The function $F(n)$ defined so that

$$F(n) = P_n(2)$$

is a total-recursive function. It is not primitive-recursive. This can be demonstrated by showing that it grows faster than any primitive-recursive function. Can you show this by using arguments about programs for program computers using [0], [′] and [RPT]? Consider the program for any fixed primitive-recursive function and argue that when n is larger than the *length* of that program, $F(n)$ must in some sense be growing faster than that program. A complete, precise proof would be difficult, but it is worth some effort to get the general idea. $F(n)$ really does grow fast; the known universe would not hold a microfilm copy of the decimal for $F(5)$. For, while $F(4)$ has only about $2^{2^{2^2}}$ decimal digits, $F(5)$ has about

$$2^{2^{2^{2^{2^{2^2}}}}}$$

decimal digits!

PROBLEM 11.7-4. Try to write programs to replace [RPT] operations, using the earlier types of operations. Analyze the difficulties.

PROBLEM 11.7-5. Relate the function $F(n)$ to the function $A(x)$ in problem 10.5-1.

11.8 SURVEY OF OUR EQUIVALENCE PROOFS

At this point, one is likely to have lost track of precisely what we have established. Consider the diagram of Fig. 11.8-1. We have shown in sec-

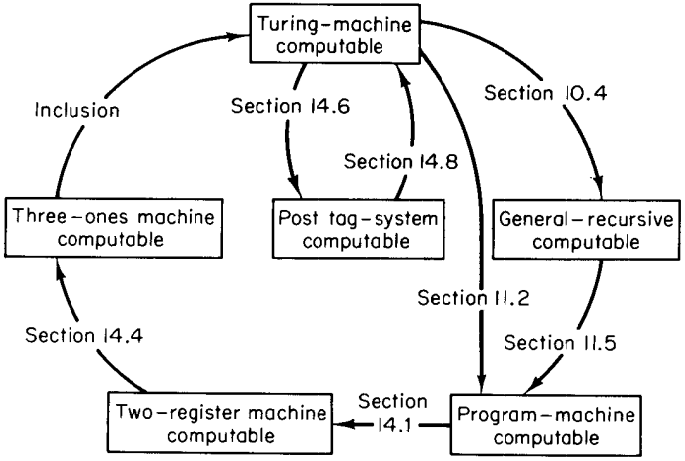


Fig. 11.8-1. Logical inclusions of computability concepts.

tion 10.4 that any Turing-machine computation can be replaced by a general-recursive function and in 11.5 that any general-recursive function can be computed by a program machine. The construction in section 11.2 shows directly that Turing-machine computations can be done by program machines. In chapter 14 we will complete the circle as indicated in the diagram. Completion of the circle will show that all the kinds of computability discussed here are *equivalent*—identical in theoretical scope despite their apparent differences in concept.

PART III

SYMBOL-MANIPULATION SYSTEMS AND COMPUTABILITY

12 THE SYMBOL-MANIPULATION SYSTEMS OF POST

12.0 INTRODUCTION

In chapter 5 we examined the idea of effective procedure and identified it with the idea of rule-obeying. We agreed that the notion of effective procedure could be identified with a system that includes

(1) A *language* for describing rules of behavior, and
(2) A *machine* that obeys statements in the language.

Up to now, we have concentrated primarily on the *machine* part of such systems, in studying Turing machines, program machines, and other instruction-obeying mechanisms. We now turn our attention toward the *language* in which rules can be expressed. Our methods will be based on the idea of Emil Post [1943] that the "expressions" or "enunciations" of a logical system or language, whatever else they may seem to be, are in the last analysis nothing but strings of symbols written in some finite alphabet. Even the most powerful mathematical or logical system is ultimately, in effect, nothing but a set of rules that tell how some *strings of symbols* may be transformed into other *strings of symbols*.

This viewpoint, in itself, appears to be an obvious but probably unprofitable truism. But, just as Turing was able to show the equivalence of his broadest notion of a computing machine with the very sharply restricted idea of a Turing machine, Post was able to reduce his broadest concept of a string-transformation system to a family of astoundingly special and simple symbol-manipulation operations. We can summarize Post's view by paraphrasing our statement, in section 5.2, of Turing's thesis:

> Any system for manipulation of symbols which could naturally be called a formal or logical (or

mathematical) system can be realized in the form of one of Post's "canonical systems."

When we see how simple are the canonical systems, this view might seem rash; but this and the following chapters will support it by showing the equivalence of these with our other formulations of effective computability.

In studying Post's systems, we will deal with rules, called "productions," which specify how one can get new strings of symbols from old ones. These productions are not, on the surface, anything like the kinds of rules we imagined for machines or for effective processes. For they are not *imperative* statements at all but are *permissive* statements. A production says how, from one statement, string, or "enunciation," of such and such a form, one *may* derive another string of a specified form. A canonical system, which is a set of such productions and some initially given statements, does not even describe a *process*; instead it specifies the extent of a set of strings by (recursively) specifying how to find things in that set. The interesting thing is how far we can proceed in our exploration without needing the idea of a machine or procedure at all. While eventually we show that the notion of a machine or process can be derived from this idea of canonical system (under certain special circumstances), the weaker, more general, idea of *set* can give us some valuable insights that are harder to see within the framework of machines and processes.

12.0.1 Plan of Part III

In this chapter we develop the idea of canonical systems, mainly by example. The formulation is oriented, as we have said, toward examining what happens to the symbols during a computation; the idea of a machine does not appear explicitly. But, by using a few technical tricks we convert what are, to begin with, hardly more than "rules of grammar" into representations of the behaviors of machines. And by a circuit of reasoning that encompasses the whole of several chapters (10–14) we finally close the loop and show that all our formulations of effectiveness are equivalent. Chapter 13 proves the beautiful theorems of Post about the equivalence of canonical systems in general with his very simple "normal-form" systems. Chapter 14 ties up a number of loose ends required to show the equivalence of Post systems with Turing machines. As by-products, we obtain a simple proof of the unsolvability of Post's famous "correspondence problem," and also a state-transition table that describes the smallest (but not simplest) universal Turing machine presently known.

12.1 AXIOMATIC SYSTEMS AND THE LOGISTIC METHOD

The *axiomatic method* is familiar to all of us from the study of Euclid's geometry. In the axiomatic method we begin with a set of "axioms"—assertions we are willing to accept, either because we believe them to be true or merely because we want to study them. Along with the axioms, we assume the validity of some "rules of inference"; these tell us precisely how we can obtain new assertions—called "theorems"—from the axioms and/or from previously deduced theorems.

Actually, Euclid paid more attention to the problem of formulating the axioms than to the problem of making perfectly clear what were the rules of inference, in geometry. Until the twentieth century, the logical structure of Euclidean geometry (as well as other axiomatic systems in mathematics) was somewhat confused because

> (1) The rules-of-inference were informal and depended on common sense,† and
> (2) the somewhat arbitrary or hypothetical nature of the axioms themselves was not clear until the discovery of the non-Euclidean geometries late in the nineteenth century.

Until the non-Euclidean geometries were discovered, there had always seemed to be a possibility that some of the axioms might be possibly derivable from some more absolute sort of reasoning process, so that it was difficult to think of geometry as a closed logical system.

> Over the last hundred years the study of the foundations of mathematics has become more intense. It was discovered that "common-sense reasoning"—ordinary logical "intuition"—was not always a sufficiently reliable tool for examining the finer details of the foundations. The study of the notion of infinity by Cantor and others led to results that were self-consistent but very surprising, and even counter-intuitive. A number of other seemingly reasonable and innocuous notions led to genuinely paradoxical results; this happened to the reasonable supposition that one could discuss relations between classes of things in terms of common-sense descriptions of the classes. (See chapter 9.) The notion of effective pro-

†Euclid's "postulates" are what we would today call axioms; some of his "axioms" we might consider to be rules of inference. Thus Euclid's axioms concerning the interchangeability of equal things might be considered to be rules of inference concerning permissible substitutions in expressions. Euclid saw, of course, that the statements concerning equal things were more general than the statements with specifically geometric content, and tried to separate these out from the "postulates."

cedure was old in mathematics, in an informal way, but the possibility of effective unsolvability, which plays so large a role in our preceding chapters, is new to mathematics; when Gödel announced his results in the 1930's, their possibility had been suspected only by a very few thinkers.† It became necessary to examine more closely the idea of a mathematical proof—to formalize all the steps of deduction—even more closely than was necessitated in the early 1900's after the discovery of Russell's paradox. The resulting theories—of the nature of "proof" itself—became more and more formal, and indeed became a new branch of mathematics. Since it is, in effect, a mathematical theory of mathematics itself, we call this study "metamathematics."‡

12.2 EFFECTIVE COMPUTABILITY AS A PREREQUISITE FOR PROOF

Before going into detail, we ought to explain how this apparent digression will lead back to the theory of machines and effective processes. The key point is this: We have to be sure, in accepting a logical system, that it is really "all there"—that the methods for deriving theorems have no dubious "intuitive" steps. Accordingly, we need to have an *effective procedure* to test whether an alleged proof of a statement is really entirely composed of deductive steps permitted by the logical system in question. Now "theorems" are obtained by applying rules of inference to previously deduced theorems, and axioms. Our requirement must then be that there is an *effective* way to verify that an alleged chain of inferences is entirely supported by correct applications of the rules.

To make this more precise, we will agree on a few definitions:

A *logistic system* L is a set of axioms and a set of rules of inference, as defined below.

An *alphabet* is a finite set of symbols. In the following definitions it is assumed that all "strings" are strings of symbols taken from some fixed alphabet $A = (a_1, a_2, \ldots, a_r)$

An *axiom* is a finite string of symbols. We will consider here only systems with finite sets of axioms.

A *rule of inference* is an effectively computable function $R(s; s_1, \ldots, s_n)$ of $n + 1$ *strings*. An R function can have only two values; 1 (for "true") and 0 (for "false"). If $R(s; s_1, \ldots, s_n) = 1$ we say that "s is immediately derivable from $s_1, \ldots,$ and s_n by the rule R." Alternatively we may write

$$\boxed{s_1, \ldots, s_n \xrightarrow[R]{} s} \qquad (R)$$

We will consider only finite sets of rules of inference. Often, in a logistic system, we will say simply that "s is immediately derivable from $s_1, \ldots s_n$" (without mentioning an R) if there is an R that can justify it. Since each R-test is effective, so is the test to see if *any* R will do, since there are only a finite set of R's.

A *proof*, in L, of a string s, is a finite sequence of strings

$$s_1, s_2, \ldots, s_K$$

such that

(1) each string s_j is either an axiom of L or else is immediately derivable from some set of strings all of which precede s_j in the sequence, and
(2) s is s_K, the last string in the sequence.

A *theorem* of L is any string for which there is a proof in L. (Show that every axiom of L is a theorem of L.)

Now our goal was to define "proof" in such a way that there is an effective way to tell whether an alleged proof is legitimate. This will be the case for the definitions given above. To see this, consider an alleged proof $s_1, \ldots, s_K$ of a string s. First see if s is s_K. (If not, this is not a proof of s in L.) Next see if s_K is an axiom. This test is effective, since once has to look at only a finite number of axioms, and each axiom is only a finite string of letters.

If s_K is s, but is not an axiom, we have to see if s_K is immediately derivable from some subset of earlier s_j's. There are a finite number of such subsets. For each we have to test a finite number of R's. Each R test is, by definition, effective. If s_K passes these tests, we go on to s_{K-1} and do the same thing (except that we don't require that s be s_{K-1}.). Clearly a finite number (namely, K) of iterations of this process will confirm or reject the proof, and clearly the whole procedure is effective, since it is composed of a finite set of finite procedures.

REMARK

Our definition of *logistic system* has a number of restrictions. One might want to consider infinite sets of axioms. We *can* do this, if we require that there be an effective test of whether an arbitrary string is an axiom; then the above argument still works, and the notion of proof is

†Notably Post. See his autobiographical remarks in Davis [1965].

‡There are several "branches" of metamathematics, of which the theory of proofs is one. Others are concerned with theories of sets, descriptions, relations between different formal systems and their decidability properties, independence and consistency of axiom systems, models, etc.

still effective. This is often done in logic, where one may use an "axiom schema"—a rule which says that "any string of such and such a form is an axiom." One might also want to allow an infinite set of rules of inference. (Again one could get an effective proof-test if one had an effective schema to determine whether there is an appropriate rule of inference at each step.) For our purposes, the finite-based systems are adequate. Shortly we shall show that any finite-based system can be reduced, without any real loss of generality, to a system with *just one axiom and a rule of inference of a kind that operates on just one previous theorem at a time*! At that point, we will have the symbolic equivalent of a machine—a step-by-step process.

12.3 PROOF-FINDING PROCEDURES

The proof-checking process just described is certainly effective; given an *alleged proof* it gives us a perfectly methodical, unambiguous procedure to decide whether the alleged proof is valid. Now let us consider a slightly different question; given an *alleged theorem*, can we decide whether it is really a theorem—that is, whether there is a proof, in the system, for it. The answer is that, while there are some logistic systems for which such procedures exist, in general there are no such procedures. In any logistic system it *is* always possible to devise a procedure that will search through all possible proofs—so that if the alleged theorem is really a theorem, this fact will be confirmed. But, in general, if the alleged theorem is not really a theorem, there is no way to find this out.

To elaborate on this a little, let us see how one can, effectively, generate all proofs—and thus all theorems—mechanically. To do this, one can use a procedure that generates all *sequences of strings*, treats each as an alleged proof, and tests each by the proof-testing procedure described above. How does one generate all *sequences of strings*? One begins with a procedure that generates all *strings*, separately. How does one do that? First generate all *one-letter strings*; that is, go through the alphabet. Next, generate all *two-letter strings*. How? Take each string generated at the previous state (i.e., one-letter strings) and append to each, in turn, every letter of the alphabet. Next we can generate all *three-letter strings* by taking each two-letter string and appending to each, in turn, every letter of the alphabet. Clearly, one gets any finite string, eventually, this way.

PROBLEM 12.3-1. Design a Turing machine that writes out on its tape all finite strings in the three-letter alphabet (a, b, c) with these strings separated by some special punctuation letter X. The machine should be subject to a constraint (see, for example, section 9.2) that the machine never moves to the left of an X, so that one can distinguish the strings the machine has enumerated from those it is currently working on.

PROBLEM 12.3-2. Before reading on, design a Turing machine that enumerates all *finite* sequences *of finite strings in* (a, b, c).

Now to enumerate all finite *sequences of finite strings*, one can modify the above procedure for generating all strings. After each step of the above procedure, one has obtained a new n-letter string. Now let us cut the string S into two parts, *in all ways*. (There are $n - 1$ ways to do this.) If we do this, after each string-generating act of the main procedure, then we have obtained all sequences of pairs of strings! In fact, we have done it so that each pair of strings is generated in exactly one way. So this gives us *all possible alleged proofs of length two*. (The original procedure gives us all of length one.) But now, suppose that after each *pair* of strings is generated, we again cut the *first* string of each pair into two parts, in all ways. (There is a variable number of ways, now, depending on where the original string was cut.) This gives us (verify this) *all possible sequences of three strings*. Indeed, suppose that each time a string is cut into two parts, we (recursively) then perform all possible cuts of the first part into two more parts. It is clear that for each originally generated string, this process, while lengthy and tedious, will be finite. In fact for an original string of length n, there are exactly 2^{n-1} sequences of strings that can be made of it.

PROBLEM 12.3-3. Prove that each string of length n can be cut into sequences of strings in exactly 2^{n-1} ways. Hint: There is a trivial proof.

So, finally, we have an effective process that generates all finite sequences of finite strings in the given alphabet. There are many other procedures to do this besides the one we have given; ours has the unimportant technical advantage that each sequence is generated exactly once. The order of generation of the sequences can serve as a Godel numbering (see section 14.3) for (alleged) proofs. In any case, as each sequence of strings is generated, one can apply our effective test to see whether it is a proof of some previously given theorem-candidate. If the string is really a theorem, this process will, in some finite time, yield a proof of it! *But if the string is not a theorem, the process will never terminate.* Hence we have a "proof procedure" for theorems, but we do not have a "theoremhood decision procedure" for strings. We will see, shortly, that in general there is no possibility of such a decision procedure.

REMARK

Some branches of mathematics actually do have decision procedures. For example, Tarski [1951] showed that a certain formulation of Euclid-

ean geometry has this property. The propositional calculus—i.e., the logic of deduction for simple sentences or for Boolean algebra—has a decision procedure; in fact the well known method of "truth tables" can be made into a decision procedure. But "elementary logic"—propositional logic with the quantifiers or qualification clauses "for all x..." and "there exists an x such that..." and symbols for functions or relations—is not, in general, decidable. See Ackermann [1954] for a study of cases where the decision problem is solvable, and Kahr, Moore, and Wang [1962] for some of the best results to date on showing which problems are not decidable. The reader who has read only this book is not quite prepared to study these papers and may have first to read a text like Rogers [1966]. Of course, even showing that a decision procedure exists, and presenting one, as in the case of Tarski's decision procedure for Euclidean geometry, does not necessarily mean that one gets a *practical* method for proving theorems! The methods obtained by logical analysis of a decision problem usually lead to incredibly lengthy computations in cases of practical interest.

12.4 POST'S PRODUCTIONS. CANONICAL FORMS FOR RULES OF INFERENCE

In 12.2 we defined a rule of inference to be an effective test to decide whether a string s can be deduced from a set of strings $s_1, \ldots, s_n$. We required the test to be effective so that we could require the test of a proof to be effective. But we did not really tie things down securely enough, for one might still make a rule of inference depend (in some effective way) upon some understanding of what the strings "mean." That is, one might have some rule of inference depend upon a certain "interpretation" of the strings as asserting things about some well-understood subject matter; the strings might, for example, be sentences in English. (For example, the strings in chapter 4—the "regular expressions"—were understood to represent the "regular sets," and our proofs about regular expressions used references to these understood meanings.)

To avoid such dangerous questions, we propose to restrict ourselves to rules of inference that concern themselves entirely with the arrangement of symbols within strings—i.e., to the visible form of the strings as printed on a page—and we rule out reference to meanings. This will force us, for the present, to direct our attention toward what is often called the domain of "syntax"—questions of how expressions are assembled and analysed—rather than the domain of "semantics"—questions of the meanings of expressions.

To make it plausible that this can actually be done with any prospect of success, we will paraphrase the arguments of Turing (as recounted in chapter 5) as he might have applied them to the situation of an imaginary finite mathematician who has to manipulate symbolic mathematical expressions.

At all times our mathematician must work with finite strings of symbols using a finite alphabet, a finite set of axioms, and a finite set of rules of inference. Imagine that he is verifying the validity of an alleged proof of an alleged theorem. At each step, then, he will be confronted with some assertion—that is, a string of symbols

$$s = a_{i_1} \ldots a_{i_n}$$

and also with a sequence of assertions whose proofs he has already verified

$$s_1 = a_{11}\, a_{12} \ldots a_{1n_1}$$
$$s_2 = a_{21}\, a_{22} \ldots a_{2n_2}$$
$$s_r = a_{r1}\, a_{r2} \ldots a_{rn_r}$$

where n_j is the number of letters in the jth string. He must use one of the rules of inference, and he may, for example, try to apply each rule systematically to each subset of the established strings.

Now, paraphrasing Turing's argument, we will further suppose that the mathematician's resources are limited to a certain set of talents:

> He can "scan" a string of symbols and recognize therein certain fixed subsequences.†
>
> He can dissect these out of the string, and keep track of the remaining parts.
>
> He can rearrange the parts, inserting certain fixed strings in certain positions, and deleting parts he does not want.

We shall see that these abilities are all that are needed to verify proofs or to perform any other effective procedure!

A rule telling precisely how to dissect a string and rearrange its parts (perhaps deleting some and adding others) is called a "production." Rather than begin with a precise definition, we will start with a few examples of simple but complete and useful systems based on productions. Then, when we formalize the definition, it will be perfectly clear what is meant and why it is done that way.

†Later (in chapters 13 and 14) we will see that all that is really necessary is the ability to identify a single symbol—the first of a string—and act accordingly. This parallels Turing's observation that it is enough for a Turing machine to examine one square of its tape at a time.

EXAMPLE 1: THE EVEN NUMBERS

Alphabet: The single symbol 1.
Axiom: The string 11.
Production: If any string \$ is a theorem, then so is the string \$11. It is convenient to write this rule of inference simply as

$$\$ \rightarrow \$11$$

It is evident that the theorems of this system are precisely the strings

$$11, \quad 1111, \quad 111111, \quad 11111111, \quad \text{etc.}$$

that is, the even numbers expressed in the unary number system.

EXAMPLE 2: THE ODD NUMBERS

Alphabet: 1
Axiom: 1
Production: \$ → \$11

EXAMPLE 3: THE PALINDROMES

Alphabet: a, b, c.
Axioms: a, b, c, aa, bb, cc.
Productions: \$ → $a\$a$
\$ → $b\$b$
\$ → $c\$c$

The "palindromes" are the strings that read the same backwards and forwards, like *cabac* or *abcbbcba*. Clearly, if we have a string \$ that is already a palindrome, it will remain so if we add the same letter to the beginning and end. Also clearly, we can obtain all palindromes by building in this way out from the middle.

PROBLEM 12.4-1. Prove that this gives *all* the palindromes. (It obviously gives nothing else.)

PROBLEM 12.4-2. This example, while quite trivial, is interesting because this is a set of strings that cannot be "recognized" (in the sense of chapter 4) by a finite-state machine. Prove this. Show that if we remove the last three axioms but append the production

$$\$ \rightarrow \$\$$$

we still obtain the same set of theorems.

EXAMPLE 4: SIMPLE ARITHMETIC EQUATIONS

Suppose that we want a system whose theorems are *all true statements about adding positive integers*—that is, all sentences like

$$3 + 5 = 8$$
$$21 + 35 = 56, \quad \text{etc.}$$

For simplicity, we use unary notation, so that the theorems will resemble

$$111 + 11111 = 11111111$$

Alphabet: 1, +, =
Axiom: 1 + 1 = 11
Productions: $\$_1 + \$_2 = \$_3 \rightarrow \$_1 1 + \$_2 = \$_3 1$ (π_1)
$\$_1 + \$_2 = \$_3 \rightarrow \$_1 + \$_2 1 = \$_3 1$ (π_2)

Here the first production means: If there is a theorem that consists of some string $\$_1$, followed by a '+', then another string $\$_2$, then an '=', and finally another string $\$_3$; we can make a new theorem that consists of the first segment $\$_1$, then a '1', then '+', then $\$_2$, then '=', then $\$_3$, and finally another '1'. To see how this works, we derive the theorem that means "2 + 3 = 5"

1 + 1 = 11	axiom
11 + 1 = 111	by π_1
11 + 11 = 1111	by π_2
11 + 111 = 11111	by π_2

Another proof of the same theorem is

1 + 1 = 11	axiom
1 + 11 = 111	by π_2
1 + 111 = 1111	by π_2
11 + 111 = 11111	by π_1

There can be many proofs for the same theorem, in such a system.

PROBLEM 12.4-3. Replace π_2 by $\$_1 + \$_2 = \$_3 \rightarrow \$_2 + \$_1 = \$_3$ and prove the same theorem in the new system.

One can do much the same for multiplication:

Alphabet: 1, ×, =
Axiom: 1 × 1 = 1
Productions: $\$_1 \times \$_2 = \$_3 \rightarrow \$_1 1 \times \$_2 = \$_3 \$_2$
$\$_1 \times \$_2 = \$_3 \rightarrow \$_2 \times \$_1 = \$_3$

Prove that 3 × 4 = 12 in this system.

PROBLEM 12.4-4. Design a system whose theorems include arithmetic statements involving both addition and multiplication. This may be difficult at this point but will be easier after further examples.

EXAMPLE 5: WELL-FORMED STRINGS OF PARENTHESES

In 4.2.2 we defined the set of "well-formed strings of parentheses," namely the strings like

$$(),\quad (()),\quad (()()),\quad (())(),\quad ((()())((())))$$

in which each left parenthesis has a matching right-hand mate. We can obtain all and only such strings as theorems of the system:

Alphabet: (,)
Axiom: ()
Productions: $\$ \rightarrow (\$)$ (π_1)
$\$ \rightarrow \$\$$ (π_2)
$\$_1()\$_2 \rightarrow \$_1\$_2$ (π_3)

For example, to derive the string (()(())):

()	axiom
(())	by π_1
(())(())	by π_2
((())(()))	by π_1
(()(()))	by π_3

PROBLEM 12.4-5. Prove (()()((()()))) in this system.

PROBLEM 12.4-6. Consider the same alphabet and axiom with the single production:

$$\$_1\,\$_2 \rightarrow \$_1()\$_2$$

Prove that this system generates all and only the well-formed parenthesis strings. Note: a $ is allowed to represent an empty, or "null" string, so that () → ()() is permitted, for example.

12.5 DEFINITIONS OF PRODUCTION AND CANONICAL SYSTEM

Now let us define "production" more precisely. In each example of 12.4, every production could be written as a special case of the general form

ANTECEDENT	CONSEQUENT	
$g_0\$_1 g_1 \$_2 \ldots \$_n g_n$	$\rightarrow h_0\$'_1 h_1 \$'_2 \ldots \$'_m h_m$	(π)

with the qualifications:

Each g_i and h_i is a certain *fixed* string; g_0 and g_n are often null, and some of the h's can be null.
Each $\$_i$ is an "arbitrary" or "variable" string, which can be null.
Each $\$'_i$ is to be replaced by a certain one of the $\$_i$.

Take for example the production

$$\$_1 \times \$_2 = \$_3 \rightarrow \$_1 1 \times \$_2 = \$_3\$_2$$

from example 4 of 12.4. We can represent it in the form above by making the assignments:

g_0 = null g_3 = null h_2 = '='
g_1 = '×' h_0 = null h_3 = null
g_2 = '=' h_1 = '1×' h_4 = null

and

$\$'_1 = \$_1$, $\$'_2 = \$_2$
$\$'_3 = \$_3$, $\$'_4 = \$_2$

Note that two (or more) of the $\$'_i$-s can be the same $\$_i$, as in the production above, where $\$'_4 = \$'_2 = \$_2$. This breaks up, diagrammatically, as:

ANTECEDENT CONSEQUENT

$$\begin{array}{ccccccccccccc} \$_1 & \times & \$_2 & = & \$_3 & \rightarrow & \$_1 & 1 & \times & \$_2 & = & \$_3 & \$_2 \\ | & | & | & | & | & & | & | & | & | & | & | & | \\ g_0\$_1 & g_1 & \$_2 & g_2 & \$_3 g_3 & \rightarrow & h_0\$_1 & h_1 & \$_2 & h_2 & \$_3 h_3 & \$_2 h_4 & \end{array}$$

REMARKS

Post's most general formulation allowed each production to have several *antecedents*. This is discussed in 13.2, and we prefer not to introduce this complication here; in 13.2 we show that the more general form is equivalent, in a sense, to the special single-antecedent forms used here.

Also in Post's most general formulations, he allowed two of the $'s in the *antecedent* to be the same. This meant that the rule of inference would apply only to a string (theorem) in which there was an exact repetition of some (variable) sub-string in two places in the antecedent. We prefer to prohibit antecedents of this form, not because we want to restrict the generality of the systems, but because it would run counter to our intuitive picture of what ought to be permitted as elementary, unitary actions. The recognition of the identity of two arbitrarily long strings ought to have to be done by an iterative process; otherwise it violates Turing's dictum (see 5.3) about what can be "seen at a glance" and what requires a multi-stage process.[1]

DEFINITIONS

A *production* is a string-transforming rule of the general form of π above, or (more generally) of π in 13.2.

A *canonical system* is a logistic system specified by

(1) an *alphabet* A
(2) some *axioms* (strings in A)
(3) some *productions* whose constant strings are strings in A.

12.6 CANONICAL SYSTEMS FOR REPRESENTATION OF TURING MACHINES

Now we can show how a formal system, with only axioms and productions, can be arranged to model a process! On the surface, a formal system seems *permissive* rather than *imperative*; there would seem to be nothing that corresponds to the *process control* in a machine, nothing like an obvious mechanism that dictates "what is to be done *next*." Indeed, there is no notion of time or sequence, except, perhaps, the sequence of steps in a proof of a theorem. *Because, in general, there are many different proofs of a theorem, one would not expect to be able to use the proof-step sequence as a process-control mechanism.* But one can. By using some tricks—mainly the use of special punctuation symbols in various ways—we can, indeed, embed the notion of a machine within the concept of a formal system with only axioms and productions. We will show this by constructing a formal system which, in a very straightforward way, "simulates" the activity of a Turing machine.

EXAMPLE 6: PRODUCTIONS FOR TURING-MACHINE COMPLETE-STATE CHANGES

Let $(s_1, s_2, \ldots, s_r)$ be the alphabet, and $(q_1, q_2, \ldots, q_n)$ the states, of a certain Turing machine T. At any time t, T's tape will contain some finite sequence of symbols

$$s_{i_1}, s_{i_2}, \ldots, s_{i_{(n_t)}} \qquad (S)$$

where n_t is the length of the written part of the tape at time t. To show the complete state of the Turing machine at time t, we have also to specify (1) the current internal state of the machine and (2) where the machine is located on the tape. We can incorporate *all* of these facts, in a single tape-representing string, by including the machine's state-symbol at the appropriate place, e.g., by writing

$$s_{i_1}, s_{i_2}, \ldots, s_{i_{k-1}}, \boxed{q_i}, s_{i_k}, \ldots, s_{i_{n_t}}$$

This string is interpreted as follows: the machine is in state q_i; it is scanning the kth written square of its tape; the tape has on it the letter sequence (S); it is understood that the q_i is *not* written on the *machine's* tape.

Now we can represent the machine's operation by a set of Post productions! Let the quintuples of T be

$$(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$$

If d_{ij} is "*Right*," then the machine ought to proceed from any complete state represented by

$$\ldots s_k q_i s_j \ldots$$

to that represented by

$$\ldots s_k s_{ij} q_{ij} \ldots$$

while if d_{ij} is "*Left*," the machine ought to proceed to that represented by

$$\ldots q_{ij} s_k s_{ij} \ldots$$

This suggests using a set of productions, one for each (i, j, k) triple:

$$\$_1 s_k q_i s_j \$_2 \rightarrow \$_1 s_k s_{ij} q_{ij} \$_2 \qquad (\text{if } d_{ij} \text{ is Right})$$
$$\$_1 s_k q_i s_j \$_2 \rightarrow \$_1 q_{ij} s_k s_{ij} \$_2 \qquad (\text{if } d_{ij} \text{ is Left})$$

These rules cannot work when the machine—that is, when the symbol q_i—comes to an end of the written part of the tape. But we can make the system automatically extend the representation by adjoining the productions

$$\$q_i \rightarrow \$q_i 0 \qquad (\text{all } i)$$
$$q_i \$ \rightarrow 0 q_i \$ \qquad (\text{all } i)$$

These serve to add blank squares whenever the machine comes to an end of the tape. We can thus think of the Turing machine's computation as always represented by a finite string of symbols, with provision for lengthening this string when necessary. We now make the following assertion.

ASSERTION

Given the above productions, and given a string containing one q_i symbol for an axiom, the theorems of this canonical system will be precisely the sequence of all future complete states of the Turing machine, if started in the complete state represented by the axiom. Each theorem of the system will have a single, unique, proof, and the steps of the proof will be exactly those of the steps in that computation. (We include the tape-extending operations as steps in the Turing machine's operation.)

The proof of the assertion is simply that, if a string contains only one q symbol, then only one production can apply to it (Verify this!), and the

result produces a string that represents the next step of the Turing machine's computation because of the way the quintuples are realized in the system of productions.

PROBLEM 12.6-1. Why would the assertion be false if the s_k's were left out of the first (right) productions?

PROBLEM 12.6-2. Carry out this construction for the Turing machine of 6.1.1 (p. 120).

PROBLEM 12.6-3. Suppose that a certain Post canonical system has the single production

$$\$_1 xy\$_2 \rightarrow \$_1\$_2$$

that it is known to have a single axiom, and that it is known that the string xy is a theorem of the system. What can you say about the unknown axiom? More precisely, describe the class of all axioms from which the string xy can be derived, using only this production. Prove your statement.

EXAMPLE 7: A CANONICAL SYSTEM FOR GENERATING THE SQUARE NUMBERS

For our next example, we want to generate the sequence of numbers

$$1, 4, 9, 16, 25, \ldots$$

(in the form of unary strings: 1, 1111, 111111111, ...). Observing that $(n + 1)^2 = n^2 + (2n + 1)$—that is, that we get from one square number to the next by adding the corresponding *odd* number—we use the system:

Alphabet: 1, P

Axiom: 1 A

Production: $\$_1 A \$_2 \rightarrow \$_1 11 A \$_2\$_1$

This generates, in sequence, the strings

$1P$	$= 1^1P1^0$
$111P1$	$= 1^3P1^1$
$11111P1111$	$= 1^5P1^4$
$1111111P111111111$	$= 1^7P1^9$
$11111111P1111111111111111$	$= 1^9P1^{16}$

In a sense, the square numbers are being generated. The symbol P is used to separate two quantities; on the left is computed the next odd number to be used; on the right is the sum of the odd numbers of earlier stages (which is also the desired square number). Generalizing, we see that one could use more punctuators like P to keep track of more different auxiliary quantities that one might want to keep, during a computation. In the next example, we will keep track of three quantities in this way.

PROBLEM 12.6-4. See if you can sketch, at this point, how one could realize the computations of a program machine, of the kind described in 11.1, by a system of productions, using one punctuation letter for each machine register. How many auxiliary letters are really needed? The solution is given in section 12.8.

PROBLEM 12.6-5. (Fairly difficult.) Prove that there is no system of productions whose theorems are the square numbers (in unary notation) which uses only the symbol '1' in its alphabet—that is, which has no extra punctuation letters.

12.7 CANONICAL EXTENSIONS. AUXILIARY ALPHABETS

There is a serious defect in example 7 of the previous section. We would really like to find a canonical system whose *only* theorems are the strings 1, 1111, 111111111, etc. Instead, we found a system which generates, inside its theorems, the desired information but does not produce it in the desired form. If we adjoin one more production

$$\$_1 P\$_2 \rightarrow \$_2$$

then we obtain, in addition to the theorems already found, also the theorems

$$1, 1111, 111111111, 1111111111111111, \text{etc.}$$

We now have the theorems we want, but we also have the "working results"

$$1P, 111P1, 11111P1111, \text{etc.}$$

which we do not want. Now we can easily distinguish between the desired theorems and the working results because the latter all contain the symbol P while the former do not. So we can say that

The square numbers are those theorems of the canonical system

Alphabet: 1, P

Axiom: $1P$

Productions: $\$_1 P\$_2 \rightarrow \$_1 11 P\$_2\$_1$

$\$_1 P\$_2 \rightarrow \$_2$

which are also strings in the smaller alphabet containing only 1.

It turns out, in general, that auxiliary letters, like the P above, are necessary when canonical systems are used to produce sets of theorems of

theoretical interest. Let us make a general definition, to recognize and deal with this fact.

Suppose that we are interested in a certain system M, whose theorems are expressed in a certain alphabet A. (M is not necessarily a canonical system.) Suppose that M' is another system, with a larger alphabet A'. Some theorems of M' may use only letters in A, other theorems will use additional letters.

DEFINITION

If the theorems of M are precisely those theorems of M' that use only the letters of A, then we say that M' is an extension of M over A. If M' is a Post canonical system, then we say M' is a *canonical extension* of M over A.

In example 7 of section 12.6, the system is a canonical extension of any system whose theorems are the square numbers. The first production does the work, while the second production is used as a sort of output device; it converts a string with a P to one without a P. In doing this, it serves to *release* the result—a square number—which cannot be further modified (because it contains no P). In this way we can use an "auxiliary letter"—one in the extension alphabet but not in the original—to control what is transformed and to protect already-derived theorems from being incorrectly transformed. The next section gives a more elaborate example of such a computation.

PROBLEM 12.7-1. Construct a Post canonical extension for the set of repeated strings, e.g., those that have the form \$\$. The palindromes of section 12.4, example 3, were realized without using any extension letters. Can this be done here? If not, prove it. The alphabet should have two original letters.

EXAMPLE 8: A CANONICAL EXTENSION FOR THE PRIME NUMBERS

This example shows how a few productions can lead to quite complicated behavior. It is an extension of the (unary) prime integers over the alphabet '1'. We have added the string '11' as an axiom because the system otherwise generates only the primes from '111' on.

Alphabet: $1, A, B, C, D$

Axioms: $A111, 11$

Productions:

$$A\$ \rightarrow A\$1 \qquad (\pi_1)$$

$$A\$1 \rightarrow C\$DB\$1 \qquad (\pi_2)$$

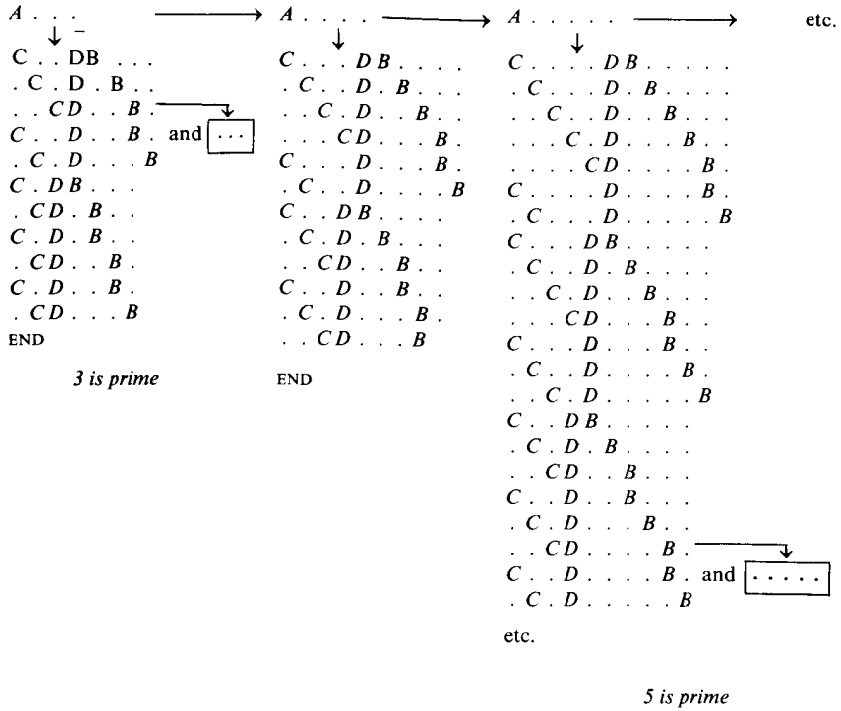
$$\$_1 1 D \$_2 1 \rightarrow 1 \$_1 D 1 \$_2 \qquad (\pi_3)$$

$$\$_1 C D \$_2 1 \rightarrow C \$_1 D \$_2 1 \qquad (\pi_4)$$

$$\$_1 C \$_2 1 D \$_3 B \rightarrow C \$_1 \$_2 D B \$_3 \qquad (\pi_5)$$

$$11CD\$B1 \rightarrow \$1 \qquad (\pi_6)$$

Table 12.7-1



The diagram in Table 12.7-1 traces out all the strings generated by the system. We use a dot instead of '1' for typographical clarity.

12.8 CANONICAL SYSTEMS FOR PROGRAM-MACHINES

By using a rich variety of auxiliary letters, one can design canonical systems to simulate the steps of a complicated process. We have already done this (section 12.6, example 6) by using a new auxiliary letter for each state of a Turing machine. We can do the same for each of the instructions of a program machine (chapter 11). Let the instructions of a program machine P be numbered $I_1, \ldots, I_n$ and let the registers of the machine be $R_1, \ldots, R_r$. Let us consider a machine whose instructions are of the two kinds: Each I_j is either

add 1 to R_k and go to I_{j+1}

or

if R_k contains zero **then** go to $I_{j'}$, **else** subtract 1 and go to I_{j+1}.

By section 11.4, this is a universal base for program machines. We can simulate this program machine with a canonical system constructed as follows:

Alphabet: $I_1, \ldots, I_n, R_1, \ldots, R_r, R_{r+1}, 1$

Axiom: $I_1 R_1 11 \ldots 11 R_2 11 \ldots 11 R_3 \ldots R_r 11 \ldots 11 R_{r+1}$

which is understood to mean that the machine starts with I_1 and has the given unary numbers in its registers at the start.

Productions: If I_j is "Add 1 to R_k and go to I_{j+1}," use:

$$I_j \$_1 R_k \$_2 \rightarrow I_{j+1} \$_1 R_k 1 \$_2$$

If I_j is "**If** $R_k \neq 0$ **then** subtract 1 and go to I_{j+1}, **else** go to $I_{j'}$," use the pair:

$$I_j \$_1 R_k R_{k+1} \$_2 \rightarrow I_{j'} \$_1 R_k R_{k+1} \$_2$$
$$I_j \$_1 R_k 1 \$_2 \rightarrow I_{j+1} R_k \$_2$$

This system is truly "monogenic" (see 14.6). That is, given a string S, it is never possible for more than one production to be applicable to S. For, in the case of an *addition* instruction I_j, there is only one production that begins with the letter I_j. In the case of a *subtraction* instruction I_j, there are two productions beginning with I_j, but since, in any string, R_k is followed either by a '1' or by R_{k+1}, only one of these two productions can apply. To release the final result as a number, suppose that I_n is a halt instruction and the answer is to be found in R_r. Then we adjoin the production

$$I_n \$_1 R_r \$_2 R_{r+1} \rightarrow \$_2$$

which will release the unary string contained in register R_r. For an improved form of this theorem, see the remark at the end of section 14.1.

PROBLEM 12.8-1. Show how to make a canonical extension for the set of strings represented by a regular expression.

PROBLEM 12.8-2. Show how to construct a set of productions that gives the set of strings (as an extension) recognized by a finite-state machine, directly from the state-transition table of the machine, without reference to the regular-expression analysis.

PROBLEM 12.8-3. Show, given a system of Post productions and axioms —i.e., given a canonical system—how to make a Turing machine that will generate all the theorems of this system.

NOTE

1. Several recent advances in computer programming languages are based on string-dissecting operations that resemble closely Post productions. In particular they derive their power, in part, from unrestricted use of occurrences of the same string variable. The first such language was COMIT, developed by Victor Yngve [1962] for use in research on linguistic analysis. Following this came SNOBOL (see Farber [1964]) and a series of related languages embedded in the LISP system, described in Bobrow [1964], Guzman [1966], and Teitelman [1966].

13 POST'S NORMAL-FORM THEOREM

13.0 INTRODUCTION

The theorem proved in this chapter is the normal-form theorem of Post's [1943] paper. I feel that it is one of the most beautiful theorems in mathematics: Any formal system can be reduced to a Post canonical system with a single axiom and only productions of the simple form

$$g\$ \rightarrow \$h \qquad \text{("Normal" production)}$$

Post's proof of the theorem is quite lengthy. Our proof seems to us considerably simpler, because it is less concerned with the order in which things happen. Some clarity is gained because of fewer auxiliary symbols; some is perhaps lost because of the simultaneous operation of many rules. To make up for this, we illustrate the proof with a detailed example that should help the reader see intuitively why the theorem is true.

We will state and prove the theorem in a series of forms of increasing strength, then give some new results using the same general methods used in the proofs. In the subsequent chapter we will examine the relation between these and the results of earlier chapters.

13.1 THE NORMAL-FORM THEOREM FOR SINGLE-ANTECEDENT PRODUCTIONS

THEOREM 13.1

Given a Post canonical system P with alphabet A and productions of the form

$$g_0\$_1 g_1\$_2 \ldots \$_n g_n \rightarrow h_0\$'_1 h_1\$'_2 \ldots \$'_m h_m \qquad (\pi)$$

we can construct a new "normal" canonical system P^* whose productions all have the simple form

$$g\$ \rightarrow \$h$$

and which is a canonical extension of P over A. That is, those theorems of P^* which contain only letters from the original alphabet A of P will be precisely all the theorems of P.

Recalling the meaning of the production (π) we can foresee the following difficulties. We will have to overcome the apparent limitation of normal productions to "look" only at the initial letters of a theorem. They can tell when a string begins with g_0, but how can they check whether a string contains, for example, a copy of g_2 preceded by a copy of g_1? More precisely, in spite of this limitation, we have to (1) determine when a string has the antecedent form of (π)—that is, when a string contains non-overlapping occurrences of $g_0, g_1, \ldots, g_n$ in that order, and (2) separate out the \$ strings that come between the discovered g_i's and insert copies of them into the proper positions of the consequent form—that is, between copies of the constant strings $h_0, h_1, \ldots, h_m$. Furthermore, we must (3) provide that if the antecedent form is satisfied in several different ways, then all corresponding consequent forms for these are generated.

The construction turns around a technique of "rotating" strings so that each part of the string eventually comes around to the front.

In all constructions we will use upper-case letters for the auxiliary symbols introduced for the new system P^*. We will begin by supposing that the letter T (for "Theorem of P") is available and that theorems of P are represented in P^* by their form in P preceded by the single letter T. We proceed first by illustrating the construction of P^* for a particular example.

13.1.1 Antecedent form recognition

Suppose (for example) that our production is actually

$$Tab\$_1 cb\$_2 b\$_3 \rightarrow Taa\$_3 bb\$_1 aa\$_3 c \qquad (\pi\text{-example})$$

so that we have

$g_0 = Tab$	$h_0 = Taa$
$g_1 = cb$	$h_1 = bb$
$g_2 = b$	$h_2 = aa$
$g_3 = \text{null}$	$h_3 = c$

and we want to apply this to the string S

$$Tabefcbgbdc \qquad (S)$$

(which does fit the antecedent form, with $\$_1 = ef$, $\$_2 = g$, and $\$_3 = dc$). The result should be

$$T\underline{aa}dc\underline{bb}ef\underline{aa}dc\underline{c}.$$

We begin by providing normal productions to check whether a string has this antecedent form. We will use the following system:

Alphabet: letters of A, and $T, T_1, T_2, \ldots, T_n$

Productions:
$$Tab\$ \to \$T_1$$
$$T_1cb\$ \to \$T_2$$
$$T_2b\$ \to \$T_3$$
$$x\$ \to \$x \qquad \text{(for all } x \text{ in } A\text{)}$$
$$T_ix\$ \to \$T_i \qquad \text{(for all } x \text{ in } A\text{, and all } i = 1, \ldots, n\text{)}$$

This system of productions has the recognition property: *If a string has the form* $Tab\,\$_1cb\$_2b\$_3$, *then, and only then, this system will produce the string* T_3. To see why this is so, let us apply these productions to the string. The following strings are produced in order:

$T\ a\ b\ e\ f\ c\ b\ g\ b\ d\ c \to$
$e\ f\ c\ b\ g\ b\ d\ c\ T_1 \to$
$f\ c\ b\ g\ b\ d\ c\ T_1\ e \to$
$c\ b\ g\ b\ d\ c\ T_1\ e\ f \to$
$b\ g\ b\ d\ c\ T_1\ e\ f\ c \to$
$g\ b\ d\ c\ T_1\ e\ f\ c\ b \to$
$b\ d\ c\ T_1\ e\ f\ c\ b\ g \to$
$d\ c\ T_1\ e\ f\ c\ b\ g\ b \to$
$c\ T_1\ e\ f\ c\ b\ g\ b\ d \to$
$T_1\ e\ f\ c\ b\ g\ b\ d\ c$

so that the T symbol has rotated around to the front of the string. Now (only) the production $T_1e\$ \to \$T_1$ applies, yielding

$T_1\ e\ f\ c\ b\ g\ b\ d\ c \to$
$f\ c\ b\ g\ b\ d\ c\ T_1 \to$
$c\ b\ g\ b\ d\ c\ T_1\ f \to$
$\ldots$
$T_1\ f\ c\ b\ g\ b\ d\ c \to$
$c\ b\ g\ b\ d\ c\ T_1 \to$
$\ldots$
$c\ T_1\ c\ b\ g\ b\ d \to$
$T_1\ c\ b\ g\ b\ d\ c$

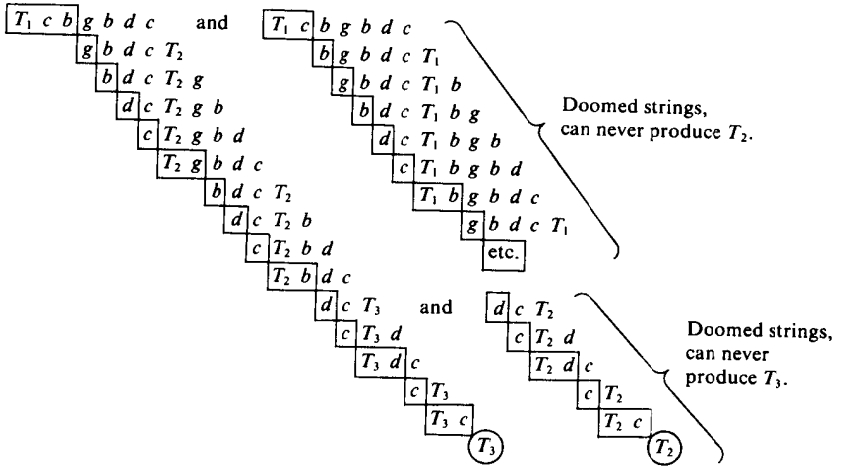
At this point, for the first time, two different productions can apply, generating two families of strings: that is, both

$$T_1cb\$ \to \$T_2$$

and

$$T_1c\$ \to \$T_1$$

apply, yielding



So the system produced the string T_3. Observe that, to produce T_3 the system *must* produce T_1 and T_2 along the way, in order, and that each advance—in the index of the T symbol—requires that the T symbol encounter g_i, the required substring of the antecedent form. Thus, for instance, the string to the upper right, $bgbdcT_1$ is "doomed" in that its T_1 will never encounter the substring cb it needs to become upgraded to T_2. Clearly, then, if the string has the antecedent form, there is a route which produces T_3, and if the string has not the required form, production of T_3 is impossible. The example thus illustrates a perfectly general method:

The system of productions:

$$\begin{cases} x\$ \to \$x & \text{(all } x \text{ in } A\text{)} \\ T_jx\$ \to \$T_j & \text{(all } x \text{ in } A\text{, all } j\text{)} \\ T_jg_j\$ \to \$T_{j+1} & \text{(all } j\text{)} \end{cases}$$

produces the string T_{n+1} *from a string* S *if and only if the string* S *contains the non-overlapping substrings* $g_0, g_1, \ldots, g_n$ *in that order.*

We have now solved most of our problem: we know how to get around

the apparent limitations of the normal-form production (namely, by the rotation trick), and we know how to recognize that a string has a given antecedent form. We still have to show how to generate the appropriate consequent form.

REMARK

If the input string has the antecedent form in different ways, then our system will produce T_3 by different routes. For example

$$Tabcbcbba$$

will have three interpretations:

$$\underline{Tab}cb\underline{cbb}\underline{a}, \quad \underline{Tab}\underline{c}b\underline{cbb}\underline{a}, \quad \text{and} \quad \underline{Tab}\underline{c}b\underline{c}b\underline{b}\underline{a}$$
$$\$_1 \quad \$_3 \qquad \$_2 \quad \$_3 \qquad \$_2 \quad \$_3$$

These should eventually result in three different consequent strings.

The only trouble with the present system is that, while it recognizes antecedents, it destroys the information about what the \$'s are, and this information is needed to construct the consequent. To save this information, we will use the more complicated system of productions below. We introduce an array of new upper-case letters:

$$A^j_x \qquad \begin{matrix} (j = 1, \ldots, n) \\ (\text{all } x \text{ in } A) \end{matrix}$$

and a new (and final) system of productions:

$$\left.\begin{array}{rl} x\$ \rightarrow & \$x \\ A^j_x\$ \rightarrow & \$A^j_x \\ T_j x\$ \rightarrow & \$A^j_x T_j \\ T_j g_j\$ \rightarrow & \$T_{j+1} \\ T_n A^j_x\$ \rightarrow & \$QA^j_x \end{array}\right\} \begin{matrix} (\text{all } x \text{ in } A) \\ (j = 1, \ldots, n) \end{matrix} \qquad (P^*)$$

This system is like the previous system except that it preserves the \$ information instead of deleting it. For example, on the string

$$\underline{Tab}b\underline{c}b\underline{aab}$$

we obtain, following only the main path and ignoring doomed strings,

$$\boxed{T\ a\ b}\ b\ c\ b\ a\ a\ b$$
$$b\ c\ b\ a\ a\ b\ T_1$$
$$\cdots$$
$$\boxed{T_1\ b}\ c\ b\ a\ a\ b$$
$$c\ b\ a\ a\ b\ A^1_b\ T_1$$
$$\cdots$$
$$A^1_b\ T_1\ c\ b\ a\ a\ b$$
$$\boxed{T_1\ c\ b}\ a\ a\ b\ A^1_b$$
$$a\ a\ b\ A^1_b\ T_2$$
$$\cdots$$
$$\boxed{T_2\ a}\ a\ b\ A^1_b$$
$$a\ b\ A^1_b\ A^2_a\ T_2$$
$$\cdots$$
$$b\ A^1_b\ A^2_a\ A^2_a\ T_2$$
$$\cdots$$
$$\boxed{T_2\ b}\ A^1_b\ A^2_a\ A^2_a$$
$$A^1_b\ A^2_a\ A^2_a\ T_3$$
$$\cdots$$
$$\boxed{T_3\ A^1_b}\ A^2_a\ A^2_a$$
$$A^2_a\ A^2_a\ Q\ A^1_b$$
$$\cdots$$
$$Q\ A^1_b\ A^2_a\ A^2_a$$

asserting that $\$_1 = b$ and $\$_2 = aa$. Similarly the string

$$Tabefcbgbdc$$

yields

$$QA^1_e A^1_f A^2_g A^3_d A^3_c \qquad (S')$$

while the string

$$Tabcbcbba$$

yields

$$QA^1_c A^1_b A^3_a, \quad QA^2_c A^2_b A^3_a, \quad \text{and} \quad QA^2_c A^3_b A^3_a$$

13.1.2 Consequent form construction

To construct the consequent, we have to make copies of the \$ strings into the proper positions in the consequent form. That form itself will be

set up by the production

$$\boxed{Q\$ \rightarrow \$h_0 V_{i_1} h_1 V_{i_2} \ldots V_{i_m} h_m Z Y} \qquad (P^*\text{—continued})$$

in which the h's are the constant strings of the consequent and the V's are new letters indicating where the corresponding \$'s are to go—that is, a copy of $\$_i$ is to be inserted at each occurrence of its V_i. The new letters Z and Y will be used later. When this production is added to the system of the last section and applied to the string *Tabefcbgbdc* we obtain (using the h's of our original example)

$$A_e^1 A_f^1 A_g^2 A_d^3 A_c^3 aaV_3 bbV_1 aaV_3 cZY \qquad (S'')$$

Our trick will be to cause the A's to "trickle" across the string, without changing their order. Whenever an A_x^i passes a V_i with the same index (i), it will leave behind a copy of its subscript letter x. Thus when all the A's have passed all the V's there will be a copy of $\$_i$ next to each V_i!

The productions to do the trickling are:

$$\boxed{\begin{array}{ll} A_x^i y\$ \rightarrow \$yA_x^i & (\text{all } x \text{ in } A,\ y \text{ in } A) \\ & (\text{all } i = 1, \ldots, n) \end{array}}$$

which lets the A's pass over lower-case letters,

$$\boxed{A_x^i V_j\$ \rightarrow \$V_j A_x^i \qquad (\text{if } i \neq j)}$$

so that the A's can skip over non-matching V's, and $(P^*\text{—continued})$

$$\boxed{A_x^i V_i\$ \rightarrow \$V_i x A_x^i \qquad (\text{all } x \text{ in } A, i = 1, \ldots, n)}$$

which leaves a copy of x to the right of the V_i. We also will need

$$\boxed{\begin{array}{l} Y\$ \rightarrow \$Y \\ Z\$ \rightarrow \$Z \end{array}}$$

to allow strings to rotate around.

Applying these productions to our example string S'' yields a great many routes and paths of generated strings, but there is only one final result. Typical strings in the process, as the A's migrate to the right, are:

$$YA_e^1 A_f^1 A_g^2 A_d^3 A_c^3 aaV_3 bbV_1 aaV_3 cZ$$
$$YA_e^1 A_f^1 A_g^2 aA_d^3 aV_3 cA_c^3 bbV_1 aaV_3 cZ$$
$$YA_e^1 A_f^1 aaV_3 dcbA_g^2 bV_1 aaV_3 dccA_d^3 A_c^3 Z$$
$$YaaV_3 dcbbA_e^1 V_1 faaA_f^1 A_g^2 V_3 dccA_d^3 A_c^3 Z$$

and eventually,

$$YaaV_3 \boxed{dc}\, bbV_1 \boxed{ef}\, aaV_3 \boxed{dc}\, cA_e^1 A_f^1 A_g^2 A_d^3 A_c^3 Z$$

where we have marked where the \$ strings have been copied in. Now our work is done, except for removing the scaffolding—the A's and V's and Y and Z used in the construction. We use a trick based on the following observations: We can eliminate an A when it reaches Z, for then its work is done. We can eliminate a V once all the A's have passed it, for then *its* work is done. We will use the Y to inform the V's that all the A's have passed through. We will not permit Y to pass an A, but once all the A's have passed a V, Y can come up and eliminate the V. Thus the productions

$$\boxed{\begin{array}{lll} Yx\$ \rightarrow \$xY & (x \text{ in } A) & \\ YV_i\$ \rightarrow \$Y & (\text{all } i = 1, \ldots, n) & \\ A_x^i Z\$ \rightarrow \$Z & x \text{ in } A, i + 1, \ldots, n & \end{array}} \qquad (P^*\text{-continued})$$

will destroy the A's on contact with Z and the V's on contact with Y. Only when all the A's and V's are gone can Y come to stand just to the left of Z, and we celebrate this completion of the whole process with the final production of our system:

$$\boxed{YZ\$ \rightarrow \$T} \qquad (P^*\text{-completed})$$

Applying all this to our example, string S results finally in

Taadcbbefaadcc

PROBLEM 13.1-1. Reconstruct this proof using productions $g\$ \rightarrow \$h$ in which neither g nor h have more than two letters.

PROBLEM 13.1-2. Show that only two auxiliary letters are needed in proving theorem 13.1-1. In fact, only one is needed(!) but this is very much harder to prove and requires a different method for proving the theorem.

13.1.3 Completing the proof

There remain a few loose ends in the proof. First, note that we haven't really produced a legitimate extension P^* of P because P^*, so far, does not produce any strings with only lower-case letters. (Proof of this: The axioms of P contain the upper-case letter T. Every production with an upper-case letter in its antecedent also has one in its consequent. So, by induction, every produced string has an upper-case letter.) Now the strings we want to "detach" are those that begin with T, because any

string $T\$$ is an assertion that $\$$ is a theorem of the system P. It would be tempting to introduce the production $T\$ \rightarrow \$$ to simply remove an initial T, but this won't work. The reason is that this would lead to spurious lower-case theorems, namely, rotated versions of legitimate theorems, because the present system already contains the productions $x\$ \rightarrow \$x$. In fact, we have to conclude that there is *no* way, in the present system, to avoid this. The cure: begin all over again with a new alphabet, supposing that $B = b_1, b_2, \ldots, b_r$ is the real alphabet of P and that $A = a_1, a_2, \ldots, a_r$ were really new letters of P^*. Now we can detach the theorems of P by the productions

$$\boxed{\begin{array}{l} Ta_j\$ \rightarrow \$Rb_j \\ a_j\$ \rightarrow \$b_j \\ R\$ \rightarrow \$ \end{array}}$$

which do not allow the b's to rotate. Of course, in this system, a's may be converted to b's prematurely, but this leads only to "doomed strings" and not to spurious pure b strings.

PROBLEM. Prove this.

What if the original system P has more than one production? We simply carry out the whole construction again for each P-production, using entirely new sets of auxiliary letters. Only the key symbol T is common. Then the P-productions operate independently, linked only by the common T that allows any P-production to operate on the *final* results of the operation of other P-productions.

We need one more theorem to complete the proof that the canonical systems of Post, in their full generality, can be replaced by normal extensions. We have to account for those more powerful productions whose antecedent concerns several strings rather than just one. For example, in logic one often has rules of inference like: "From a theorem R and another theorem of the form R *implies* Q we can deduce Q." This says, in the language of productions, that something like

$$R \text{ AND } (R \Rightarrow Q) \rightarrow Q$$

should be a production of the system, where AND is not a string but a way of saying that the production has two separate antecedents.† In the next section we will prove a more general theorem about such systems. The

†There is a serious difficulty in representing logical systems directly as systems of productions, unless one introduces auxiliary symbols for making sure that parenthesis sets are not erroneously broken.[1]

method of proof there is of interest in itself, as we shall see later, because it shows another connection between logistic systems and machines.

13.2 THE NORMAL FORM THEOREM FOR MULTIPLE-ANTECEDENT PRODUCTIONS. REDUCTION TO SINGLE-AXIOM SYSTEM

We now consider a more general form of a Post system, in which a production can combine several strings (theorems) to form a new theorem. In this most general system a typical production has the form

$$\left.\begin{array}{ll} g_{10}\$_{11}g_{11}\$_{12}\ldots\$_{1n_1}g_{1n_1} & \textbf{and} \\ g_{20}\$_{21}g_{21}\$_{22}\ldots\$_{2n_2}g_{2n_2} & \textbf{and} \\ \ldots\ldots\ldots\ldots\ldots\ldots & \\ g_{p0}\$_{p1}g_{p1}\$_{p2}\ldots\$_{pn_p}g_{pn_p} & \end{array}\right\} \rightarrow h_0\$'_1h_1\$'_2\ldots\$'_mh_m \qquad (\pi)$$

where all the g's and h's are given constant strings and all the $\$$'s are variables and where each $\$'_k$ is some one of the $\$_{ij}$'s. The meaning is that, if there are already p theorems that satisfy the p antecedent conditions, then a new theorem is produced according to the consequent form, which may use parts from all the antecedent strings. Post argued that any operation performed by a mathematician could be accounted for by some such production. We want to show that any system of this form can be replaced by a normal canonical extension. We will show, incidentally, that this can be done in such a way that only one axiom is required. As you will observe, however, this refinement is pretty much forced upon us.

THEOREM 13.2-1

Given a Post canonical system P of the general multi-antecedent kind, we can construct a new canonical system P^* that has only single-antecedent productions, has a single axiom, and is a canonical extension of P. (Then by the Theorem of 13.1 we can further construct a *normal* extension P^{**} of P, still with *a single axiom.*)

Let A be the alphabet of P (assumed as before to be lower-case) and let $\Phi_1, \ldots, \Phi_r$ be the axioms of P. The one axiom of the new system P^* will be

$$BB\Phi_1BB\Phi_2BB\ldots BB\Phi_rBB$$

There would be little point in having several axioms now, anyway, since with single-antecedent productions there could be no possibility of their interacting to produce theorems. Corresponding to each production of

the form π belonging to the original system P, we will introduce the following monstrously complicated production:

$$B\$_0^*B\ g_{10}\$_{11}\dots\$_{1n_1}g_{1n_1}\ B\$_1^*B\ \dots\quad\dots B\$_{p-1}^*Bg_{p0}\$_{p1}\dots\$_{pn_p}g_{pn_p}B\$_p^*B$$

$$\longrightarrow$$

$$\boxed{B\$_0^*B\,..\text{(exactly the same as antecedent)}..B\$_p^*B}\ h_0\$_1'h_1\$_2'h_2\dots$$

$$\dots\$_m'h_mX\$_{11}\$_{12}\dots\$_{1n_1}\$_{21}\$_{22}\dots\$_{2n_2}\dots\quad\dots\$_{p1}\$_{p2}\dots\$_{pn_p}X$$

What does this do? The antecedent attempts to analyze a string to see if it contains, sandwiched between B's, substrings that fit the antecedent forms

$$g_{10}\$_{11}\dots\$_{1n_1}g_{1n_1}$$
$$\dots$$
$$g_{p0}\$_{p1}\dots\$_{pn_p}g_{pn_p},$$

that is, to see if the proper ingredients for the production are to be found somewhere among a string of theorems of the form

$$BB\$BB\$BB\dots BB\$BB\$BB$$

If this happens, the proper consequent will be assembled, eventually, and adjoined to the string:

$$BB\$BB\$BB\dots BB\$BB\$BB\$_CBB$$

so that in the future the new theorem $\$_C$ will have the same status as an axiom; that, after all, is what a theorem is.

That is the plan, anyway. Two difficulties arise. The first is that our monstrous production requires the antecedent strings to occur in a given order—an undesirable restriction we will lift shortly. The other difficulty is more serious. We have to check that the analyses of the antecedents are *correct* in that the strings assigned to the $\$_{ij}$'s are *proper* parts of old theorems and axioms. The danger is that a $\$_{ij}$ may contain too much—it may run from the beginning of one P string, through some B's, to the end of a different P string. Now this will be the case if, and only if, a $\$_{ij}$ contains one or more B's in its interior. Our monstrous production is designed to make it easy to test for this contingency. That is why the production appends *the concatenation of all the variable strings,* in the form

$$X\$_{11}\dots\$_{ij}\dots\$_{pn_p}X$$

(Observe also that the consequent does not end in a B; so the monstrous production cannot operate on it again, immediately.) We can now test to see that this appendage contains no B's by using the productions

$$\boxed{\begin{aligned}\$_1Xx\$_2X &\to \$_1X\$_2X \qquad (\text{all } x \text{ in } A)\\ \$XX &\to \$BB\end{aligned}}$$

The result is that the inner X moves to the right, erasing lower-case letters. If it ever encounters a B, the string is doomed. If the inner X gets across to the final X, there were no B's in the appendage, and the system reverts to the axiom form, but with the new theorem appended to the theorem list!

We must now lift the restriction that the antecedent components occur in some fixed order. One method might be to introduce a distinct production system for each permutation of the antecedents—this would mean $p!$ forms for each original production. A more elegant solution is simply to adjoin the single production

$$\boxed{B\$_1BB\$_2BB\$_3B \to B\$_1BB\$_2BB\$_3BB\$_2B}$$

PROBLEM. Why does this eliminate the need for permuted productions?

As usual, we must finish by providing a mechanism by which P strings can be released without any auxiliary letters (namely, B's and X's). To do this, we use the same trick we used in checking for B's, by adding the productions

$$\boxed{\begin{aligned}B\$_1B\$_2B\$_3B &\to Y\$_2Y\\ \$_1Yx\$_2Y &\to \$_1xY\$_2Y \qquad (\text{all } x \text{ in } A)\\ \$YY &\to \$\end{aligned}}$$

which releases any lower-case string that is enclosed by B's.

13.3 A UNIVERSAL CANONICAL SYSTEM

It is possible to construct a canonical system that is a sort of analogue to a universal Turing machine. We will do this by using the methods developed in the previous section. The innovation is that instead of applying the technique to the *theorems* of another system we will apply it to the *productions* of the other system, regarding the productions themselves as strings of symbols. A "universal" system is one which works with the *description* of another system to compute what that other system does. Here, as we shall see, it is so easy to work with descriptions of Post systems that no arithmetic tricks, or the equivalent, are needed.

THEOREM 13.3-1

There exists a certain system U of Post productions with the property: Given any other canonical system Q, then we can construct a single axiom A_Q for U such that the system U with the axiom A_Q is a canonical extension of Q in the following sense. Let $(a_1, \ldots, a_r)$ be the alphabet of Q. Let **a** and **b** be new letters, and encode $(a_1, \ldots, a_r)$ into the alphabet **(a, b)** by representing a_j by j **a**'s followed by a **b**. Then the strings of U, with axiom A_Q, that have only letters **a** and **b** will be exactly this encoding of the theorems of Q.

To prove this, we have to exhibit the productions of U and show how to construct the axiom A_Q.

By the theorems proved earlier in the chapter, the arbitrary system Q has a normal extension P. Suppose that this system P has axioms $\Phi_1, \ldots, \Phi_s$ and productions $g_i\$ \rightarrow \$h_i$. Let $(a_1, \ldots, a_r)$ be the alphabet of P and let A, C, S, and T be new letters. Our system U will be supplied with the axiom

$$A_P = \boxed{ACAg_1Ch_1Ag_2Ch_2A \ldots Ag_nCh_nAS\Phi_1S\Phi_2S \ldots S\Phi_sSTTT}$$

Observe that this axiom is a complete description of P since from it one can reconstruct all the axioms and productions of P.

Next, consider the production

$$\boxed{\begin{array}{l}\$_1A\$_AC\$_CA\$_2S\$_B\$S\$_3TTT \\ \qquad\qquad \rightarrow \\ \$_1A\$_AC\$_CA\$_2S\$_B\$S\$_3\$\$_CST\$_A\$\$_CT\$_B\$\$_CT\$\$_C\end{array}} \qquad (\pi_U)$$

This production looks (in the first, or production, part of a theorem) for a "production" of the form $A\$_AC\$_CA$ that matches a "theorem" of the form $S\$_B\$S$ and attempts to adjoin to the theorem list the string $\$\$_C$ in accord with the "production" $\$_A\$ \rightarrow \$\$_C$. This would be a legitimate thing to do only (1) if it is the case that the strings $\$_B$ and $\$_A$ are *identical* and (2) if none of the strings concerned (namely, \$, $\$_B$, $\$_A$, and $\$_C$) contain any upper-case letters. For in that case we can be sure that $\$\$_C$ is a theorem of the system P and that it is appropriate to add it to the theorem- (or axiom-) list represented by the strings between the S's. The following productions check both these conditions:

$$\boxed{\$_1Tx\$_2Tx\$_3T\$_4 \rightarrow \$_1T\$_2T\$_3T\$_4 \qquad (x \text{ in } (a_1, \ldots, a_r))} \qquad (\pi_i)$$

The trick is this: These productions remove, one at a time, *identical lower-case* letters from $\$_A\$\$_C$ and from $\$_B\$\$_C$. If, and only if, both strings are identical and entirely lower-case, both strings will vanish, leaving a string of the form $\$_1TTT\$_2$. So we adjoin also the production

$$\boxed{\$_1TTT\$_2 \rightarrow \$_1TTT} \qquad (\pi_T)$$

which restores the string to its original form, except with the new theorem $\$\$_C$ inserted in the theorem list. To release the new theorem into the pure lower-case alphabet, we appended an extra copy of it at the end of the intermediate string, and it can be released by the production

$$\boxed{\$_1TTT\$_2 \rightarrow \$_2} \qquad (\pi_R)$$

We put the initial ACA into the main production just so that the axioms themselves could be produced as theorems of the system, just in case the production $\$ \rightarrow \$$ was not included among the productions of P.

The system has the defect that the productions used above depend on the alphabet of P. If we use the binary encoding mentioned in the statement of the theorems, then the system of test productions is replaced by two fixed productions:

$$\$_1T\mathbf{a}\$_2T\mathbf{a}\$_3T\$_4 \rightarrow \$_1T\$_2T\$_3T\$_4 \qquad (\pi_{\mathbf{a}})$$

$$\$_1T\mathbf{b}\$_2T\mathbf{b}\$_3T\$_4 \rightarrow \$_1T\$_2T\$_3T\$_4 \qquad (\pi_{\mathbf{b}})$$

and our whole system U has only five productions: π_U, π_T, π_R, $\pi_{\mathbf{a}}$, and $\pi_{\mathbf{b}}$!

PROBLEM 13.3-1. The production π_T

$$\$_1TTT\$_2 \rightarrow \$_1TTT$$

can be eliminated by making a trivial addition to the antecedent of the main production π_U. What is this change? This reduces the universal system to just four productions. I can see no way to reduce the number of productions further because it would seem that we need one to "do the work," two to check the binary alphabet conditions, and one more to release the pure strings. Such reasoning, however, often turns out to be unsound.

PROBLEM 13.3-2. Construct a version of U that has only one upper-case letter, and only the lower case letters **a** and **b**. Can you do it with still only four productions? The resulting system is probably minimal, in some sense, with three symbols and four productions.

PROBLEM 13.3-3. Construct a version of U with only normal productions. Do not try to minimize the numbers of letters or productions.

PROBLEM 13.3-4. Using the analogy with a universal Turing machine, construct an unsolvable decision problem about the strings produced by U

as a function of the given axiom. Consider the prospect of developing a theory of computability on this basis as compared with the Turing or recursive-function basis.

PROBLEM 13.3-5. We have not quite proved everything claimed in theorem 13.3-1. We have proved it for any normal system P. Now prove it for the original, general system Q. The present system, as described, will release the theorems of P (which include those of Q but also some others). To complete the proof requires a slightly more complicated release system.

PROBLEM 13.3-6. We have never allowed an antecedent with a double occurrence of a $\$_i$. As noted in the remarks of section 12.5, Post allowed this, but we feel that it is not entirely in the spirit of the finite-state approach. In any case, show that for any system with this more general production permitted, there are canonical extensions of our more conservative kind. In particular, begin by showing that we can simulate the effect of the production

$$\$_1 0 \$_1 11 \$ \rightarrow \$_1 0 \$$$

by an extension that does not use double $\$_i$'s in its antecedent. Use the T method, as in section 13.1.[2]

NOTES

1. For example, there is no canonical system for the theorems of the propositional calculus, in its conventional form, that does not use at least one extension letter. Or so I believe, but I have not been able to reconstruct what I think was a proof of this.
2. See note 1 of chapter 12.

14 VERY SIMPLE BASES FOR COMPUTABILITY

14.1 UNIVERSAL PROGRAM MACHINES WITH TWO REGISTERS

In section 11.1 we introduced "program machines" which could compute any recursive function by executing programs, made up of the two operations below, on the contents of registers—number-containing cells.

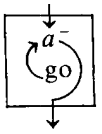
$\boxed{a'}$	Add unity to the number in register **a**, and go to next instruction.
$\boxed{a^-(n)}$	If the number in **a** is not zero, then subtract 1 from **a** and go to the next instruction, otherwise go to the nth instruction.

We showed, in 11.2, that these operations working on just five registers were enough to construct an equivalent of any Turing machine, and we remarked in 11.4 that this could be done with just two registers; we will now prove this. But our purpose is not merely to reduce the concept of program machine to a minimum, but to also use this result to obtain a number of otherwise obscure theorems.

We recall (from 11.4) that the operation $\boxed{a^0}$, i.e., put zero in register **a**, can be simulated if we have a register **w** already containing zero; then we can also use $w^-(n)$ as a $\boxed{\text{go}(n)}$ instruction.

For our purposes here it is more convenient to assume that we have

$\boxed{a'}$, $\boxed{a^-(n)}$, and $\boxed{\text{go}(n)}$ at the start. Then



will serve as a^0, and we can assume that we have

$$a^0 \qquad a^-(n) \qquad a' \qquad \text{go}(n)$$

These are the only operations that appear in the diagram of 11.2, so they are shown to be all we need to simulate any Turing machine.

To reduce the machine of Fig. 11.2-1 to one with just two registers **r** and **s**, we will "simulate" a larger number of virtual registers by using an elementary fact about arithmetic—namely that the prime-factorization of an integer is unique. Suppose, for example, that we want to simulate *three* registers **x**, **y**, and **z**. We will begin by placing the number $2^x3^y5^z$ in register **r** and zero in register **s**. The important thing is that from the single number $2^x3^y5^z$ one can recover the numbers x, y, and z simply by determining how many times the number can be divided by 2, 3, and 5 respectively. For example, if $r = 1440$, then $x = 5$, $y = 2$, and $z = 1$. For our purposes, we have only to show how we can obtain the effect of the operations x' and x^-, y' and y^-, and z' and z^-.

INCREMENTING

Suppose we want to increment **x**, that is, add unity to x. This means that we want to replace $2^x3^y5^z$ by $2^{x+1}3^y5^z = 2 \cdot 2^x3^y5^z$. But this is the same as doubling the number in **r**! Similarly, incrementing **y** and **z** is trebling and quintupling (respectively) the contents of **r**. And this is done by the programs of Fig. 14.1-1. The first loop in each program counts **r** down while counting **s** up twice (or three or five times) as fast; the second loop transfers the contents of **s** back into **r**.

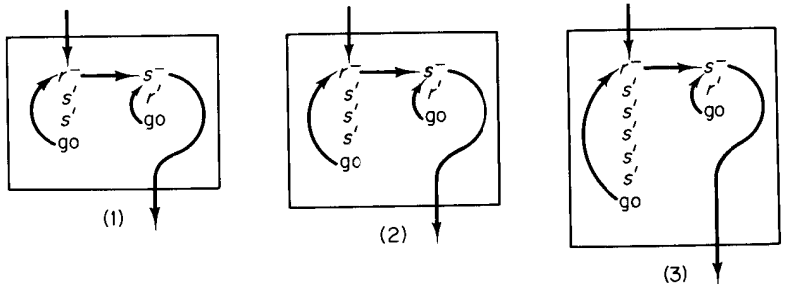


Fig. 14.1-1

DECREMENTING

Subtracting unity from x is a little more tricky, since we have to determine whether x is zero; if x is zero, we want to leave it unchanged and do a $\boxed{\text{go}}$. If x is not zero, then we want to change $2^x3^y5^z$ into $2^{x-1}3^y5^z$—that is, divide the contents of **r** by two. Similarly, decrementing y and z is (conditional on not being zero) equivalent to dividing by three and five. We illustrate in Fig. 14.1-2 a program for the 5-case.

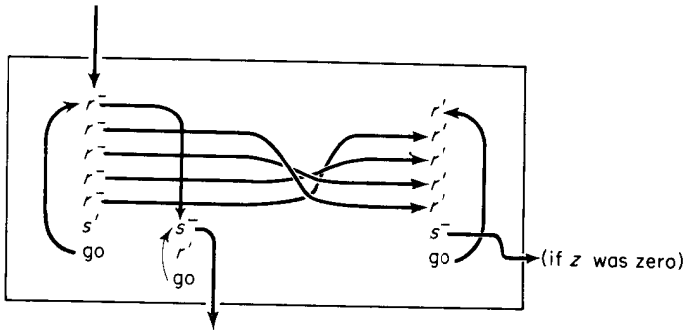


Fig. 14.1-2

The loop to the left does the division, by repeated subtraction. If the division comes out exact—that is, has no remainder—then the lower loop copies the quotient back into **r**. If the division was inexact (i.e., if z was zero), the loop to the right first restores the remainder (which at that moment is stored in the state of the machine—i.e., the location in the program) and then multiplies the quotient by the divisor, putting the result (which is the original contents) back into **r**.

This is all we have to show, for clearly we can do the same for the *five* registers mentioned in 11.2. We simply put in **r** the number $2^m3^n5^a7^z11^w$ and use the same techniques given just above. To build a program machine equivalent to that of Fig. 11.2-1, we take that diagram and replace *each* of its individual instructions by a copy of the equivalent program structure we have just developed. This proves:

THEOREM 14.1-1

For any Turing machine T there exists a program machine M_T with just two registers that behaves the same as T (in the sense described in sections 10.1 *and* 11.2*) when started with zero in one register and $2^a3^m5^n$ in the other. This machine uses only the operations* $\boxed{'}$ *and* $\boxed{-}$.

REMARK

Now that we know there are universal program machines that use only *two* registers, we can improve the result of section 12.8. For such a machine there are just four kinds of instructions: "add 1 to R_1," "add 1 to R_2," "subtract (conditional) 1 from R_1," "subtract (conditional) 1 from R_2." In such a machine, we can omit the R letters and just use the I's to separate the two registers, so that we can have a canonical system for such a machine with simple productions like

$$\$_1 I_j \$_2 \rightarrow \$_1 1 I_{j+1} \$_2$$

for addition, and

$$\$_1 I_j 1 \$_2 \rightarrow \$_1 I_{j+1} \$_2$$
$$\$ I_j \rightarrow \$ I_{j'}$$

for the conditional subtraction.

14.2 UNIVERSAL PROGRAM-MACHINE WITH ONE REGISTER

We can get an even stronger result if we admit multiplication and division operations (by constants) as possible machine instructions, for then we can do everything with a single register! The proof of theorem 14.1-1 shows that we need only a single register if we can perform on it the operations "multiply by 2 (or 3, 5, 7, 11)" and "divide by 2 (or 3, 5, 7, 11) conditionally upon whether the division is exact." Now one last observation. The proof of theorem 14.1-1 shows that the effects of multiplication by 2 (or by 3) and division by 2 (or by 3) on a number of the form 2^x3^y are precisely the same as the effects of incrementing **x** (or **y**) and decrementing **x** (or **y**). Furthermore, by theorem 14.1-1, we don't really need anything but these four operations, if we are willing to raise our numbers up to another exponent level. Thus, by applying the result of theorem 14.1-1 to its own proof (!) we obtain:

THEOREM 14.2-1

For any Turing machine T there exists a program machine M_T^* with just one register that behaves the same as T (in the usual sense) when started with $2^{2^a 3^m 5^n} \cdot 3$ in its register. This machine uses only the operations of multiplication and (conditional) division by integers 2 and 3.

PROBLEM 14.2-1. Verify the correctness of this fast-talking proof for Theorem 14.2-1.

PROBLEM 14.2-2. In both theorems 14.1-1 and 14.2-1, the final result was a program machine with *four* instruction types. In one case there were two operations that could each be applied to each of two registers; in the other case, there were *four* operations on one register. Show that in each case we can reduce the number to three, by replacing two of them with one that affects both numbers or registers. For example, in the machine M_T we can eliminate s^- and s' if we keep r^- and r' and adjoin *exchange* (r, s). In M_T^* we can use *multiply by* 6, *divide by* 3, and *divide by* 2. We can even reduce the basis to two instructions: Show that for machine M_T it is sufficient to have:

Add 1 to r and exchange r and s.
If r is not zero, subtract 1 from r and exchange;
otherwise exchange and go(n).

A slight change in input form is required.

DISCUSSION

Theorem 14.2-1 looks, superficially, like a better result than theorem 14.1-1, since one register is less than two. However, from the point of view of Turing's argument (chapter 5), we prefer theorem 14.1-1 because we can think of m', m^-, n', and n^- as unitary, basic, finite actions. On the other hand, the operation of multiplying the contents of a register by two cannot be regarded as a fixed, finite action, because the amount of work involved must grow with the size of the number in the register, beyond any fixed bound. (The same objection could be held against theorem 14.1-1 if the number in the register is represented as a binary string. It must be unary. Why?)

14.3 GÖDEL NUMBERS

The methods of section 14.1 draw their surprising power from the basic fact that one can represent any amount of information by selecting a single integer—or, more precisely, that one can effectively compute such an integer and effectively recover the information from it. Thus, in the proof of theorem 14.1-1 we take an arbitrary quadruple (m, n, a, z) of integers and encode them into the single integer $2^m3^n5^a7^z$. There is nothing new here, for in our original arithmetization of Turing machines (section 10.1) we use the numbers m and n themselves to represent arbitrary sequences of 1's and 0's, and in our discussion (10.3) of enumeration of the partial-recursive functions we envisioned an even more complex encoding of information into single integers. In 12.3 we showed that a single integer could represent an arbitrary list of lists of letters from an alphabet. More generally, as was apparently first pointed out by Gödel [1931], a single integer representation can represent an arbitrary list structure (see also 10.7). Let us define such a correspondence, inductively:

If K is a number, then $K^* = 2^K$
If K is a list $(a, b, c, d, \ldots)$, then $K^* = 3^{a^*}5^{b^*}7^{c^*}11^{d^*}\ldots$

For example,

$$3^* = 2^3$$

$$(2, 4, 6)^* = 3^{2^2} \cdot 5^{2^4} \cdot 7^{2^6}$$

$$(1, (2, 3), 4)^* = 3^{2^1} \cdot 5^{3^{2^2} \cdot 5^{2^3}} \cdot 7^{2^4}$$

$$(0, (0, 0))^* = 3 \cdot 5^{3 \cdot 5}$$

PROBLEM 14.3-1. Show that any two different list structures of integers yield different *-numbers, and describe an effective procedure to recover the list structure from the *-number.

PROBLEM 14.3-2. Which of the following definitions have the property that the list structure can be unambiguously recovered from the number?

(1)

(1) $\begin{cases} K^* = K & \text{(if } K \text{ is a number)} \\ K^* = 2^{a^*}3^{b^*}5^{c^*}7^{d^*}\ldots & \text{(if } K = (a, b, c, d, \ldots)) \end{cases}$

(2) $\begin{cases} K^* = 2K & (K \text{ a number}) \\ K^* = 3^{a^*}5^{b^*}7^{c^*}11^{d^*}\ldots & (K \text{ a list}) \end{cases}$

(3) $\begin{cases} K^* = 3^K & (K \text{ a number}) \\ K^* = 3^{2^{a^*}}5^{2^{b^*}}7^{2^{c^*}}11^{2^{d^*}}\ldots & (K \text{ a list}) \end{cases}$

The ideal of a Gödel numbering was important in mathematical logic because while that subject is, on the surface, concerned with notions of number and the foundations of mathematics, it has come more and more to be a theory of *mathematical theories*. It is possible to use the number-theoretic formalisms, as Gödel showed, to talk about the sentences of the theory (and sentences are not numbers) by assigning numbers to sentences and properly interpreting corresponding numbers. In particular this makes it possible to interpret a theory as talking about its own sentences, without paradoxes; and Gödel proved his celebrated theorem, about the impossibility of a non-contradictory theory containing a proof of its self-consistency, using this technique. More recently it has been noted that one can use numbering schemes much simpler than Gödel's for the same effect, avoiding number-theory facts like the unique-factorization theorem. The methods of Smullyan [1961] are particularly elegant; Smullyan was strongly influenced by Post's inclination to deal directly with symbolic expressions rather than numerical representations of them. (So was I, for example, in the proof of theorem 13.3-1). But I believe methods like those of section 10.7, or those of 13.3, that step around arithmetic entirely, are ultimately the clearest and most illuminating.

14.4 TWO-TAPE NON-WRITING TURING MACHINES

It is easy enough to generalize the idea of a Turing machine to a machine with two or more tapes. One way to do this is to specify, with each state of the Turing machine, what it is to do with each tape (what to write, and which way to move) for each *combination* of symbols seen at the set of reading heads. In this formulation one would use, instead of the quintuple $(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$ of the ordinary machine, a specification like the following—for K tapes, a $2K + 3$-tuple:

$$(q_i; s_{i_1 \cdots i_K}; q_{i,i_1 \cdots i_K}; s_{i_{11} \cdots i_{1K}}, \ldots, s_{i_{K1} \ldots i_{KK}}; D_{i_{11} \ldots i_{1K}}, \ldots, D_{i_{K1} \cdots i_{KK}})$$

Another, more convenient, way to describe a multi-tape machine would be in terms of a program machine whose instructions include:

Move tape j in direction d.
Write symbol s_i on tape j.
If head j reads s_i, go to instruction ....

As we know, it is very hard to construct Turing machines to do tasks of any significant complexity, even for theoretical purposes, and no one would consider using one for a practical purpose. It is much easier to sketch out the organization of a multi-tape Turing machine for a complex computation if one simply assigns different tapes to different memory functions; then it is not necessary to resort to tricky punctuation devices. For example, it is very easy to describe a universal Turing machine using two tapes—one for simulating the tape of, and one for holding the description of, the (one-tape) machine being imitated.

PROBLEM. Show how to construct a simple universal machine using two tapes.

It should come as no surprise that, in spite of the greater apparent power of multi-tape machines, their ultimate computation range is the same as usual, namely, the computation of any partial-recursive function. This can be shown by describing how to make a conventional, one-tape, Turing machine that imitates K-tape machines.

In our original, heavy-handed construction of a universal Turing machine (section 7.3), we reserved two regions of a machine's tape for different purposes. One of these—the description region—was finite, but it could just as well have been infinite. The same method won't work for more than two tapes, of course, since a tape has only two directions; but one can get the effect of K tapes simply by assigning every Kth square of the tape to be used for imitating a given tape. We leave the construction as an exercise.

PROBLEM. Show that any K-tape machine can be simulated by a one-tape machine.

Although K-tape machines, in general, are equivalent to one-tape machines, this conceivably might not be so if we impose restrictions on what can be done with the tapes. It comes, therefore, as something of a surprise that:

THEOREM 14.4-1

Any computation that can be done by a Turing machine can be simulated by a machine with two semi-infinite (single-ended) tapes which can neither read nor write on its tapes, but can only sense when a tape has come to its end.

In view of theorem 14.1-1, the proof is hardly worth writing down. We use the *length* from the reading heads to the ends of the two tapes as our representation of m and n; the operations m' and n' move the heads away from the ends, and m^- and n^- move the heads towards the ends, conditional on reaching an end of tape.

Another consequence of the same situation is this corollary:

COROLLARY 14.4-1

Any Turing-machine computation can be simulated by a Turing machine whose tape is always entirely blank, save for at most three 1*'s.*

For one can construct a Turing machine, equivalent in the (m, n) sense to the given machine, which works with a tape of the form

$$0\ 0\ 0 \ldots 0\ 1\ 0 \ldots (m \text{ zeros}) \ldots 0\ 1\ 0 \ldots (n \text{ zeros}) \ldots 0\ 1\ 0$$

PROBLEM. Show how to construct the machine for the proof of the corollary.

14.5 UNIVERSAL NON-ERASING TURING MACHINES

We can now demonstrate the remarkable fact, first shown by Wang [1957], that for any Turing machine T there is an equivalent Turing machine T_N that *never changes a once-written symbol*! In fact, we will construct a two-symbol machine T_N that can only change blank squares on its tape to 1's but can not change a 1 back to a blank.

Our proof will have two parts. First we will show, given T, how to make an equivalent machine T'_N that uses four symbols 0, A, B, C and is subject to the *symbol-changing restriction* that the only changes permitted are

$$0 \to A, 0 \to B, 0 \to C, A \to B, A \to C, B \to C$$

Then we will show how to make the non-erasing machine T_N out of this T'_N by making the two-symbol T_N work with its tape-squares grouped into binary triplets, so that we can make the identification:

T'_N symbols		T_N symbols
0	$\leftrightarrow$	0 0 0
A	$\leftrightarrow$	1 0 0
B	$\leftrightarrow$	1 1 0
C	$\leftrightarrow$	1 1 1

The point is that the permitted symbol-changes in T'_N correspond to changes for T_N that only transform 0's to 1's; no 1's need ever be changed to 0's.

THEOREM 14.5-1

For any Turing machine T there is an equivalent two-symbol machine which can only print 1*'s on blank squares but can never erase a* 1*, once printed.*

Proof: First we construct the machine T'_N mentioned above. T'_N will work with our standard two-integer (m, n) representation for the content of a Turing-machine tape, and we will represent the state of T's tape of the form

$$\ldots CCCC \ldots CCCBB \ldots BBAA \ldots AA000 \ldots 000 \ldots$$

where there are m B's, n A's, and any number of C's to the left. We have only to show that we can make state diagrams that will perform the basic

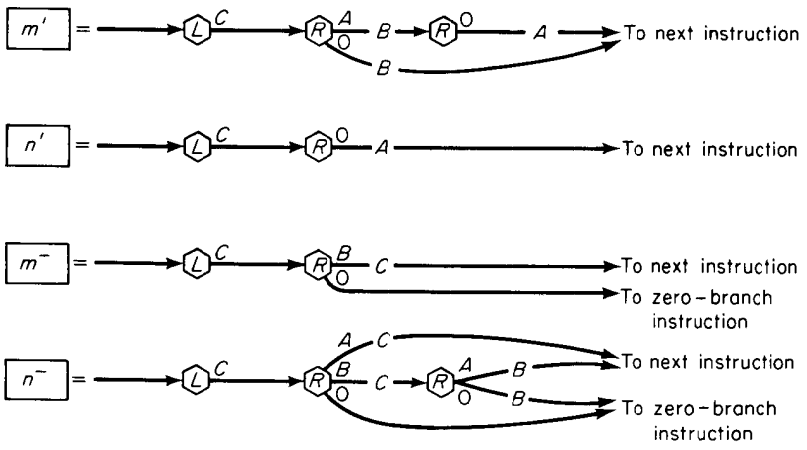
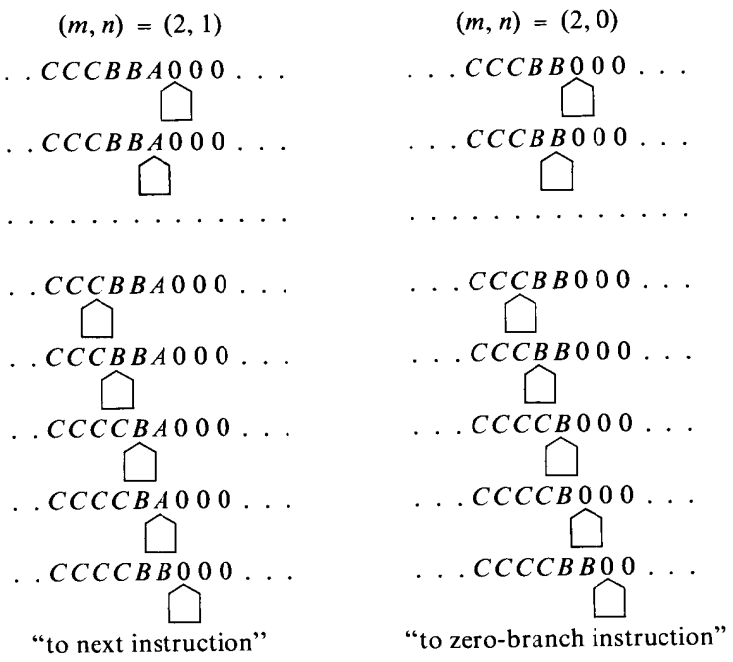


Fig. 14.5-1

operations of theorem 14.1-1, namely, m', n', m^-, and n^-; the latter being conditional on whether m and n are already zero. That is, we have to show that we can increment and (conditionally) decrement m and n—the numbers of B's and A's on T'_N-s tape, without violating the symbol-changing restrictions. Now consider the four state diagrams in Fig. 14.5-1. These do exactly what is wanted. For example, apply the state-diagram for n^- to the strings for $(m, n) = (2, 1)$ and $(2, 0)$, starting somewhere to the right in each case:

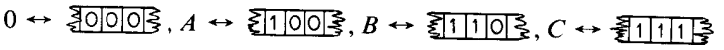


In each case the machine first runs to the left until it encounters the block of C's. (It will never move left of the rightmost C.) It then starts back to the right to perform the desired operation. Only operation n^- has any subtlety at all; the machine is supposed to "erase" an A (if there is one); it can only do this by changing that A to a higher letter—B or C. It prepares for this by removing the leftmost B (if there is one); when it meets an A, it will change this to a B, thus reducing the number of A's and restoring the number of B's. The reader can verify that we have also accounted for the exceptional case in which there is no B; finally, in the case that there is no A (that is, if $n = 0$), B is unchanged but the machine takes a different exit from the state-diagram. Note also that in each case

the machine ends somewhere to the right of the block of C's, so that we can safely link the output of one diagram to the input of another.

To obtain the machine T'_N we now simply find a program machine equivalent to T by using theorem 14.1-1. Then we take each instruction of the program machine and replace it by the appropriate one of the four state-diagram machines just exhibited. Last, we realize the "jump" or "control" structure of the program machine by making corresponding connections of the inputs and outputs of our set of little state diagrams. Since none of the state diagrams violate the symbol-changing restriction, we have constructed T'_M as required.

To complete the proof of theorem 14.5-1, we have only to show how to convert T'_M into a two-symbol non-erasing machine T_M. To do this, we have to provide the mechanism, mentioned earlier, through which T_M will work with its tape as though it were formed of triplets of squares. We will do this by replacing *each state* of T'_M by a state network that operates on triplets of squares in accord with



For example, the state of  , which means "move left (by triplets) until encountering the triplet 1 1 1, is realized in T'_N by the network of Fig. 14.5-2, where the loop-structure is used to make sure that the system always moves three squares. The right-moving states of T'_M are replaced by similar structures; for example, the right side of Fig. 14.5-3 would replace the left side, and the right side of Fig. 14.5-4 would replace the left side. The latter example shows again how a searching state keeps moving along by repeating triplets until it finds one of the symbols it is searching for.

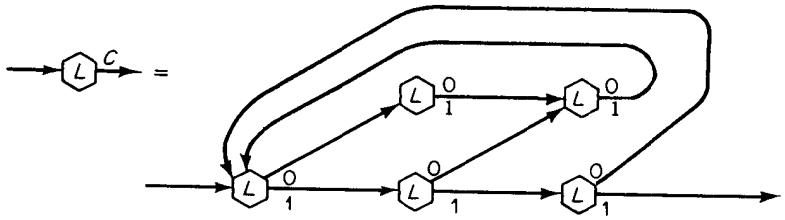


Fig. 14.5-2

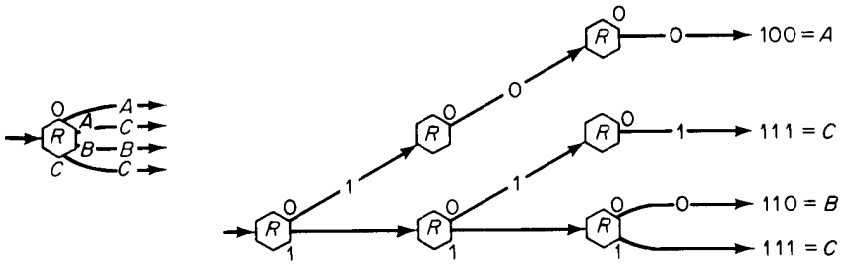


Fig. 14.5-3

PROBLEM. The machine T_M just described wipes out all prior records of its computation, by changing all old symbols to C's. It is possible to make a machine similar in operation except that a permanent record of all prior steps in the computation (that is, all steps of the original machine T, being simulated) is retained. Construct such a machine T_M^* with the property that, if (m_t, n_t) are the (m, n)-representation of T's tape at time t, then T_M^*'s tape at time t has the form $m_1n_1m_2n_2 \ldots m_tn_t \ldots$ in some straightforward sense. For example, one can use a five-level "non-erase" encoding

$$\ldots EEE \ldots EEED^{m_1}C^{n_1}D^{m_2} \ldots D^{m_{t-1}}C^{n_{t-1}}B^{m_t}A^{n_t}000 \ldots 000 \ldots$$

First make a state diagram that will "copy" m_tn_t by converting the above string into

$$\ldots EEE \ldots EEED^{m_1}C^{n_1}D^{m_2} \ldots D^{m_{t-1}}C^{n_{t-1}}D^{m_t}C^{n_t}B^{m_t}A^{n_t}000 \ldots$$

and then show how to modify the "copy" structure to realize each of the operations m', n', m^-, n^-, without violating the symbol-changing restrictions. Compare this system with the universal Post system of theorem 13.3-1, which also keeps a record of all previous work.

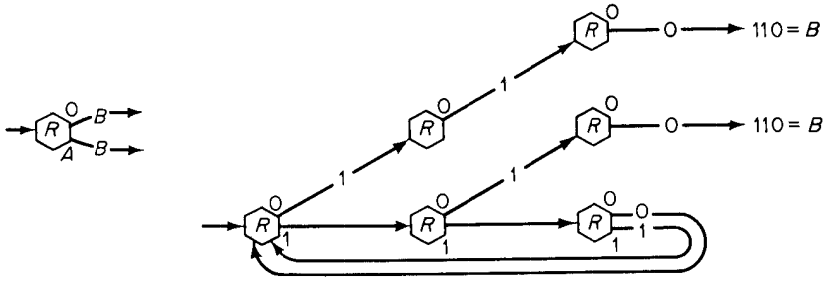


Fig. 14.5-4

14.6 THE PROBLEM OF "TAG" AND MONOGENIC CANONICAL SYSTEMS

While a graduate student at Princeton in 1921, Post [1965] studied a class of apparently simple but curiously frustrating problems of which the following is an example:

Given a finite string S of 0's and 1's, examine the first letter of S. If it is 0, delete the first three letters of S and append 00 to the result. If the first letter is 1, delete the first three letters and append 1101. Perform the same operation on the resulting string, and repeat the process so long as the resulting string has three or more letters.

For example, if the initial string is 10010, we get

10010
101101
1011101
11011101
111011101
0111011101
10111010**0**
1101001101
100110111101
1101110111101
11101110111101
011101110111101
10111011101**0**0
1101110100**1**101
11101001101**1**101
0100110111011101
repeats { 011011101110100
0111011101000**0**
10111010000**0**0
1101000000**1**101
100000011011101
0000110111011101
011011101110100 }

The string has grown, but it has just repeated itself (and hence will continue to repeat the last six iterations forever). Suppose that we start with a different string S'. The reader might try, for example, $(100)^7$, that is, 10010010010010010010, but he will almost certainly give up without answering the question: "Does this string, too, become repetitive?" In fact, the answer to the more general question "Is there an effective way to decide, for any string S, whether this process will ever repeat when started with S?" is still unknown. Post found this (00, 1101) problem "intractable," and so did I, even with the help of a computer. Of course, unless one has a theory, one cannot expect much help from a computer

(unless *it* has a theory) except for clerical aid in studying examples; but if the reader tries to study the behavior of 100100100100100100 without such aid, he will be sorry.

Post mentions the (00, 1101) problem, in passing, in his [1943] paper—the one that announces the normal-form theorem—and says that "the little progress made in the solution . . . of such problems make them candidates for unsolvability." As it turns out he was right. While the solvability of the (00, 1101) problem is still unsettled (some partial results are discussed by Watanabe [1963]), it is now known that *some* problems of the same general character are unsolvable. Even more interesting is the fact that there are systems of this class that are universal in the sense of theorem 14.1-1; namely, there is a way to simulate an arbitrary Turing-machine computation within a "tag" system.

DEFINITION

A tag system is a Post normal canonical system that satisfies the conditions: If $A = (a_1, \ldots, a_n)$ is the alphabet of the system, and

$$g_i\$ \rightarrow \$h_i \qquad (i = i, \ldots, n)$$

are its productions, then

(1) All the antecedent constant strings g_i have the same length P.
(2) The consequent string h_i depends *only on the first letter* of the associated g_i.

For example, in the (00, 1101) problem just mentioned, there are really eight productions, forming the system with $P = 3$:

000\$ → \$00	100\$ → \$1101
001\$ → \$00	101\$ → \$1101
010\$ → \$00	110\$ → \$1101
011\$ → \$00	111\$ → \$1101

Because of the fact that the consequent h_i is determined by the first letter only of g_i, and that the number of letters in the g's is a constant P, the tag systems all have the character of the (00, 1101) problem; namely, to operate a tag system, one has to:

Read the first letter a_i.
Erase P letters from the front of the string.
Append the associated consequent string h_i to the end of the string.

It is very important to observe that the very definition of a tag system gives it a property not found, generally, in normal or other canonical systems; namely, a tag system is *monogenic*.

DEFINITION

A Post canonical system (or any other logical string-manipulation system) is *monogenic* if, for any string S, there is at most one new string S' that can be produced from it (in one step).

Clearly a tag system is monogenic since, for any string, what happens to it depends only on its first letter; two different strings can be produced only if a string has two different first letters, which would be absurd. What is the importance of the monogenic property? It is that, if a string-manipulation system is monogenic, then it is like a machine in all important respects, for it defines a definite *process* or sequence of things that happen—and these can be regarded as happening in real time, rather than as mere theorems about an unchanging mathematical world or space.

In fact, one can imagine a special machine associated with a tag system. (See Fig. 14.6-1.) This machine is a little like a Turing machine except that

(1) There are two heads, one for reading and one for writing.
(2) The tape begins at a source, runs through the writing head, has an arbitrarily long piece of "slack,"

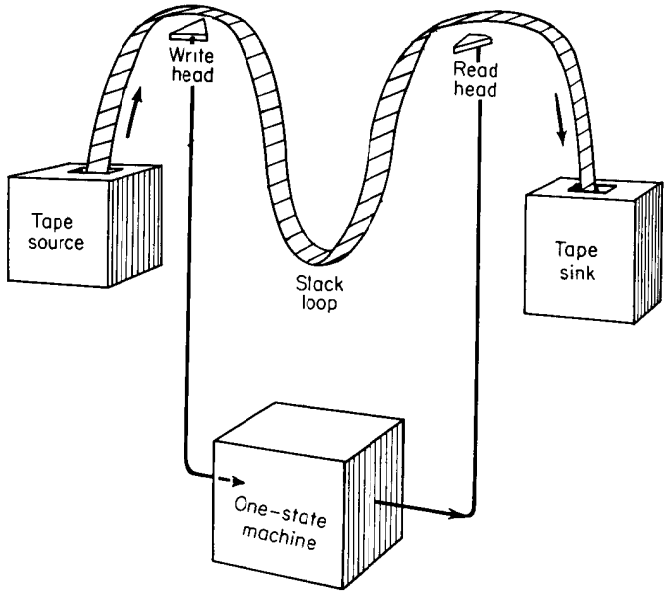


Fig. 14.6-1

then runs through the reading head, and finally disappears forever into a "sink." *It can move only in one direction.*

(3) The finite-state part of the machine must be able to read a symbol, advance the tape P squares, and write the appropriate h_i with the write head. *But otherwise, the machine has no internal states*!

Note that the tag machine cannot erase a symbol. It would do it no good to erase a symbol, since *it can only read a symbol at most once*!

The name "tag" comes from the children's game—Post was interested in the decidability of the question: Does the reading head, which is advancing at the constant rate of P squares per unit time, ever catch up with the write head, which is advancing irregularly. (We do not require the h_i's to have the same lengths.) Note, in the (00, 1101) problem, that the read head advances three units at each step, while the write head advances by two or four units. Statistically, one can see, the latter has the same average speed as the former. Therefore, one would expect the string to vanish, or become periodic. One would suppose this for most initial strings, because if the chances are equal of getting longer or shorter, then it is almost certain to get short, from time to time. Each time the string gets short, there is a significant chance of repeating a previously written string, and repeating once means repeating forever, in a monogenic process. Is there an initial string that grows forever, in spite of this statistical obstacle? No one knows. All the strings I have studied (by computer) either became periodic or vanished, but some only after many millions of iterations!

THEOREM 14.6-1 (Cocke [1964])

For any Turing machine T there exists a tag system T_T that behaves like T, in the (m, n) sense of 11.2, *when given an axiom that encodes T's tape as Aa aa aa...aa Bb bb bb...bb with m aa's and n bb's. The tag system T_T* has deletion number $P = 2$.

COROLLARY 14.6-1

Computability with monogenic normal systems is equivalent to computability with general-recursive functions, Turing machines, general canonical systems, etc.

Proof: We will construct separate tag systems for each of the states of the machine T. Then we will link these together (by identifying certain letters of the different alphabets) to form a single tag system that behaves like the whole machine T. We will begin by constructing a tag system that behaves like the right-hand side of Fig. 11.2-1.

We have been accustomed to thinking of a Turing machine as operating according to the scheme of Fig. 14.6-2. It is equivalent, more con-

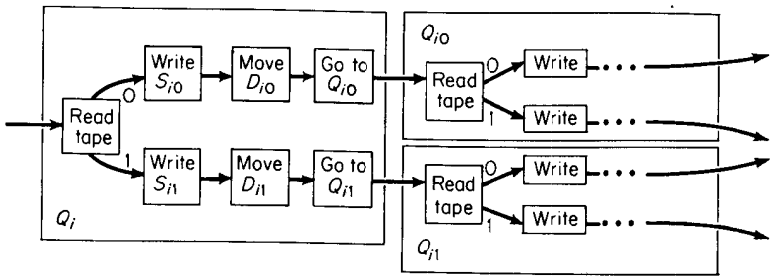


Fig. 14.6-2

venient here, and actually fundamentally simpler to think of the Turing machine as made up of states according to the scheme of Fig. 14.6-3. Looked at this way, a Turing machine will have twice as many states, but in some sense these were concealed in it already, for it must have had a secret pair of states to remember what it had read while it was writing something else.

PROBLEM. Obviously states in the new system do not correspond exactly to states in the old system. The new states are also quintuples, but of the form:

(State	Write	Move	Read:	if 0 go to	if 1 go to)
Q_i	S_i	D_i		Q_{i0}	Q_{i1}

and there is only one quintuple for each state, rather than two. Use this formulation of Turing machines to simplify the development of section 11.3.

Now to realize such a state by a tag system, we will exhibit a set of productions that will have the effect of a *move-right* state. In such a case, we will want to change m and n so that

$$m \to 2m + S_i$$

$$n \to H(n) = \begin{cases} n/2 & \text{if } n \text{ is even,} \\ (n-1)/2 & \text{if } n \text{ is odd.} \end{cases}$$

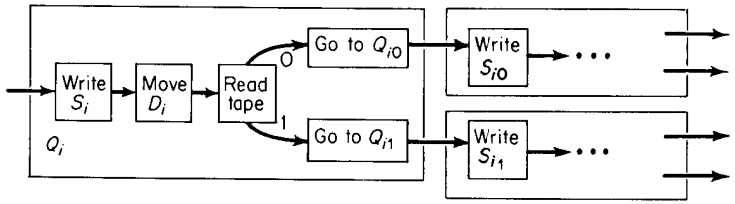


Fig. 14.6-3

and we must prepare the system so that it will next go to state Q_{i0} if n is even, and to Q_{i1} if n is odd.

Let us begin with the string

$$Aa(aa)^m Bb(bb)^n$$

where $@^n$ means to repeat the string $@$ n times. Our first productions will be

$$\boxed{\begin{aligned} A &\to Cc \\ a &\to cccc \end{aligned}} \quad \text{or} \quad \boxed{\begin{aligned} A &\to Cccc \\ a &\to cccc \end{aligned}}$$

depending on whether S_i is supposed to be 0 or 1. (This depends only on what state we are in.) *For brevity, we will abbreviate tag productions by showing only the antecedent first letter and the consequent string*, since writing the $a_i x\ \$ \to \$h_i$ form is so redundant. In every case P, the number of letters deleted, is 2. Applying the productions above leads to $Bb(bb)^n Cc(cc)m'$ where m' is either $2m$ or $2m + 1$ depending on whether or not the state required the machine to write 1 or 0 on the square it is leaving. The key problem is now to determine whether n is even or odd. The procedure for doing this is initiated by the productions

$$\boxed{\begin{aligned} B &\to S \\ b &\to s \end{aligned}} \quad \text{i.e.,} \quad \begin{aligned} Bx\$ &\to \$S \\ bx\$ &\to \$s \end{aligned}$$

which result in the string

$$Cc(cc)^{m'} Ss^n$$

and

$$\boxed{\begin{aligned} C &\to D_1 D_0 \\ c &\to d_1 d_0 \end{aligned}}$$

which lead to

$$Ss^n D_1 D_0 (d_1 d_0)^{m'}$$

Now the oddness or evenness of n is, at last, to have an effect; for

$$\boxed{\begin{aligned} S &\to T_1 T_0 \\ s &\to t_1 t_0 \end{aligned}}$$

yields either

$$D_1 D_0 (d_1 d_0)^{m'} T_1 T_0 (t_1 t_0)^{n-1/2} \quad \text{or} \quad D_0 (d_1 d_0)^{m'} T_1 T_0 (t_1 t_0)^{n/2}$$

depending on whether n was odd or even, respectively. This has a profound effect, because in the first case the system will see only subscript '1' symbols from now on, while in the other case it will see only subscript '0' symbols! Hence, we can control what happens now with two distinct sets of productions:

$$\boxed{\begin{aligned} &n \text{ odd} \\ D_1 &\to A_1 a_1 \\ d_1 &\to a_1 a_1 \end{aligned}} \qquad \boxed{\begin{aligned} &n \text{ even} \\ D_0 &\to a_0 A_0 a_0 \\ d_0 &\to a_0 a_0 \end{aligned}}$$

which yields

$$T_1 T_0 (t_1 t_0)^{n'} A_1 a_1 (a_1 a_1)^{m'} \quad \text{or} \quad T_0 (t_1 t_0)^{n'} a_0 A_0 a_0 (a_0 a_0)^{m'}$$

where we have written n' for $(n - 1)/2$ or $n/2$ as the case may be. Finally,

$$\boxed{\begin{aligned} &n \text{ odd} \\ T_1 &\to B_1 b_1 \\ t_1 &\to b_1 b_1 \end{aligned}} \qquad \boxed{\begin{aligned} &n \text{ even} \\ T_0 &\to B_0 b_0 \\ t_0 &\to b_0 b_0 \end{aligned}}$$

produce the desired final strings

$$A_1 a_1 (a_1 a_1)^{m'} B_1 b_1 (b_1 b_1)^{n'=(n-1)/2} \quad \text{or} \quad A_0 a_0 (a_0 a_0)^{m'} B_0 b_0 (b_0 b_0)^{n'=n/2}$$

The important thing about these two possible final results is that *they are in entirely separate alphabets.* This means that we can now write different productions to determine what will next become of the string in the case that n was even and in the case that n was odd. Thus we are, in effect, able to lay out the structure of a program. What should we do, in fact? We write a set of productions like the ones above for each state Q_i of the Turing machine, using entirely different alphabets for each. Then we link them; whenever an exit state Q_{i1} is the state Q_j, we make the output letters A_1, a_1, B_1, and b_1 of Q_i the same as the input letters A, a, B, and b of Q_j. Similarly we identify the output letters A_0, a_0, B_0 and b_0 of Q_{i0} with the input letters of whatever state is Q_{i0}. Thus, we can simulate the interconnections of states of an arbitrary Turing machine by combining in one large tag system, all tag productions described above. This completes the proof of theorem 14.6-1.

14.7 UNSOLVABILITY OF POST'S "CORRESPONDENCE PROBLEM"

In 1947 Post showed that there is no effective procedure to answer questions of the following kind:

THE CORRESPONDENCE PROBLEM

Given an alphabet A and a finite set of pairs of words (g_i, h_i) in the alphabet A, is there a sequence $i_1 i_2 \ldots i_N$ of selections such that the

strings

$$g_{i_1}g_{i_2}\dots g_{i_N} \quad \text{and} \quad h_{i_1}h_{i_2}\dots h_{i_N}$$

formed by concatenating—writing down in order—corresponding g's and h's are identical?

While Post's original proof of the unsolvability of such problems is complicated, the result of 14.6—that monogenic normal systems can be universal—makes it very simple to prove; for we can show that any procedure that could effectively answer all correspondence questions co qually well be used to tell whether any tag, or other monogenic norm. system will ever reach a halting symbol, and this is equivalent to telling whether any Turing machine computation will halt.

Proof: Let M be a monogenic normal system with axiom A and productions $g_i\$ \rightarrow \$h_i$. Now suppose that this system happens to terminate by eventually producing a string Z which does not begin with any of the g_i's. Define G_i and H_i as follows:

$$\text{if } g_i = a_p a_q \dots a_t, \text{ let } G_i \text{ be } \boxed{Xa_pXa_qX\dots Xa_t}$$

and

$$\text{if } h_j = a_u a_v \dots a_z, \text{ let } H_i \text{ be } \boxed{a_u Xa_v X\dots a_z X}$$

Now consider the correspondence system:

$$\begin{array}{ccc} G_0 & G_i & \overline{Z}XY \\ \updownarrow & \updownarrow & \updownarrow \\ X\overline{A}H_0 & H_i & Y \end{array}$$

where X's are placed *after* each letter of A and *before* each letter of Z to form $\overline{A}$ and $\overline{Z}$. Y is a new letter.

ASSERTION

This system will have a matching pair of identical strings if and only if the monogenic normal system ($g_i\$ \rightarrow \$h_i$), *when started with A, terminates with the string Z.* In fact, if there is any solution to the correspondence question, there is just one, and that solution is (for the G's) the sequence of antecedents and (for the H's) the sequence of consequents encountered in producing Z from A.

If we can establish the truth of the assertion, then the unsolvability of the general correspondence follows, for one can adapt any Turing-machine halting problem to a question of whether the machine in question reaches a certain special state with a blank tape; then, in the normal system, we can reduce this to the question of terminating in a particular

string Z. (Or, we can simply equate this to the unsolvable halting problem for tag systems.)

Why is the assertion true? Let us first note what the X's and Y's are for. The X's are to make sure that *if a matching pair exists at all, it must begin with the transcription* $X\overline{A}$ *of the axiom A.* This is assured by the fact that the *only way an H string can start with an X is by starting with* $X\overline{A}H_0$. And since all *G strings must start with an X*, we can be sure that *any matching set starts with* $X\overline{A}$. Similarly, any matching pair of strings must end with a transcription of Z—more precisely, must end with $\overline{Z}XY$—because a G string can't end in X and an H string can end only with X or Y. It follows that if there is any solution at all to this correspondence problem, then the solution must be a string which can be resolved into the two forms:

$$G_0G_{i_1}, G_{i_2}\dots G_{i_N}\overline{Z}XY$$
$$X\overline{A}H_0H_{i_1}H_{i_2}\dots H_{i_N}Y$$

But if this is the case, then it follows that the sequence of G_{i_j}'s is exactly the sequence that would be followed by the original monogenic normal system (g_i, h_i)! We can see this inductively: we have already established that the H string must begin with $X\overline{A}$. Then the G string *must* begin with G_0. Why? Because the system is *monogenic*! That means that the beginning of axiom A can be matched only by g_0. Then g_0 determines h_0—the string to be added to A—and let us remove G_0 from the front. Then there is *only one* G_{i_1} that can match the beginning of the remaining string; this corresponds to the g_{i_1} that the normal system would apply at the next step. Then G_{i_1} determines H_{i_1}—the string that is to be added to what is left after deleting G_{i_1} from the front. Again, G_{i_2} is determined, because the system is monogenic and it, too, must be precisely the production antecedent the normal system would use at its second stage.

Thus the sequence of G_i's must be the same as that of the g_i's underlying the monogenic normal system (and hence must represent the steps on the computation of the still-further underlying Turing-machine computation). If the process terminates, then the last G that was used will be followed in the string by Z—by definition that which will remain after no more productions can be applied.

Therefore, if the strings match, then they must both be the sequence mentioned in the assertion.

So far, the only monogenic normal systems we have are the tag systems. We could, however, have used theorem 14.1-1 more directly to show that *monogenic normal systems are universal*: Consider an arbitrary two-register machine, and represent its state by a word of the form

$$I_j \quad 111\dots111 \quad K_j \quad 111\dots111$$

Then, using the same methods as we used in 12.6, we can realize the instruction types of theorem 14.1-1 as follows.

Conditional subtract from first register:

$$I_jK_j\$ \rightarrow \$I_{j'}K_{j'} \qquad \text{(i.e., go to } I_{j'} \text{ if register is empty)}$$
$$1\$ \rightarrow \$1 \qquad \textit{or}$$
$$I_j1\$ \rightarrow \$I_{j+1} \qquad \text{(subtract 1 and go to } I_{j+1})$$
$$K_j\$ \rightarrow \$K_{j+1}$$

Conditional subtract from second register:

$$I_j\$ \rightarrow \$I_j^* \qquad \text{(rotate to examine second register)}$$
$$K_jI_j^*\$ \rightarrow \$K_j^*I_{j'} \qquad \text{(go to } I_{j'} \text{ if register is empty)}$$
$$K_j^*\$ \rightarrow \$K_{j'}$$
$$K_j1\$ \rightarrow \$K_{j+1} \qquad \text{(subtract 1 and go to } I_{j+1})$$
$$I_j^*\$ \rightarrow \$I_{j+1}$$

Add to first:

$$I_j\$ \rightarrow \$I_{j+1}1, \qquad K_j\$ \rightarrow \$K_{j+1}$$

Add to second:

$$I_j\$ \rightarrow \$I_{j+1}, \qquad K_j\$ \rightarrow \$K_{j+1}1$$

PROBLEM. Can you make a similar construction for three or more registers?

14.8 "SMALL" UNIVERSAL TURING MACHINES

The existence of universal machines is surprising enough, and it is startling to find that such machines can be quite simple in structure. One may ask just how small they can be; but to answer this, one needs to have some measure of size, or complexity, of a machine. Several measures can be defended; Shannon [1956] suggests that one might consider the product of the number of symbols and the number of states, since, as he shows, this product has a certain invariance. One can exchange states and symbols without greatly changing this product. To count the number of quintuples would be almost the same.

In this section we describe the universal Turing machine with the smallest known state-symbol product. This machine is the most recent entry in a sort of competition beginning with Ikeno [1958] who exhibited a six-symbol, ten-state "(6, 10)" machine, Watanabe [1960] (6, 8), Minsky [1960] (6, 7), Watanable [1961] (5, 8), Minsky [1961] (6, 6), and finally the

(4, 7) machine of this section (described in Minsky [1961]). The reader is welcome to enter the competition (I believe that a certain one of the (3, 6) machines might be universal, but can't prove it)—although the reader should understand clearly that the question is an intensely tricky puzzle and has essentially *no* serious mathematical interest. To see how tricky things can become, the reader can refer to my 1961 paper describing the (6, 6) machine; it is much more complicated than the machine of this section.

14.8.1 The four-symbol seven-state universal machine

The very notion of a universal Turing machine entails the notion of *description*; the machine to be simulated has to be described, on the tape of the universal machine, in the form of some code. So, also, must the initial tape or data for the simulated machine be described. One way to do this encoding is to write, almost literally, the quintuples for the simulated machine on the tape of the universal machine; this is what we did for the machine of chapter 7. On the other hand, there is no particular virtue in the quintuple formulation, and one might be able to get a simpler universal machine with some other representation. Nevertheless, we must not go too far, for if one is permitted an arbitrary partial-recursive computation to do the encoding and is permitted to let the code depend on the initial data, then one could use as the code the result of the Turing-machine computation itself, and this would surely be considered a cheat! (It would give us a (2, 0) machine, since the answer could be written in unary on a tape, and no computer would be necessary.) We have to make some rule, e.g., that nothing like full computation power may be spent on the encoding. Informally, *this will be guaranteed if the encodings for the machine structure and for the data are done separately.* Then we can be sure that the machine was not applied to the data during the encoding process. This condition, we claim, justifies what we do below. More technically, one might require, for example, that the encoding process be a primitive-recursive, symbol-manipulation operation on the input; this, too, would guarantee that if the resulting machine is universal, this is not due to some power concealed in the encoding process. Davis [1956] discusses this question. We will present first the encoding for our machine and then its state-symbol transition table.

The four-symbol, seven-state machine will work by simulating an arbitrary $P = 2$ tag system. We know, by theorem 14.6-1, that if a machine can do this, it must be universal. We know, by the proof of theorem 14.6-1, that representing the quintuples of an arbitrary Turing machine in the form of a $P = 2$ tag system is a tedious but trivial pro-

cedure. Since one can see in advance for any such representation how much work will be involved, the conversion is a primitive-recursive operation. In fact, it involves only writing down sixteen productions for each quintuple.

Suppose, then, that the tag system is:

$$\begin{aligned}&\text{Alphabet: } a_1, a_2, \ldots, a_m\\ &\text{Productions: } a_1 \to a_{11}a_{12}\ldots a_{1n_1}\\ &\qquad\qquad a_2 \to a_{21}a_{22}\ldots a_{2n_2}\\ &\qquad\qquad \ldots\ldots\\ &\qquad\qquad a_m \to a_{m1}a_{m2}\ldots a_{mn_m}\end{aligned}$$

where n_i is the number of letters in the consequent of a_i. For each letter a_i we will need a number N_i computed as follows:

$$\begin{aligned}N_1 &= 1\\ N_{i+1} &= N_i + n_i\end{aligned}$$

so that $N_i = 1 + n_1 + n_2 + \cdots + n_{i-1}$. Note that this is just one more than the number of letters in the production consequents preceding that of a_i. We shall see the reason for this definition shortly.

We will represent any string (e.g., an axiom) $a_r a_s \ldots a_z$ on U's tape by a string of the form

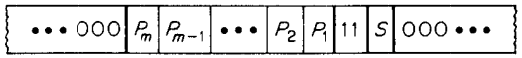
$$S = y^{N_r}Ay^{N_s}A\ldots Ay^{N_z}$$

so that N_i, used as the length of a string of y's, is used to represent a_i to the machine. The A's are spacers.

The productions will be represented as follows. Define

$$P_i = \underline{11}\,0^{N_{in_i}}\underline{01}\ldots\underline{01}\,0^{N_{i2}}\underline{01}\,0^{N_{i1}}$$

so that the representation of the consequent of a_i begins with 11 and then has representations of the consequent's letters—in reverse order—separated by 01's. (We use strings of 0's here instead of strings of y's.) Now, finally, we can describe the whole of U's tape; it is:



The secret of the encoding is this: The pairs 11 and 01 are used as punctuation marks; 11 marks the beginning of a production and 01 marks spaces between letters in a production consequent. *There are exactly n_i punctuation marks in the* ith *production P_i*, and there is one extra 11 just to the left of S. Hence, *there are exactly N_i punctuation marks between S and the beginning of P_i*. So the code y^{N_i} chosen to represent a_i contains

exactly the information needed to locate the production corresponding to that letter!

The machine will use only the letters already introduced: 0, 1, y, and A. It is understood that '0' is also the blank symbol on the remainder of the infinite tape. Table 14.8-1 is the state-symbol table for the machine; it is understood that if no new state is given, the machine remains in its present state.

Table 14.8-1

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
y	0 L	0 L/1	y L	y L	y R	y R	0 R
0	0 L	y R	HALT	y R/5	y L/3	A L/3	y R/6
1	1 L/2	A R	A L	1 L/7	A R	A R	1 R
A	1 L	y R/6	1 L/4	1 L	1 R	1 R	0 R/2

The machine starts in q_2 at the first symbol in S. It seems useless to try to explain the machine, except by following it through an example, because its various functions are all mixed up. Generally, states q_1 and q_2 read the first symbol in S, locate and mark the corresponding production P_i, and erase the first symbol in S. States q_3, q_4, q_5, and q_6 then copy the production consequent at the end of S. (The copying works from inside to out; this is why the productions were written backwards.) When the end of the production is detected (by q_4 and q_7 finding the 11), then q_7 restores the tape, removing the marking of the production region and, incidentally, erasing another symbol from S. Since two symbols were erased from S, and the appropriate production is copied, we have a $P=2$ tag process.

The problem in making a "small" machine is to avoid use of new letters for marking. All marking of working places for this machine is done by interchanging 0's and y's and 1's and A's.

If q_3 meets a 0, the machine halts. It turns out that this can happen only under special conditions; but these conditions will come about, eventually, if the special string P_H = 110101 is used as a production and this production is referenced. So if any of the letters of the tag system are supposed to cause a halt, we assign to them the production P_H. The number n_H assigned to the halt symbol P_H is 3.

Constructing and following an example is tedious. Here is a simple one.

AN EXAMPLE

We will code the machine for the simple tag system

$$\begin{aligned}a_1 &\to a_2\\ a_2 &\to a_2a_3\\ a_3 &\to \text{halt}\end{aligned}$$

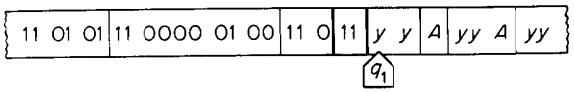
For this system,

$n_1 = 1$ $N_1 = 1$ $P_1 = 11\ 0$

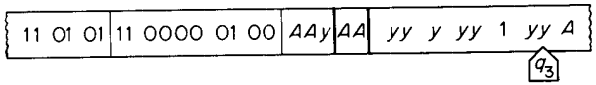
$n_2 = 2$ $N_2 = 2$ $P_2 = 11\ 0000\ 01\ 00$

$N_3 = 4$ $P_3 = 11\ 01\ 01$ (the halt production)

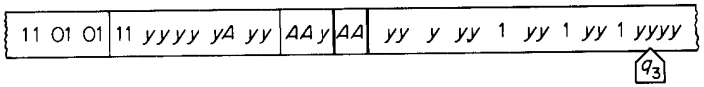
If we start with axiom $a_2a_2a_2$, this is encoded as $yyAyyAyyA$ and the tape is:



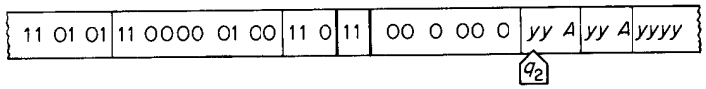
The machine marks symbols to the left, finding two punctuation groups, and then goes to the right in state q_6, writing an A at the end. The tape is then



Note that two deletions have been made from the front of the tag string! Now the production is copied (backwards) at the end; two 0's, and A and four more 0's, forming $yy\ 1\ yyyy$:



On the next trip to the left, the machine encounters the 11, meaning that copying is to stop; the machine enters q_7 and restores the tape to the form

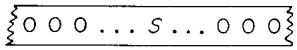


This is, in effect, like the starting state (q_2 has the same effect as q_1) except that the string $a_2a_2a_2$ has been replaced, as it should be, by $a_2a_2a_3$.

If you trace the operation through to the end, you will see how the string next becomes $a_3a_2a_3$. Following that, after some curious struggles, the symbol a_3 will cause the machine to halt; it first writes AA and this sequence eventually causes state q_3 to encounter a zero.

14.8.2 Structure of universal machines

One might suppose, or hope, that the property that a Turing machine is universal should imply some interesting conclusion about its state diagram. But it seems there is nothing much to say about this, in general, because there are universal machines with structures so trivial that one can draw no interesting conclusions. Suppose, for example, we make a straightforward machine for a $P=2$ tag system. Let the axiom S be written out on the tape



and start the machine at the beginning of S with the state diagram shown in Fig. 14.8-1. Now we can make such a tag machine for any Turing machine, by section 14.6; so we can also do it for some universal Turing machine; hence there is a machine with this structure that is universal.

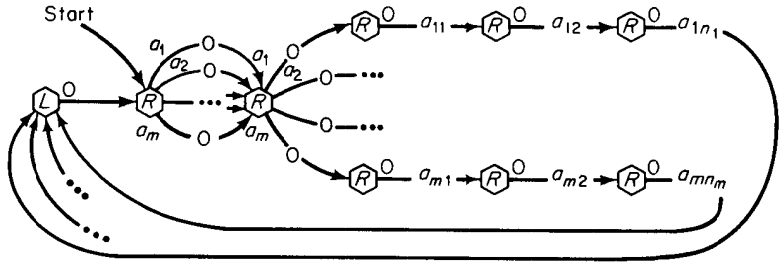


Fig. 14.8-1

There simply doesn't seem to be any structure required that is any more complicated than one needs to make a multiplication machine. Perhaps this is not surprising in view of theorems like theorem 14.2-1 and the difficulty of excluding full recursion, e.g., minimization, in any machine that has any iterative (loop) ability. In any case, the demonstration by Shannon (1956) that, allowed enough symbols, one can replace any Turing machine by a two-state machine shows that the structure of the state diagram can be hidden in the details of operation and not clearly represented in the topology of the interstate connections.

PROBLEM. Choose any two-symbol, two-state machine and show that it is *not* universal. Hint: Show that its halting problem is decidable by describing a procedure that decides whether or not it will stop on any given tape. D. G. Bobrow and the author did this for all (2, 2) machines [1961, unpublished] by a tedious reduction to thirty-odd cases (unpublishable).

15 SOLUTIONS TO SELECTED PROBLEMS

Chapter 2

2.7-2. The output 0 indicates that up to the present, 0's and 1's have always occurred in pairs. Note that the lower-right-hand state is "dead" in the sense that the machine can never leave it, once entered. In the notation of chapter 4, the output 0 corresponds to the equivalence class of histories: $(00 \vee 11)*$.

2.7-3. If the machine has k states, and it got to the point of printing the $k + 1$ zeros in $\ldots 0^{k+1}1 \ldots$, it would have repeated a state and could never print the '1' and continue the "counting."

2.7-4. The outputs do not contain any information not already contained in the history that can affect later outputs.

2.7-5. The basic idea is to propagate two kinds of waves down the line, one going *three* times as fast as the other. When the fast wave is reflected, it meets the slow wave in the *middle* of the chain. The chain can then be divided into two equal parts (it is a mere detail to account for odd and even chain lengths) and the process duplicated in each half-chain, with new "fire when ready" commands starting at the center. A soldier actually fires at the moment he finds himself an isolated chain of length 1.

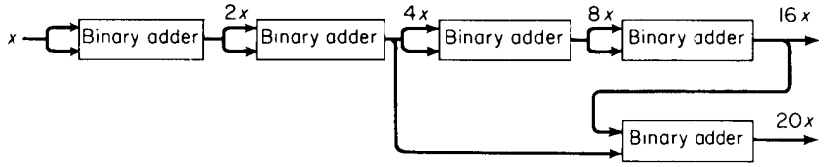
This solution takes $3n$ moments, for a chain of n soldiers. One might conjecture that there might be a solution of the order of $2n$. (Show there can be no faster solution.) In fact it is not hard to construct solutions, for any small number $\epsilon > 0$, of the order of $(2 + \epsilon)n$. It is quite a bit more difficult to see that there is an exactly $2n$ solution; this was first shown by E. Goto and one was found by Robert Balzer, of Carnegie Institute of Technology, using only 8 states for each soldier, and 8 kinds of signals flowing in either direction—equivalent to 64 inputs.

A published solution, with some further related theory and results, appears in a paper by Patrick C. Fischer [1965].

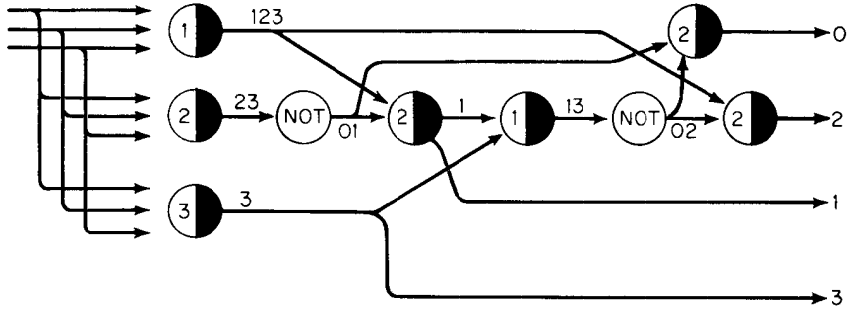
Chapter 3

3.2-1. The key fact is that the spurious numbers produced by a binary counter chain during carries represent smaller numbers than the current count.

3.2-4. We show an example of how to do it; the reader will be able to see the general method. To multiply a number x by 20, use the net shown here.

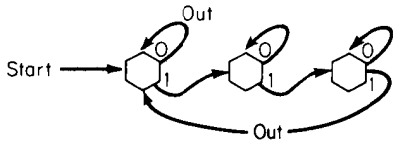


3.6-1. The key idea is shown in this net.

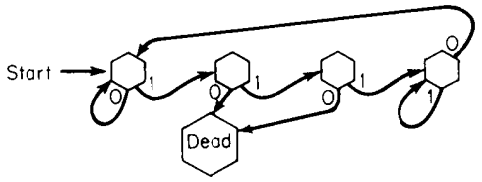


Chapter 4

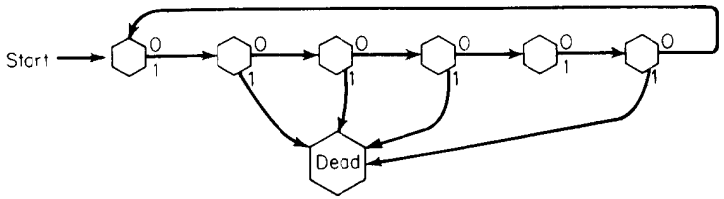
4.2-1. (1) $(0*10*10*1)*0*$



(2) $(0 \vee 1111*)* = (0*1111*)*$



(3) $((0 \vee 1)000(0 \vee 1)0)*$



(4) $[0*1(11)*0*01(11)*0*0]*(11)*$

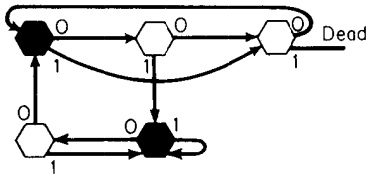
4.3-2. $R_{11}(ae \vee abg \vee dg)(h \vee fg)*$

4.3-3. Rules: $E \vee F = F \vee E$, $(E \vee F)* = (E*F)*E*$ $(EF)*E = E(FE)*$

4.3-4. True, False, False, True, True,

4.5-4. $E = (01 \vee 001*0 \vee 11*0)*$

$E^R = (10 \vee 01*00 \vee 01*1)*$



The two states cannot be merged because 0011 is accepted after 01 but not after the 000, so they cannot represent equivalent histories.

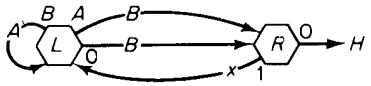
4.5-5. (1) Yes: $(0 \vee 1)*$. (2) Yes: $10*$. (3) No: finite-state machine must lose count. (4) No: must lose count. Note that an infinite machine like that in section 4.2.3 can do it. (5) No. (6) Yes: $(0 \vee 1)*E$. (7) Yes: $(0 \vee 1)*E(0 \vee 1)*$. (8) No! This is not solved by EE. A machine can recognize $E = 0*1$, but no machine can recognize

$$\{0^n10^n1\} = 0101, 001001, 00010001, \ldots$$

4.5-6. $(a*b*)*$, $(a*(bb)*(ba)*)*$, $(a*((bb)*(ab)*)*)*$. This works in general, if one operates from the outside-in.

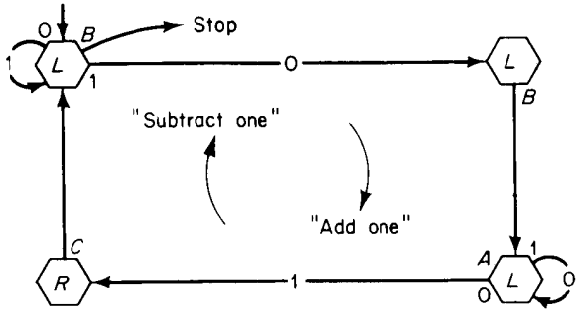
Chapter 6

6.1-1. The left-going state is a complete binary-counter in itself! (One can further omit the symbol B and use 1 for it throughout.)

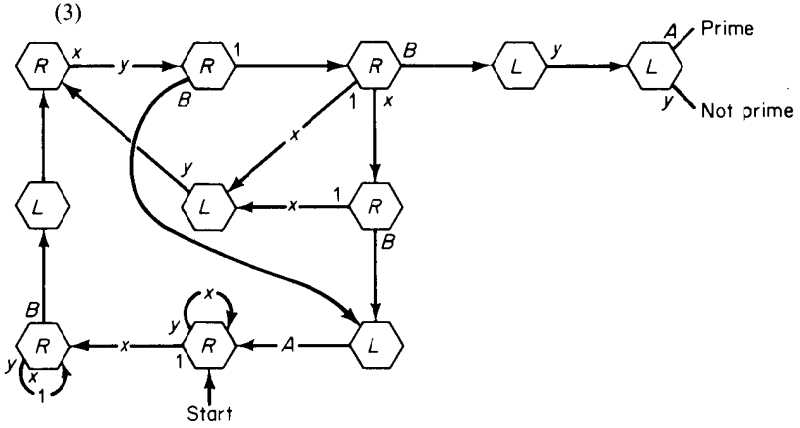


6.1-2. (1) Make a machine that first recopies the number; then acts like the unary multiplier of section 6.1.4.

(2) Naturally, there are many ways to do this, so your solution won't be the same as this one. The sum ends up to the left of B. Why do we use two A's?

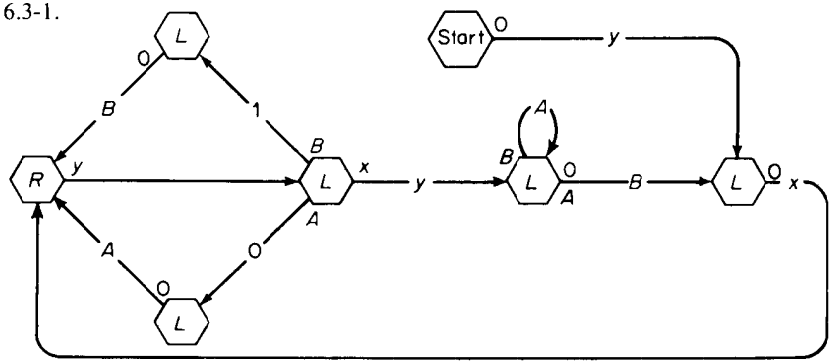


(3)

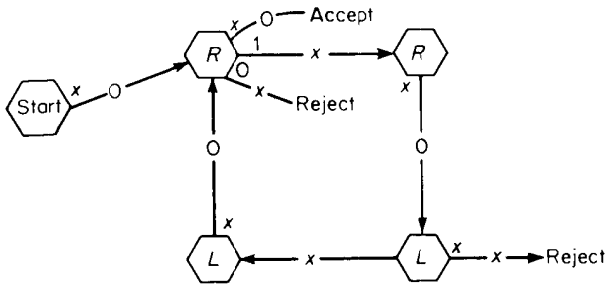


6.1-3. Eventually the tape acquires arbitrarily long segments of the infinite pattern

6.3-1.



6.3-3. (1)



(2), (3) Just use the given finite state machine functions

$$F(q_i s_j) \qquad G(q_i s_j)$$

to make quintuples

$$\{q_i, s_j, G(q_i s_j), 0, 1\} \qquad \text{for } S_j = 0, 1$$

and

$$\{q_i, X, \text{HALT}, \phi(i), 1\}$$

where $\phi(i) = 0$ if q_i is an acceptable final state and $\phi(i) = X$ if not.

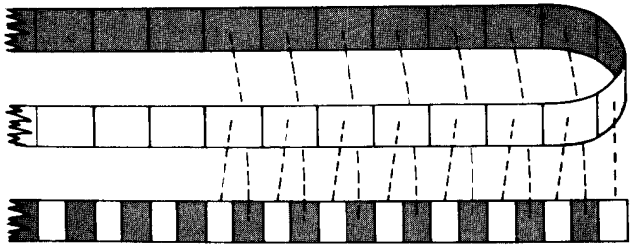
(4) Obviously, the only difference between a general right-moving machine and the machines above concerns the s_{ij} entries; and since the machine can't go back to look at them, they might as well be 0's!

(5) No. Because of (2) and the fact that a finite-state machine can't do it.

(6) No, not unless you are as clever as Turing. But you will be able to after reading chapter 8.

Chapter 7

7.4-1. It is easy to describe the key idea; think of a new tape that represents T's doubly-infinite tape as folded over, as in the accompanying figure. Then all the information is "mapped," in a one-to-one fashion, onto the new singly-infinite tape. To use this would, of course, require a drastic revision of the details in figs. 7.2-5 and 7.2-7. A more elegant solution is obtained by solving problem 7.4-2, for the machine so derived works on a doubly-infinite tape immediately.



7.4-3. Begin by designing a machine T which, given any sequence of symbols from a fixed alphabet $\{s_1, s_2, \ldots, s_r\}$:

$$s_{i_1} s_{i_2} \ldots s_{i_n}$$

converts this to a sequence

$$(q_0), 0, (q_1), s_{i_1}, 1, X, (q_1), 0, (q_2), s_{i_2}, 1, X, \ldots$$
$$\ldots, (q_n), 0, (q_{n+1}), s_{i_n}, 1, X, s_{i_1}, s_{i_2}, \ldots, s_{i_n}.$$

where (q_i) is a representation of the integer i. Then consider the machine T and let the sequence $s_{i_1} s_{i_2} \ldots s_{i_n}$ be a representation of the machine T itself, where the description of T starts with state q_{n+1}. The long sequence of quintuples then describes the desired machine!

Chapter 8

8.3-1. The machine T^B is essentially the machine T, plus machinery for erasing all information on T's initially given tape. T^B begins by writing $A\,0\,B$ and starts to compute at the location of this 0. When it needs more "blank" space, it pushes the A and B apart, sweeping away whatever might originally have been written on the tape.

8.3-3. This is simply a special case of the problem of section 8.3.4 because, for the kinds of machines used there, it is quite irrelevant whether the initial tape is finitely or infinitely inscribed.

8.8-1. (1) There *is* a decision procedure for this. For given the length L of the non-blank part of t, and the number of states N of T, one can calculate the bound $B(L, N) = L \cdot N^2$. The machine T cannot go more than $L \cdot N$ steps without either repeating a state-tape configuration, or erasing a 1, or getting into a repeating pattern moving forever down the infinite blank part of the tape. And there are at most N 1's to erase. So if T has not written a 1 in LN^2 steps, it never will.

(2) The answer is No, by Theorem 14.5.

8.8-2. (1) No, for the same reason as in 8.3.4.

(2) No, same reason, essentially.

(3) Design machines like those of 8.3.4 which, when their prototypes halt, then erase everything between A and B. Let T be a machine that halts under the conditions of 8.3.4. Then this problem is equivalent to the halting problem for T'.

(4) This *is* decidable! Calculate a bound as in problem 8.8-1(1) and modify to keep count and check to see if T is in a cyclic loop at that time.

Chapter 9

9.2-1. If the rational number is defined as p/q, we design a Turing machine $T_{p,q}$ that first prints out the integer part of p/q (by recording its digits in the

first few states of $T_{p,q}$). Then we have to print the possibly infinite decimal sequence for the remainder. But this sequence is always eventually periodic, e.g., like

$$\tfrac{22}{7} = 3.\underline{142857}\ \underline{142857}\ \underline{142857}\ \ldots$$

so this cycle can be built directly into $T_{p,q}$. Incidentally, can you use a finite-state argument to prove that the sequence of digits of a rational number is periodic?

9.2-2. Given the machine T_a which gives a_n = the nth digit of a when n is written on the tape, we build a new machine T'_a whose tape has (at the end of each digit computation) the form

$$\ldots Xa_0Xa_1Xa_2X\ldots Xa_nX(n)Y$$

The new machine T'_a reads the number between X and Y, does what T_a would do, using methods like those of 8.3.4 modified and extended to keep the T'_a calculation from overlapping the already-printed material preceding the last X. When finished, the tape should have the form

$$Xa_0Xa_1X\ldots Xa_nXa_{n+1}X(n+1)Y$$

The machine will have had to maintain a copy of n throughout the computation, but this is easy to arrange. To show the other direction of equivalence is easy: given a machine like T'_a, build a T_a that, given n, allows T'_a to run until n X's are printed; then erase everything on either side of the number to the left of the last X.

9.3-1. The key idea is simple but fruitful. Begin by supplying M with some representation of the nth tape which it can interpret as the integer N. It puts a variant of the machine T_{R_U} into "supervised operation." It waits until T_{R_U} has printed exactly n X's (as in problem 9.2-2). It then stops simulating T_{R_U} and inspects the digit to the left of the last X that was printed. This digit determines the answer.

9.6-3. The number $\frac{1}{3}$ = .33333... is obviously computable, so $f(\frac{1}{3})$, if computable, must be defined. But there is no way to decide its value at any finite point of inspecting the input tape. For at any finite point, the machine will have seen nothing but 3's. So it must check the next input digit. It can decide safely only when it first encounters a digit below 3 (in which case $f(x) = 0$) or one above 3 (in which case $f(x) = 1$). If there is no such digit, and that is the case here, it can never halt!

Chapter 10

10.3-3. If y and z are represented as binary numbers, but we write, in base three;

$$f(y,z) = \text{“}y\text{”}2\text{“}z\text{”}$$

then we waste $\frac{1}{3}$ of each digit as well as one whole digit. Then

$$f(y,z) \sim 3 \cdot (yz)^{1+1/2}$$

In general, by using base r, we get

$$f(y,z) \sim r(yz)^{1+1/r}$$

and $1/r$ can be as small as we like. One can do a little better by using more complicated codes and can keep

$$\mathsf{C}(y,z) < yz(1+\epsilon)$$

(for any ϵ and large enough y and z). One can't do better than

$$\mathsf{C}(y,z) \sim y \cdot z$$

because for any y_0, z_0 there are $y_0 \cdot z_0$ pairs (y,z) for which

$$y < y_0 \quad \text{and} \quad z < z_0$$

10.3-4. To find y, given $\mathsf{C}(y,z)$, find the largest n for which

$$\mathsf{C}(y,z) - \frac{n^2+n}{2} \geq 0$$

Then this difference will have the value of y; and there is a similar procedure for z.

10.4-1. The summation was done in 10.3.2; the product is similar.

10.4-2. Define

$$E(x,y) = 1 \quad \text{if } x = y$$
$$E(x,y) = 0 \quad \text{if } x \neq y$$

Define

$$\phi(x) = \sum_{k=1}^{x} \prod_{j=1}^{k} E(\psi(j), \chi(j))$$

This is the desired ϕ function. Verify that the definition above can be put in primitive-recursive form, using problem 10.4-1.

10.5-1. $A(0) = 1, A(1) = 3, A(2) = 7, A(3) = 61, A(4) = 2^{2^{2^{2^{16}}}} - 3.$

10.6-1. I don't know any especially neat ways to do these, though there is no particular conceptual difficulty. What I would do, if I had to describe enumerations, is to describe (1) a procedure for enumerating, say, lexicographically, all strings of permitted symbols and (2) a procedure for testing the strings to see whether they are well-formed sets of equations satisfying the definitions of primitive (or general) recursion. These procedures would take only a few lines to describe, using one of the modern "string manipulation" computer programming languages mentioned in the note to chapter 12.

10.6-2. Assuming an effective procedure for evaluating the definitions found in 10.6-1, simply evaluate the nth definition found for the argument value x. If one goes into details, one will have to worry about functions of more than one variable.

10.6-3. If one remembers that the μ operator is needed only once, this problem isn't very different from 10.6-2.

10.6-4. The function $V(a, b)$ isn't primitive-recursive.

10.7-1. (1)

$$\text{Prime}(n) = P(n, 2)$$
$$P(n, k) = (\textit{if } n = k \textit{ then } 1 \textit{ else } (\textit{if } n = k \cdot (n/k) \textit{ then } 0 \textit{ else } Pr(n, k + 1))).$$

(2) Define

$$R(m, n) = m - n \cdot \frac{m}{n} = \text{the remainder when } m \text{ is divided by } n$$

Then

$$\gcd(m, n) = (\textit{if } m > n \textit{ then } \gcd(n, m) \textit{ else } (\textit{if } R(n, m) = 0 \textit{ then } m \textit{ else } \gcd(R(n, m), m)))$$

(3)
$$\phi(n) = \psi(n, n - 1)$$
$$\psi(n, k) = (\textit{if } n = 1 \textit{ then } 0 \textit{ else } (\psi(n, k - 1) + N(\gcd(n, k))))$$

10.9-1. Define integer equivalents to be

0	1	2	
↕	↕	↕	etc.
NIL	<NIL · NIL>	<NIL · <NIL · NIL>>	

That is, we let NIL represent 0 and, if n represents an integer, we let C(NIL, n) represent its successor. Then, for integers x and y, we can define

$$\text{equal}(x, y) = (\textit{if } x = \text{NIL } \textit{then } (\textit{if } y = \text{NIL } \textit{then } \text{NIL } \textit{else } C(\text{NIL}, \text{NIL})) \text{ else } (\textit{if } y = \text{NIL } \textit{then } C(\text{NIL}, \text{NIL}) \textit{ else } \text{equal}(T(x), (T(y)))))$$

Then "equal (x, y) = NIL" will be true only if x and y are equal integers.

Chapter 12

12.3-1. The trick is to consider a, b, and c to be the 0, 1, 2 digits of a ternary number system and construct a machine like that of the solution of problem 6.3-1 but with ternary addition.

12.3-2. Use exactly the same trick as in 12.3-1, except using base-4 numbers, with the extra digit playing the role of string-separator, i.e., "comma."

12.3-3. A string of length n can be cut in any or all of $n - 1$ places. At each place there are two possibilities—divided or not. So altogether there are 2^{n-1} possibilities.

12.4-1. Prove this by induction. BASE: all palindromes of length 1 and 2 are included (as axioms). Any palindrome of length $n > 2$ has the form xPx where P is a palindrome of length $n - 2$. Use two induction hypotheses for odd and even lengths.

12.4-2. If N is the number of states of the machine, it must accept the palindrome

$$aa^Nba^Na$$

but in so doing it must have repeated a state during the first sequence of a's. If L is the length of the cycle of states thus encountered, the same machine must also accept

$$aa^{N+L}ba^Na$$

which is *not* a palindrome.

12.4-6. Any parenthesis nest can be regarded as built up from the outside. Thus we can obtain the nest of problem 12.4-5 by

$$\underline{()}$$
$$(\underline{()})$$
$$(()\underline{()})$$
$$(()()\underline{()})$$
$$(()()(\underline{()}))$$
$$(()()((\underline{)}))$$
$$(()()(((\underline{)})))$$

To get, say ( ) ( ) from ( ), we let \$1 be ( ) and \$2 be the null string. To prove the system generates the well-formed strings, use methods like those of section 4.2.3.

12.6-1. If the s_k's were left out, we would have both the productions

$$\$_1 q_i s_j \$ \rightarrow \$_1 s_{ij} q_{ij} \$_2$$

and

$$q_i \$ \rightarrow 0 q_i \$$$

Then for a string that *begins* with q_i, two productions would apply to the same string, and we could get two different proofs, eventually, of some theorem.

12.6-2. A simple set of productions that will suffice here are:

$$\$_1 q_0 0 \$_2 \rightarrow \$_1 q_0 \$_2$$
$$\$_1 q_0 1 \$_2 \rightarrow \$_1 q_1 \$_2$$
$$\$_1 q_0 B \$_2 \rightarrow q_0 H$$
$$\$_1 q_1 0 \$_2 \rightarrow \$_1 q_1 \$_2$$
$$\$_1 q_1 1 \$_2 \rightarrow \$_1 q_0 \$_2$$
$$\$_1 q_1 B \$_2 \rightarrow q_1 H$$

where the process stops by proving $q_0 H$ or $q_1 H$. Because of the character of this problem, we don't need the tape-supplying productions.

12.6-3. The axiom must have the form of a well-formed parenthesis nest, with x for '(' and y for ')'.

12.6-5. It is possible to make a system that produces only square numbers, e.g.:

Alphabet: 1

Axiom: 1

Production: \$ → \$\$\$\$

but this doesn't produce *all* the square numbers.

Now any production on the alphabet with just '1' can be put in the form

$$\$_1\$_2 \ldots \$_n 1^m \rightarrow \$_1^{k_1}\$_2^{k_2} \ldots \$_n^{k_n} 1^p$$

(Why?) Now either the k's are all equal or not. If not, then there is some $k_i \neq k_j$. Then consider any number x produced by the system, and any number $r < x$:

$$x = (x - m - r) + r + m \rightarrow k_i(x - m - r) + k_j r + l$$
$$= k_i x + (k_j - k_i)r + l - k_i m$$

If $k_i = 0$, then this implies that the system contains the arithmetic series $r(k_j - k_i) + (l - k_i m)$ in r, and the set of square numbers contains no arithmetic series. But if k_i is not zero, then the system contains arbitrarily large pairs of numbers just $(k_j - k_i)$ apart—also impossible. Hence all k's must be equal. In that case

$$x = (x - m) + m \rightarrow k(x - m) + l$$

and each production generates a sequence of numbers that grow exponentially. No finite set of such sequences can form the set of square numbers. (Show that they get too far apart.)

12.7-1.

Alphabet: 0, 1

Auxiliary letter: A

Axiom: A

Productions: \$$A$ → \$0A

\$$A$ → \$1A

\$$A$ → \$\$

It cannot be done without an auxiliary letter. A detailed proof is very difficult, but to get an idea of what happens, consider any production

$$g_0\$_1 g_1 \ldots \$_k \ldots \$_n g_n \rightarrow \ldots \$_k \ldots$$

Now consider any double string of the form

$$g_0 g_1 g_2 \ldots g_k S g_{k+1} \ldots g_n g_0 g_1 g_2 \ldots g_k S g_{k+1} \ldots g_n$$

For any S, such a string must be a theorem. When this string is given to the corresponding production, one can set $\$_j = 0$ unless $j = k$ so that

$$S g_{k+1} \ldots g_n g_0 g_1 \ldots g_k S$$

replaces each occurrence of $\$_k$ in the consequent. Because S is arbitrary, the result won't be a double string, in general, unless $\$_k$ occurs twice; and because this is true for each k, one can show that the consequent itself must be "repeated." Then one can use subtle arguments about exponential growth of the strings as in the solution to 12.6-5 (p 292).

12.8-1. The following canonical system is an extension for the regular expression $(b \vee c)ab(a \vee bc*)*$:

Axiom: XYZ

Productions: $X\$ \rightarrow b\$$

$X\$ \rightarrow c\$$

$\$Y\$' \rightarrow \$ab\$'$

$\$Z \rightarrow \$$

$\$Z \rightarrow \$WZ$

$\$W\$' \rightarrow \$a\$'$

$\$W\$' \rightarrow \$bV\$'$

$\$V\$' \rightarrow \$\$'$

$\$V\$' \rightarrow \$cV\$'$

One can see in this that X is really $(b \vee c)$, Y is ab, Z is $(a \vee bc*)*$, W is $(a \vee bc*)$ and V is $c*$. What is the general principle here? Describe the construction inductively.

12.8-2. For example, let M be described by quadruples $(q_i, s_j, q_{ij}, r_{ij})$, let q^A be the set of states regarded as indicating that M, when started in q_0, accepts the input string, and consider a system like

Alphabet: $Q_0, Q_1, \ldots, Q_n; s_1, \ldots, s_m$

Axiom: Q_0

Productions: $Q_i\$ \rightarrow Q_{ij}\$s_j$ (all i and j)

$Q_i\$ \rightarrow \$$ (all q_i in q^A)

Prove that this system is a canonical extension of the set of strings recognized by M, and admire its elegance as an alternative to the concept of finite-state machine. Modify the system so that recognition depends on the occurrence of certain output symbols rather than on the occurrence of certain states.

12.8-3. This problem is quite complicated, although conceptually not very difficult. One has to keep, along with the axioms, a *theorem list* of all the strings generated up to the present time. One must also have a procedure for determining whenever any theorem on the theorem list can be matched to the antecedent (left-hand part) of any production; in that case one must then rearrange the discovered parts to form the consequent of that production and add the result to the theorem list. One must make sure that

each production is applied to every theorem (in fact, if multi-antecedent productions are considered [see 13.2], to each subset of theorems). One must also test to find every way in which each theorem matches each antecedent—for there may be more than one correct analysis (see 13.1)—and produce the corresponding theorems. Because of all this, the Turing machine required will be very complicated. We do not go through this construction here because, once we have proved the theorems in chapter 13, the kinds of productions we have to consider will become so much simpler that the corresponding Turing-machine construction will be quite trivial! For this reason, the reader is encouraged to analyze this problem carefully to see what is involved in the complicated required procedure, but he should not attempt to complete this effort by filling out details of the Turing machine. The same effort will be much better spent on trying to understand, and perhaps improve, the proofs in chapter 13.

Chapter 13

13.1-1. Rather than adapt the construction in 13.1, it is just as easy to prove the more general theorem: Any normal canonical system has a normal extension whose production constant strings g_i and h_i have no more than two letters, with not more than a total of three in any production. This problem is fairly hard. My proof uses many new symbols, in a hierarchy that gives a sort of binary representation for the constant strings of the original system.

13.3-1. Append $\$_4$ to the right-hand end of the antecedent of π_U.

13.3-2. U uses only the upper-case letters A, C, S, and T, and lower-case a and b. The simulated alphabet of Q is ab, aab, $aaab$, ..., $(a)^r b$. So bb never occurs. Then there will be no ambiguity if we replace A, C, S, and T by AbA, $AbbA$, $AbbbA$, and $AbbbA$; and the test for falsely placed upper-case letters still works.

13.3-5. Construct P with only three letters, L, a, and b. Let ab, aab, $aaab$, etc. represent letters of Q and let Lb, Lbb, $Lbbb$, etc. serve as the auxiliary letters of P. Finally, construct U using LLb, $LLLb$, $LLLLb$, etc., as *its* auxiliary letters! Then one has only to construct a release procedure that will not release any string containing L in its interior.

13.3-6. Take A and T to be new letters: let $T\$$ assert that $\$$ is a theorem in the old system; then use

$$T\$_1 0\$_2 11\$_3 \rightarrow A\$_2 A\$_3 A\$_1 0\$_2 11\$_3 A$$

$$A\$_1 xA\$_2 xA\$_3 A \rightarrow A\$_1 A\$_2 A\$_3 A$$

$$AAA\$_1 0\$_2 11\$_3 A \rightarrow T\$_1 0\$_2$$

Chapter 14

14.2-1. If the number in the single register R is $2^r 3^s$, then there is a correspondence between the instruction sets

Add 1 to r	$\leftrightarrow$	Multiply R by 2
Add 1 to s	$\leftrightarrow$	Multiply R by 3
Subtract 1 from r	$\leftrightarrow$	Divide R by 2
Subtract 1 from S	$\leftrightarrow$	Divide R by 3

with the usual conditions for subtract and divide. Then the proof of theorem 14.1-1 shows that we can write programs, *using the left-hand set of instructions*, that have the effect, if we set r to $2^m 3^n 5^a 7^z 11^w$, of the instructions

Add 1 to m, Add 1 to n, etc.

Subtract 1 from n, etc.

But then the corresponding programs, using the right-hand instructions will have the same effect, if one starts with R containing

$$2^{2^m 3^n 5^a 7^z 11^w} \cdot 3$$

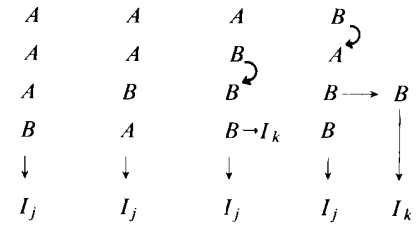
More details are given in Minsky [1961].

14.2-2. Define two new instructions that each operate on two registers, r and s.

A: Add 1 to r, exchange r and s, go to I_j

B: If $r > 0$, subtract 1 from r_1, exchange r and s, go to I_j; otherwise exchange r and s and go to I_k.

Now consider the four programs:



If we begin with even numbers in both registers, then these programs are. correspondingly, exactly like r', s', r^- and s^- except with double-sized increments.

The same thing, essentially, can be done where the "normal" sequence is the next instruction for I_j. Can you see the difficulty with the rightmost program, and how to fix it by preceding every program by "A, A, B, B"?

14.3-2. (1) No, because both 16 and (4) have the same image.
(2) Yes; if a number is even, it is not a list; if it is odd, it is a list.
(3) No, because both 8 and (1) have the same image.

16 SUGGESTIONS FOR FURTHER STUDIES AND DESCRIPTOR-INDEXED BIBLIOGRAPHY

INTRODUCTION

It is impossible to give a full account of a mathematical theory—not only because of the vast amount of knowledge accumulated in the past (as in the case of *computability*)—but also because the process of assembling an exposition, shaping and selecting different versions and images leads to new variations and models. Some of these are inessential, others are new families of structures that raise new problems in other areas of mathematical knowledge or ignorance. To keep this book compact, it was necessary to interrupt these lines firmly, and often arbitrarily. The following remarks concern lines of thought that some readers may want to pursue further. The initials at the right of each topic are used to cross-reference the bibliography.

ALGEBRAIC THEORY OF MACHINES — A

Beginning mainly with Kleene's Regular Expression Algebra, a new branch of abstract algebra has developed. It is now often seen in the more mathematical of the computer and linguistic journals. The few references we give here show how the state-transition diagrams have been studied by treating the input strings as elements of groups, semigroups, monoids, and other modern algebraic entities.

THEORY OF COMPUTATION C

There is very little theory today to help one prove that a particular program computes a particular function, or that two programs in the same or in different languages compute the same function. The fact that the problem is recursively unsolvable in its most general form should not be considered a complete obstacle, and one ought to proceed to a classification of important solvable cases, as does Ackermann [1954] for the predicate calculus.

DECISION PROBLEMS D

Some readers will want to pursue further study of recursive unsolvability and of the known solvable but still fairly large problem classes. We have just enough references for them to get started. Some of these point to fascinating variants of the sort discussed in 14.7 (Wang and his students treated these problems as infinite two-dimensional jig-saw puzzles).

FINITE AUTOMATA F

This category includes just a few references to topics in finite mathematics (*e.g.*, switching theory) that bear on the main subjects of part I.

FORMAL GRAMMARS G

This field, emerging from early works by Chomsky on linguistics, has become a significant branch of mathematics bearing both on programming languages and theory of Automata between Finite and General Recursive—the "Intermediate Machines."

COMPLEXITY HIERARCHIES H

Within the computable functions, one can find interesting definitions of hierarchies of complexity measures. Blum's [1964] theory, for example, classifies functions by the amounts of time or tape used in their computation: his theory is interesting in the way the same abstract structure arises from these two apparently different cases.

ARTIFICIAL INTELLIGENCE I

The author considers "thinking" to be within the scope of effective computation, and wishes to warn the reader against subtly defective arguments that suggest that the difference between minds and machines can solve the unsolvable. There is no evidence for this. In fact, there couldn't be—how could you decide whether a given (physical) machine computes a noncomputable number? Feigenbaum and Feldman [1963] is a collection of source papers in the field of programming computers that behave intelligently.

PROGRAMMING LANGUAGE L

There are a great many programming languages born each year which can be met in any issue of the *JACM*, the *Computer Journal*, the *Comm. ACM*, etc. We have referred here only to those most closely descended from Post Canonical Systems and List-Processing Systems.

INTERMEDIATE MACHINES M

It is obviously very important to find theories that apply to classes of computation lying between those of the simple Finite Automata and the full class of computable functions. A number of intermediate concepts are slowly taking form—"push-down automata," "linear bounded" and "real-time" computations, "counting automata," and others, many of which relate to various intermediate formal grammar (**G**) concepts. We also include here certain infinite "iterative array" and "growing" machine concepts.

NEUROPHYSIOLOGY N

We have included a few references to papers and books that might stimulate further study of the idea of the brain as a computer.

PROBABILISTIC MACHINES P

A very important practical question is that of whether "noise" or other physical realities of probabilistic nature can be tolerated by the theory. We have referenced a few papers that approach this question from different directions.

RECURSIVE FUNCTIONS AND DEGREES OF RELATIVE UNSOLVABILITY R

This theory, only introduced in Chapters 8, 10 and 11, is already a substantial branch of mathematics. The most comprehensive treatment is Rogers [1967], and one should see also Davis [1958] and Kleene [1952], for other connections with Mathematical Logic. Davis [1965] contains a splendid collection of reprints of important original source papers.
The theory of Church [1936], for example, gives a presentation of effective computation very different (at first sight) from any treated here.

TURING MACHINES T

A number of references concern slightly different formulations of Turing Machines. Fischer (1965b) sorts out many of these.

BIBLIOGRAPHY

Ackermann, Wilhelm (1954), *Solvable cases of the Decision Problem,* North-Holland. **D**

Blum, Manuel (1964), "*A machine-independent theory of recursive functions*," Doctoral Thesis, MIT. **R, H**

Bobrow, Daniel G. and Raphael, B. (1964), "A comparison of List-Processing Computer Languages," *Comm. ACM 7,* no. 4, 231–240 (April 1964). **L**

Bobrow, Daniel G. (1966), "METEOR: a LISP interpreter for string manipulation," *The Programming Language LISP: Its Operation and Applications,* MIT Press, Cambridge. July 1966. **L**

Buchi, J. R. (1962), "Turing Machines and the Entscheidungsproblem," *Math. Annalen 148,* 201–213. **D**

Bullock, Theodore H. and Horridge, Adrian G. (1965), *Structure and Function in the Nervous Systems of Invertebrates,* W. H. Freeman and Co., New York. **N**

Burks, Arthur W. and Wang, Hao (1957), "The logic of automata," *JACM 4,* 193–218 and 279–297. **F, A**

Burks, Arthur W. (1959), "Computation, Behavior and Structure in Fixed and Growing Automata," *Self-Organizing Systems,* Pergamon Press, 282–311. **M**

Chomsky, Noam (1963), "Formal properties of grammars," *Handbook of Mathematical Psychology 2,* John Wiley & Sons, 323–418. **G, A**

Chomsky, Noam and Schutzenberger, M. P. (1963), "The algebraic theory of context-free languages," *Computer Programming and Formal Systems,* North-Holland, 118–161. **G, A**

Church, Alonzo (1941), "The Calculi of Lambda-Conversion," *Annals of Mathematics Studies 6,* Princeton. **R**

Copi, Irving M., Elgot, Calvin C. and Wright, Jesse B. (1958), "Realization of events by logical nets," *JACM 5,* no. 2, 181–196 (April 1958). **F**

Courant, Richard and Robbins, H. (1941), *What is Mathematics*? Oxford Press.

Davis, Martin (1958), *Computability and Unsolvability,* McGraw-Hill. **R, T, D**

Davis, Martin (1965), *The Undecidable,* Raven Press, Hewlett, N.Y. **R, T, D**

Dertouzos, Michael (1965), *Threshold Logic: A Synthesis Approach.* Research Monograph no. 32, MIT Press, Cambridge. **F**

Eggan, L. C. (1963), "Transition graphs and the star-height of regular events," *Mich. Math. J. 10,* 385–397. **F, A**

Evey, J. (1963), "*The theory and applications of pushdown-store machines,*" Doctoral Thesis, *Report NSF-10,* Harvard University. **M**

Farber, D. J., Griswold, R. E. and Polonsky, I. P. (1964), "SNOBOL: a string-manipulation language," *JACM 11,* 21–30 (Jan. 1964). **L**

Feigenbaum, Edward and Feldman, Julian (1963), *Computers and Thought,* McGraw-Hill, 1963. **L, I**

Fischer, Patrick C. (1965), "Generation of Primes by a one-dimensional real-time iterative array," *JACM 12,* 388–394 (July 1965). **M**

Fischer, Patrick C. (1965), "On Formalisms for Turing Machines," *JACM 12,* no. 4, 570–580 (October 1965). **T**

Freudenthal, Hans (1960), *LINCOS: design of a language for cosmic intercourse,* North-Holland. **I, G**

Gilbert, E. N. (1954), "Lattice theoretic properties of frontal switching functions," *Journal of Math. and Physics, 33,* 57–67 (April 1954). **F**

Gödel, Kurt (1931), "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte fur Mathematik und Physik 38,* 173–198. **D**

Guzmán, Adolfo and McIntosh, H. (1966), "CONVERT," *Comm. ACM 9,* no. 8, 604–615 (August 1966). **L**

Haines, Leonard (1965), "*Generation and recognition of formal languages,*" Doctoral Thesis, MIT. **G**

Haring, Donald (1960), "*The sequential transmission expression for flow graphs,*" Technical Memo. Electronic Systems Laboratory, MIT (November 1960). **F**

Hartmanis, J., Lewis, P. M. and Stearns, R. E. (1965), "Classifications of computations by time and memory requirements," *IFIP International Congress, 1,* 31–35, Spartan Books. **H, M**

Hartmanis, J. and Stearns, R. E. (1965), "On the computational complexity of algorithms," *Trans. Amer. Math. Soc.* (to be published). **H, M**

Hebb, Donald O. (1949), *The Organization of Behavior,* Wiley. **N, I**

Hennie, Frederick C. (1961), *Iterative Arrays of Logical Circuits,* MIT Press, Cambridge. **M**

Hooper, Philip (1964), "*The undecidability of the Turing machine immortality problem,*" Doctoral Thesis, Harvard University (April 1965). **D**

Hubel, David H. and Wiesel, T. N. (1959), "Receptive Fields of Single Neurons in the Cat's Striate Cortex," *Journal of Physiology, 148,* 574–591. **N**

Ikeno, N. (1958), "A 6-symbol 10-state Universal Turing Machine," *Proc. Inst. of Elec. Communications,* Tokyo. **T**

Kahr, Andrew S., Moore, Edward F. and Wang, Hao (1962), "Entscheidungsproblem reduced to the AEA case," *Proc. Natl. Acad. Science, 48,* 3, 365–377 (March 1962). **D**

Kahr, Andrew S. (1963), "Improved reductions of the Entscheidungsproblem to subclasses of AEA formulas," *Mathematical Theory of Automata,* 57–70, Polytechnic Press, Brooklyn. **D**

Kleene, Stephen C. (1936), "General recursive functions of natural numbers," *Math. Annalen 112,* 340–353. **R**

Kleene, Stephen C. (1952), *Introduction to Metamathematics,* Van Nostrand, Princeton. **R, D, T**

Kleene, Stephen C. (1956), "Representation of events in nerve nets and finite automata," *Automata Studies (Annals of Mathematics Studies, no. 34)*, Princeton. **F, A**

Krohn, Kenneth B. and Rhodes, J. L. (1963), "Algebraic theory of machines," *Mathematical Theory of Automata*, Polytechnic Press, Brooklyn, 341–384. **F, A**

Lee, Chester (1963), "The construction of a self-describing Turing Machine," *Mathematical Theory of Automata*, 155–164, Polytechnic Press, Brooklyn. **T, R**

de Leeuw, Karl, Moore, E. F., Shannon, C. E., and Shapiro, N. (1956), "Computability by Probabilistic Machine," *Automata Studies*, Princeton, 183–212. **P, T**

Lettvin, Jerome Y., Maturana, H. R., McCulloch, W. S. and Pitts, W. (1959), "What the Frog's Eye tells the Frog's Brain," *Proc. IRE 47*, 1940–1959. (reprinted in McCulloch [1965]). **N**

Lottka, Alfred J. (1956), *Elements of Mathematical Biology*, Dover. (Original title: *Elements of Physical Biology*.)

Markov, A. A. (1958), "On the inversion complexity of a system of functions," *JACM, 5*, no. 4, 331–334 (October 1958). **F, A**

Mason, Samuel and Zimmerman, H. (1960), *Electronic Circuits, Signals and Systems*, John Wiley. **F**

McCarthy, John (1956), "The inversion of functions defined by Turing Machines," *Automata Studies*, 177–181, Princeton. **I**

McCarthy, John (1960), "Recursive functions of symbolic expressions," *Comm. ACM, 3*, 184–195 (April 1960) **C, L**

McCarthy, John (1961), "A basis for a mathematical theory of computation," Proc. Western Joint Computer Conference (May 1961). **C, L**

McCarthy, John, *et al.* (1962), *The LISP 1.5 Programmer's Manual*, MIT Press, Cambridge. **L**

McCarthy, John (1963), "A basis for a mathematical theory of computation," (revised and extended version of McCarthy [1961]) *Computer Programming and Formal Systems*, Braffort and Hirschberg (Eds.), North-Holland, Amsterdam, 33–70. **C, L, I**

McCarthy, John and Painter, J. (1967), "Correctness of a compiler for arithmetic expressions," *Proceedings of a Symposium on Mathematical Aspects of Computer Science* (April 1966), American Mathematical Society, 1967 (to be published). **C, G**

McCulloch, Warren S., and Pitts, Walter (1943), "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics 5*, 115–133 (reprinted in McCulloch [1965]). **F, N, G**

McCulloch, Warren S. (1959), "Agathe Tyche: of nervous nets—the lucky reckoners," *Mechanization of Thought Processes 2*, 611–634, Her Majesty's Stationery Office, London (reprinted in McCulloch [1965]). **P**

McCulloch, Warren S. (1960), "The reliability of biological systems," *Self-Organizing Systems*, 264–281, Pergamon Press. **P**

McCulloch, Warren S. (1965), *Embodiments of Mind*, MIT Press, Cambridge. **F, N, I, P, G**

McNaughton, Robert and Yamada, H. (1960), "Regular expressions and state graphs for automata," *Transactions of the IRE Professional Group on Electronic Computers, EC-9*, no. 1, 39–47 (March 1960). **F, A**

McNaughton, R. (1961), "The theory of automata, a survey," *Advances in Computers 2*, 379–421, Academic Press. **G, M, F, A**

Minsky, Marvin L. (1956), "Some universal elements for finite automata," *Automata Studies*, Princeton. **F**

Minsky, Marvin L. (1959), "Some methods of Heuristic Programming and Artificial Intelligence," *Proc. Symposium on the Mechanization of Intelligence*. HMSO, London, 3–36. **I**

Minsky, Marvin L. (1961), "Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines," *Annals of Math. 74*, 437–454. **R, T, D**

Minsky, Marvin L. (1962), "Size and structure of universal Turing machines using Tag Systems," *Recursive Function Theory, Symposia in Pure Mathematics 5, Amer. Math. Soc.* **T**

Minsky, Marvin L. (1963), "Steps toward Artificial Intelligence," *Computers and Thought*, Feigenbaum and Feldman (Eds.), McGraw-Hill. Reprinted from *Proc. IRE* (January 1961). **I**

Minsky, Marvin L. (1965), "Matter, Mind and Models," Proc. IFIP Congress, Vol. I, Spartan Books, 45–50. **I**

Minsky, Marvin L. and Papert, S. *Perceptrons: Threshold Function Geometry*, MIT Press, Cambridge (in preparation). **C**

Moore, Edward F. and Shannon, Claude E. (1956), "Reliable circuits using less reliable relays," *Journal of the Franklin Institute 262*, 191–208, 291–297. **P**

Moore, Edward F. (1956), "Gedanken-Experiments on Sequential Machines," *Automata Studies* C. E. Shannon and J. McCarthy (Eds.), Princeton, 129–153. **F**

Moore, Edward F. (1964), *Sequential Machines: selected Papers*, Addison-Wesley. **F, A, M**

Myhill, John (1960), "Linear bounded automata," *WADD Technical Note 60-165*, Wright-Patterson AFB, Ohio. **M**

Newell, Allen, Shaw, J. C. and Simon, H. A. (1956), "The logic theory machine," *IRE Transactions on Information theory IT-2*, no. 3, 61–79. **L**

Ott, G., and Feinstein, N. H. (1961), "Design of sequential machines from their regular expressions," *JACM 8*, no. 4, 585–600 (October 1961). **F**

Papert, Seymour and McNaughton, R. (1966), "On topological events," *Theory of Automata*, Univ. of Mich. Engrg. Summer Conferences. **F, A**

Péter, Rózsa (1951), "Rekursive Funktionen," Akadémiai Kiadó, Budapest. **R**

Post, Emil L. (1943), "Formal reductions of the general combinatorial decision problem," *Am. Journal of Math. 65*, 197–268. **R, D**

Post, Emil L. (1946), "A variant of a recursively unsolvable problem," *Bull. Amer. Math. Soc. 52*, 264–268. **D**

Post, Emil L. (1965), "Absolutely unsolvable problems and relatively undecidable propositions—account of an anticipation," M. Davis, *The Undecidable* (m.s. unpublished, 1941). **D, G, R**

Quine, Willard (1960), *Word and Object*, Wiley.

Rabin, Michael O. and Scott, Dana (1959), "Finite automata and their decision problems," *IBM Journal of Research and Development, 3*, no. 2, 114–125 (April 1959). **F, D**

Rabin, Michael O. (1963), "Probabilistic automata," *Information and Control, 6*, no. 230–245 (September 1963). **F, P**

Rabin, Michael O. (1963), "Real-time computation," *Israel J. Math.*, 203–211. **M**

Rashevsky, Nicholas (1938), *Mathematical Biophysics*, Chicago, rev. ed. Dover (1960). **N, F**

Rashevsky, Nicholas (1940), *Advances and Application of Mathematical Biology*, Univ. of Chicago Press. **F, N**

Robinson, Raphael M. (1948), "Recursion and double recursion," *Am. Math. Soc. 54*, 987–993. **R**

Rogers, Hartley Jr. (1959), "The present theory of Turing machine computability," *J. SIAM 7*, 114–130. **T, R**

Rogers, Hartley Jr. (1966), *Theory of Recursive Functions and Effective Computability*, McGraw-Hill (to be published in 1967). **R, D, H, M, T**

Rosenblatt, Frank (1962), "A comparison of several perceptron models," *Self-Organizing Systems*, Spartan Books, Washington. **N**

Schutzenberger, Marco P. (1963), "On context-free languages and push-down automata," *Information and Control 6*, 246–264. **G, M**

Schutzenberger, Marco P. (1965), "On finite monoids having only trivial subgroups," *Information and Control* 8, 190–194. **A**

Shannon, Claude E. (1948), "A mathematical theory of communication," *Bell System Tech. Journal 27*, 379–423, 623–656. **F**

Shannon, Claude E. (1949), "Synthesis of two-terminal switching-circuits," *Bell System Tech. Journal 28*, 59–98 (January 1949). **F**

Shannon, Claude E. (1956), "A universal Turing machine with two internal states," *Automata Studies* (*Annals of Math. Studies 34*), Princeton. **I**

Shepherdson, J. C. (1959), "The reduction of two-way automata to one-way automata," *IBM J. of Research and Development 3*, no. 2, 198–200 (April 1959). **F**

Shepherdson, J. C. and Sturgis, H. E. (1963), "Computability of recursive functions," *J. Assoc. Comp. Mach. 10*, 217–255. **R, T**

Smullyan, Raymond (1962), *Theory of Formal Systems*, Princeton. **R, D**

Solomonoff, Ray J. (1964), "A formal theory of inductive inference," *Information and Control 7*, no. 1, 1–22; vol. 7, no. 2, 224–254, March–June, 1964. **I**

Tarski, Alfred (1951), *A Decision Method for Elementary Algebra and Geometry*, Univ. of California Press, Berkeley. **D**

Teitelman, Warren (1966), "*PILOT, a step toward man-computer symbiosis*," Ph.D. Thesis, MIT. **L**

Turing, Alan M. (1936), "On computable numbers, with an application to the Entscheidungsproblem," *Proc. London Math. Soc., Ser. 2-42*, 230–265. **T, R, D**

Turing, Alan M. (1950), "Computing Machinery and Intelligence," *Mind 59* (n.s. 236) 433–460, also *The World of Mathematics 4*, Simon and Schuster, 1954. **I**

von Neumann, John (1956), "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, Princeton, 43–98. **F, P**

Wang, Hao (1957), "A variant to Turing's theory of computing machines," *JACM 4*, no. 1. **R, T**

Watanabe, Shigeru (1960), "On a minimal Universal Turing Machine," *MCB Report*, Tokyo. **T**

Watanabe, Shigeru (1963), "Periodicity of Post's system of Tag," *Math. theory of Automata*, Polytechnic Press, Brooklyn, 83–99. **R**

Yamada, H. (1962), "Real-time computation and recursive functions not real-time computable," *IRE Trans. on Electronic Computers EC-11*, 753-760. **M**

Yngve, H. (1962), "COMIT as an Information Retrieval language," *Comm. ACM 5*, no. 1, 19–28. **L**

Yngve, H. (1963), "COMIT," *Comm. ACM 6*, no. 3, 83–84 (March 1963). **L**

TABLE OF SPECIAL SYMBOLS

INDEX AND GLOSSARY

A

B

C

D

E

F

G

H

I

J

K

L

Mc

M

N

O

P

Q

R

S

T

U

V

W

Y

Z