S.A. Dec. 83

COMPUTER RECREATIONS

On the finite-state machine, a minimal model of mousetraps, ribosomes and the human soul

by Brian Hayes

The most powerful computers have neither hardware nor software; they are built out of pure thought stuff. Among these abstract machines the most celebrated is the one invented in 1936 by the British mathematician Alan Mathison Turing. It can do more than any computer made of mere silicon ever could; indeed, it can compute anything that can be computed. A related class of conceptual computers lack the omnipotence of the Turing machine, but they are no less interesting. They are called finite-state machines or finite-state automata, and they establish the minimum specifications of a working digital computer.

Properly defining a finite-state machine calls for a degree of mathematical rigor that is not appropriate here. The nature of the concept can be made clear, however, by means of a few examples. When I went out looking for finite-state machines, I found an excellent specimen in a station of the Lexington Avenue subway in New York. It is a turnstile, an old one made not with the compact steel tripod of current practice but with four oak crossarms, worn smooth by a river of hands and hips.

The turnstile has two states: locked and unlocked. Suppose it is in the locked state, so that the arms cannot be turned. Putting a token into the slot alters the internal mechanism in some way that allows the arms to move; in other words, the token induces a transition to the unlocked state. Rotating the arms by 90 degrees causes another transition that restores the turnstile to the locked state. The transitions are shown schematically in the upper illustration on the next page. The states of the system are represented by nodes (boxes) and the transitions by arcs (arrows) between them.

In the finite-state analysis of the turnstile, inserting a token and pushing on the arms are the possible inputs to the system. The response of the machine depends both on the input and on the state at the time of the input. Pushing on the crossarm when the turnstile has not yet received a token will not get you a ride on the subway. Inserting a token when the arms are already unlocked is also futile, although in a slightly different way. The second token is accepted, but it has no effect on the state of the machine; one person is admitted and then the turnstile locks again. Three or four tokens in sequence are likewise accepted but buy only one ride. Skeptics may want further evidence before accepting the generalization that all tokens after the first have no effect, but they will have to supply their own tokens.

The reason the turnstile cannot give multiple rides for multiple tokens is that it has no means of counting the tokens it has received. Its only form of memory is a rudimentary one: by changing from one state to the other it "remembers" whether the most recent input was a token or a push on the crossarms. All earlier inputs are lost. It is worth noting that this forgetfulness can never work to the disadvantage of the city. It could be worse: a turnstile could be designed to change state after every token, regardless of the present state, in which case two tokens in a row would admit no one.

The turnstile illustrates most of the essential properties of a finite-state machine. Obviously the machine must have some states, and there can be only a finite number of them. There can be inputs and outputs associated with any state. The states must be discrete, or clearly distinguishable, and the transitions between them must be effectively instantaneous. In these matters much depends on the point of view: day and night are discrete states if one is willing to define sunrise and sunset as instantaneous processes. The set of states, the inputs and the outputs constitute the entire machine; there can be no auxiliary devices, and in particular no facilities for the storage of information.

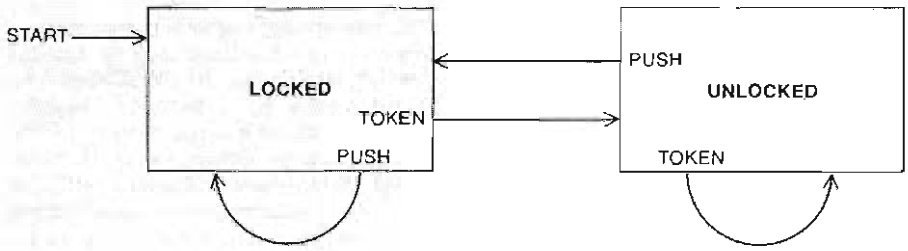
The rules for building a finite-state machine allow some scope for variation. There are deterministic and nondeterministic machines, Moore machines and Mealy machines. In a deterministic machine a given input in a given state invariably has the same result; in a nondeterministic machine there can be several possible transitions. In the Moore machine (named for Edward F. Moore) each state has a unique output. In the Mealy machine (named for G. H. Mealy) the outputs are associated with the transitions rather than the states. It turns out, however, that the variety of architectures is something of an illusion. Any task that can be done by one kind of finite-state machine can be done by the other kinds as well, although the number of states needed may vary. Here I shall discuss mainly deterministic Moore machines, which have the simplest structure.

When you start looking for finite-state machines, you find them everywhere. Coin-operated devices are favorite textbook examples. Some vending machines are less rapacious than the subway turnstile: once they have received the proper amount of money they enter a state in which all additional coins are rejected. The coin-operated device with the largest number of possible states is surely the Las Vegas slot machine. In principle it is deterministic, but finding an input (a coin and a pull on the handle) that will cause a transition to a particular final state is nonetheless challenging.

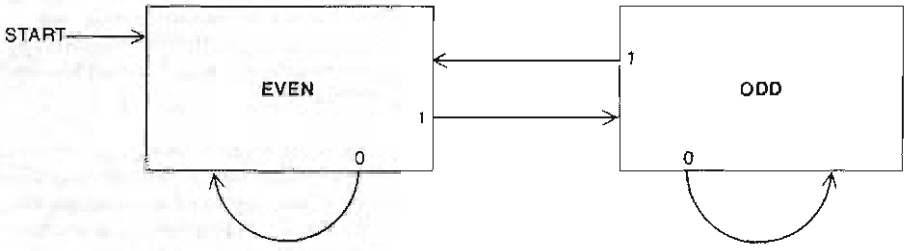
Many household appliances can be regarded as finite-state machines, although they tend to be rather dull ones. A clothes washer goes through an inflexible sequence of states—filling, agitating, rinsing, spinning—and the few meaningful inputs, such as pulling the plug out of the electric outlet, generally have the same effect in all the states. Similarly, a traffic light has a small repertory of states, which repeat indefinitely. To me the most boring of all finite-state machines is a digital clock. If it displays the month, the date and the passage of hours, minutes and seconds, it has some 31 million states; in the course of a year it visits each state exactly once.

A mousetrap is a finite-state machine; the mouse, usually to its misfortune, triggers a transition from the cocked state to the sprung state. A combination lock is a finite-state machine with many possible inputs, only one of which causes a state transition. A telephone has states that might be labeled on hook, off hook, waiting, dial tone, dialing, ringing, connected and out of order. An automobile can demonstrate vividly that the effect of an input varies according to the present state of the system. What happens when you press the accelerator pedal to the floor? It depends. Is the engine running? Is the clutch engaged? Is the parking brake off? Is the transmission in gear? Is it in forward or reverse? Is the garage door open?

In the living cell the molecular system made up of the ribosome and the vari-



A state-transition diagram for a subway turnstile



The parity-testing machine

ous species of transfer RNA operates as a finite-state machine. The inputs are the four nucleotide bases of messenger RNA, designated by the abbreviations *U, A, G* and *C*. The outputs are the 20 amino acid components of proteins. A chain of nucleotides is recognized as a valid input to the machine only if it begins with the "start" signal *AUG*. Thereafter the machine reads the input stream continuously, changing state as each codon, or triplet of nucleotides, is recognized. The three special codons *UAA, UAG* and *UGA* are "stop" signals: when one of them is encountered, the machine halts. Many other biological systems can usefully be represented as finite-state machines; examples that come to mind are the hemoglobin molecule and the promoter and repressor proteins of bacteria.

In the theology of Thomas Aquinas the soul is a finite-state machine, a wonderfully elaborate and fully deterministic one. It is created in a state of jeopardy, as a consequence of original sin. On baptism it enters a state of grace, but certain acts (idolatry, blasphemy, adultery and so forth) induce a transition to a state of sin. Confession, repentance and absolution are then needed to restore the soul to grace. The effect of a final input, death, depends critically on the state of the soul at the moment of death: in a state of grace death leads to salvation but in a state of sin it leads to damnation. The soul machine is actually more complicated than this description suggests. A full account would have to distinguish among the various grades of sin (venial and mortal, actual and habitual) and would have to include other possible states of the soul (such as those associated with limbo and purgatory) and other possible inputs (such as the Last Judgment).

In quantum mechanics even the atom becomes a finite-state machine, and hence so does everything that is made up of atoms. The states of the atom are the allowed energy levels; the inputs and outputs are photons, or quanta of electromagnetic radiation. In a precise description I think the atom would be classified as a nondeterministic Mealy machine with epsilon transitions. It is nondeterministic because the effect of an input cannot be predicted with certainty. It is a Mealy machine because the nature of the output (namely the energy of the photon) is determined by the transition, not by the state entered. Epsilon transitions are those that can take place in the absence of any input; they must be included in the model because an atom can emit a photon and change its state spontaneously.

Is the brain a finite-state machine? As it happens, the modern study of finite-state systems began with a model of neural networks introduced in 1943 by Warren S. McCulloch and Walter Pitts. The neurons of McCulloch and Pitts were simple cells with excitatory and inhibitory inputs; each cell had a single output and two internal states: firing and not firing. The cells could be arranged in networks to carry out various logic functions, including the "and," "or" and "not" functions that are now commonplace elements of electronic logic systems. The equivalence of the idealized neural networks to state-transition diagrams of the kind shown here was established in 1956 by Stephen C. Kleene of the University of Wisconsin at Madison.

Forty years after the work of McCulloch and Pitts it is still subject to dispute whether the brain can reasonably be classified as a finite-state system. Of course the number of neurons is necessarily finite, but that is not the only issue. A real neuron is far more complicated than a two-state cell, and some of its properties may vary over a continuous range rather than being constrained to occupy discrete states. Furthermore, the prohibition of auxiliary information storage in a finite-state model of the brain is awkward at best. If mental life is no more than a succession of instantaneous states, without knowledge of its own history, then what is memory?

The states of mind discussed in psychology, such as boredom, fear, thirst, ecstasy and grief, seem to fit more readily into the apparatus of a finite-state theory. On the other hand, the states are so numerous and the transitions are so poorly understood that the model is useless. Only for lower animals is it possible to draw more than isolated fragments of the state-transition diagram, and in those species the experimenter can have no direct access to the presumed mental states. Indeed, much work of this kind has been done by behaviorists who deny the very existence of mental states.

The case of the digital computer—and here I mean the tangible machine, the hardware—is also problematic. The common mental model of a computer, formulated by John von Neumann, divides the machine into a central processing unit and an array of memory cells. There is no doubt that the finite-state concept can be applied to the various components of the central processor, such as registers, adders and the control mechanism that directs the internal operations of the processor.

The trouble begins when the memory is taken into account. Under the rules for building a finite-state machine no external memory is allowed, and so each cell must be viewed not as a storage facility separate from the processor but as a part of the overall machine state. If all the cells are blank, the computer is in one state; if a single cell is filled, another state is entered, and so on. This conception of the computer is singularly unilluminating, in part because it makes no connection between the state of the machine and what it is doing. Moreover, the number of states is immense. Even a computer of trivial size (100 binary elements), running continuously throughout the age of the universe, could not possibly have worked through all its states.

The primary role of the finite-state machine in computer science is at a higher level of abstraction than the clockwork mechanisms of the hardware. A computer running under the direction of a program is no longer an assemblage of logic gates, registers, memory cells and other electronic paraphernalia; it is a "virtual" machine whose working parts are defined by the program and can be redefined as neces-

sary. Whereas the hardware knows only binary integers and simple commands for moving and manipulating them, the virtual computer deals with far more expressive symbol systems: words, equations, arrays, functions, vectors, codons, lists, images, perhaps even ideas. Finite-state techniques can be valuable in creating the virtual computer, and sometimes the virtual computer *is* a finite-state machine.

Consider a program whose object is to read a series of binary digits (1's and 0's) and report whether the number of 1's received is even or odd. (The task has practical significance; for example, such parity-checking programs are employed to detect errors when digital data are transmitted by telephone.) The program can be constructed as a finite-state machine with two states, as is shown in the lower illustration on the opposite page. Operation begins in the even state, because initially no 1's have been received and 0 is considered an even number. Each 1 in the input stream causes a change of state, whereas a 0 received in either state leaves the state unchanged. Even though the machine cannot "remember" any inputs before the most recent one and certainly cannot count the 1's or 0's, its output always reflects the parity of the input stream.

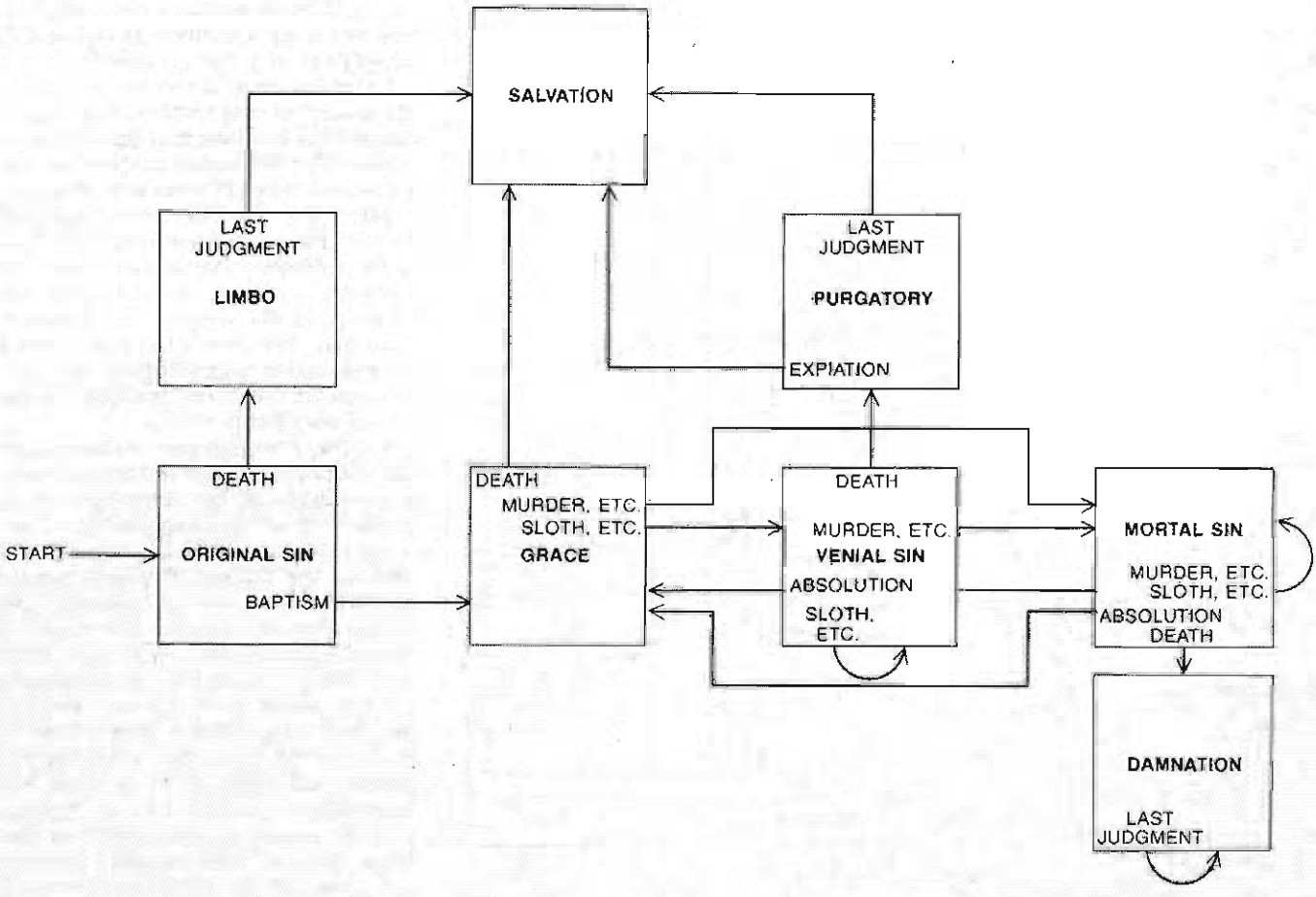
The finite-state model of computation is commonest in programs that deal in some way with text or other information that takes a linguistic form. The preeminent example is found in compilers: programs that translate programming statements in a source language into equivalent statements in a target language, most often the "machine language" of a particular computer. Compilers and other translating programs are essential to the notion of the virtual machine; they mediate between symbols with human meaning and those recognized by the computer.

The part of a compiler that can be designed as a finite-state machine is called the lexical scanner. Like the subway turnstile, it is a token-gobbling device. In this case, however, the tokens are the words, or fundamental lexical units, of the language. The scanner examines each group of characters and determines whether it is a genuine token, such as a command or a number; if it is not, the scanner rejects it as nonsense, just as the turnstile would reject a slug.

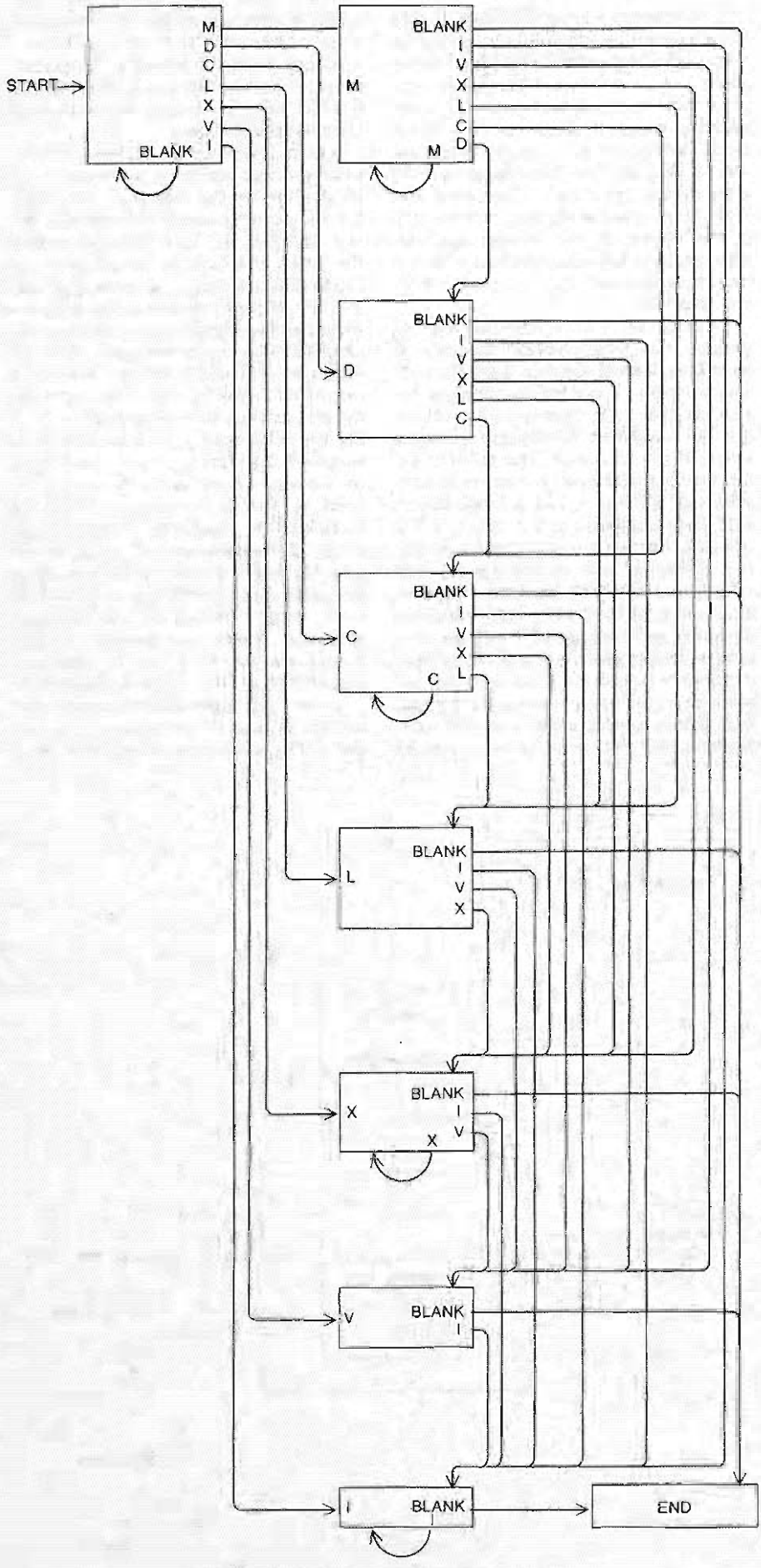
The operation of a lexical scanner can be illustrated by a finite-state machine designed to recognize the tokens of a simple language, albeit one of limited expressive range: the tokens consist exclusively of Roman numerals. Indeed, only Roman numerals of a special form are accepted; they must be given in strict additive notation, so that 9 is represented by VIIII rather than by IX. (There is evidence that the Romans themselves employed the additive notation; the subtractive form is thought to have been a German innovation.)

A state-transition diagram for the Roman-numeral machine is shown in the illustration on the next page. Its alphabet of input symbols includes the letters M, D, C, L, X, V and I as well as the space symbol, or blank. Any initial blanks are simply ignored, but once the first letter is received the program makes an immediate transition to a state identified (for convenience) by the name of the letter. If the first letter is an M, it can be followed by any character from the allowed set, including another M. If the next character is a D, however, the situation is different. From the D state no transition back to the M state is defined, because any series of symbols that includes DM cannot be a well-formed token in the language of additive Roman numerals. Furthermore, there is no transition from the D state to the D state itself, so that DD is also an excluded sequence. (The reason is that the "half value" symbols D, L and V cannot be repeated in proper Roman numerals.)

In the D state the only recognized letters are the lower-valued ones C, L, X, V and I. The same set is accepted in the C



States of the soul in the theology of Thomas Aquinas



A lexical scanner for a language of Roman numerals

state (because C can be repeated), but in the L state only the letters X, V and I are recognized. The rule governing the transitions should be clear. The states are arranged in a hierarchy, and once a given level has been reached the machine can never return to a higher level; in the half-value levels it cannot even remain at the same level. By the time the I state is reached only an additional I or a blank is allowed. The blank, entered at this point or at any other time after the first letter, indicates the end of the token and sends the machine back to its starting state, ready to receive the next Roman numeral.

No programming language known to me allows numbers to be entered in Roman form, but virtually all such languages have facilities for handling Arabic numbers. The techniques for recognition are similar, although there is a greater variety of formats. Simple integers such as 137 can be handled in principle by a one-state machine, but the several parts of a number such as $+6.625 \times 10^{-27}$ require a more elaborate lexical analysis.

The ribosome-transfer-RNA system can be regarded as a lexical scanner that recognizes biologically meaningful nucleotide sequences in a molecule of messenger RNA. To be accepted a sequence must begin with a start codon and end with one of the three stop codons; between these boundaries any combination of the input symbols *U*, *A*, *G* and *C*, taken three at a time, is allowed.

Lexical analysis is only the first step in the process of compilation. The components of the compiler that are called into action after the lexical scanner are the parser and the code generator. The parser takes as its input the tokens identified by the scanner and analyzes their syntactic relations; this is the closest the compiler comes to understanding the meaning of the program statements it translates. The code generator writes a program in the target language that carries out the functions specified by the parsed statements.

For the toy languages considered here the tasks of the parser and the code generator are trivial. The compiled form of a statement in the Roman-numeral language might be simply the Arabic equivalent of the number. It could be generated by the following strategy. Before a token is scanned a storage cell is specified and is set equal to zero. Then each time the scanner enters the M state 1,000 is added to the value in the cell; for the D state 500 is added, and so on. When the scanning is complete, the memory cell holds the value of the Roman numeral. Note that the toy compiler is no longer a pure finite-state machine, because it has auxiliary storage.

A compiler for the genetic code is even simpler and can be realized entire-

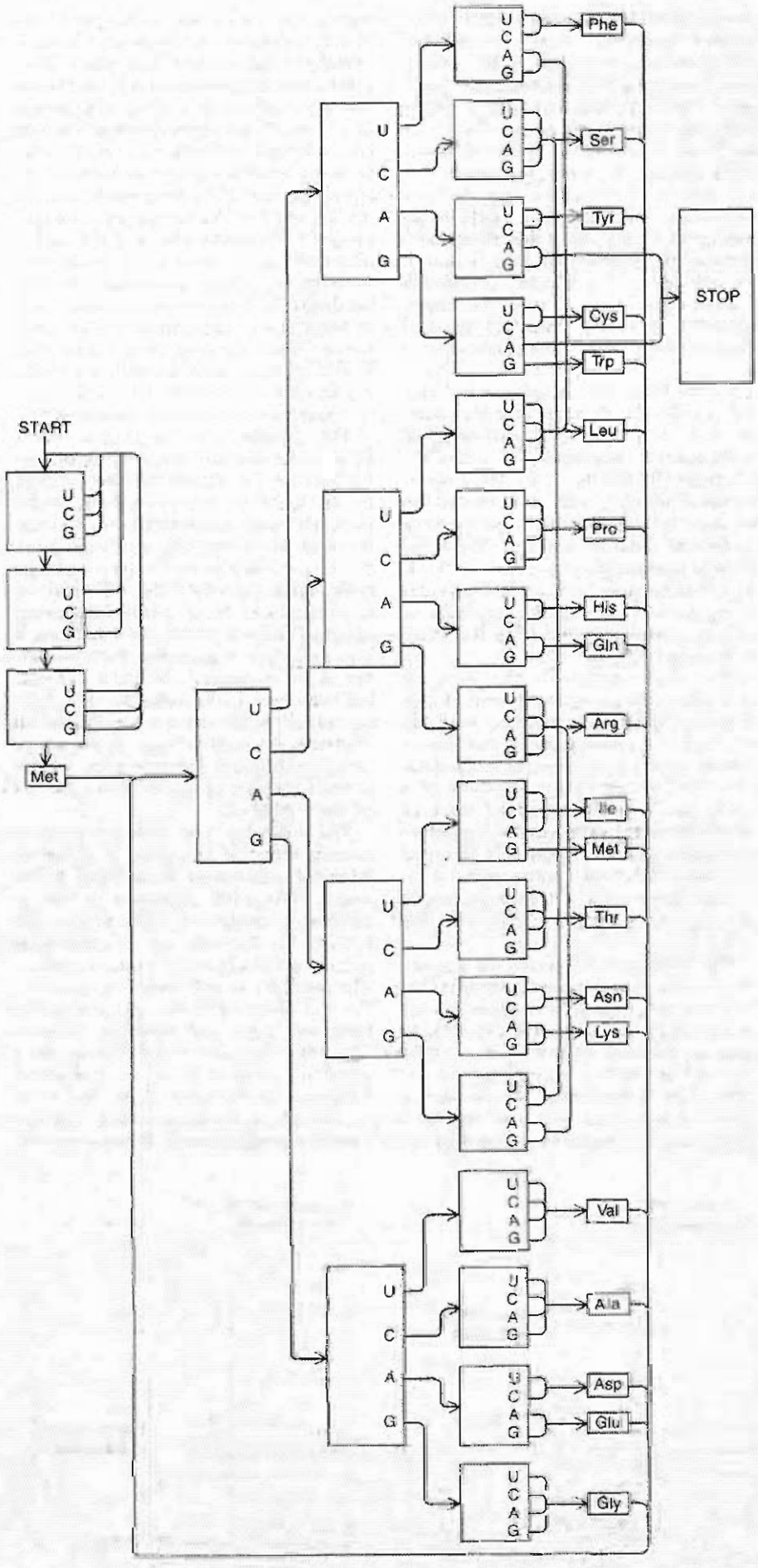
ly within the context of a finite-state system. The compiled "program" is a sequence of the standard three-letter symbols for amino acids; the symbols can be generated as the output of the states of the scanner that recognize codons. The three states corresponding to stop codons have no output.

Creating a compiler for a language large enough to be of general utility is not a casual undertaking, but the underlying architecture of the finite-state machine can at least provide an organizing principle. If the syntax of the language is specified with sufficient precision, part of the work can even be mechanized: it can be done by a compiler compiler, a program whose input is a formal description of a language and whose output is another program that translates statements in the language. As far as I know no one has yet written a compiler compiler compiler.

The identification of tokens by a lexical scanner is in itself a kind of parsing, and the set of all possible sequences of symbols in a token is a kind of language. Indeed, it is an infinite language: unless some artificial limit is put on the length of individual sequences, an infinite variety of recognizable tokens can be formed. How can a machine with only a finite number of parts recognize an infinity of well-formed statements and exclude an infinity of ill-formed ones? The key is in the structure of the language itself. If the statements of an infinite language are to be recognized by a finite-state machine, they must be formed according to strict rules.

The rules were set forth by Kleene in 1956; they define a class of languages called regular languages or regular sets. Kleene proved that a finite-state machine can recognize a language only if it is regular, and further that every regular language can be recognized by some finite-state machine. What is meant by regular can be indicated briefly (although not rigorously) by two rules. First, any finite language is regular and therefore can be recognized by a finite-state machine; after all, one could build a machine with a state for each possible expression of the language. Second, if a language is infinite, it must be possible to parse all its statements by reading one symbol at a time from left to right, or beginning to end, without backtracking or looking ahead. If the acceptability of any symbol is contingent on the presence of another symbol, the governing symbol must be the one immediately to the left.

The second rule is a direct consequence of the limitations of a finite-state machine, which can neither foresee its future states nor keep a record of its past ones; it must choose a state transition based only on the current state and the current input symbol. It is for this



A finite-state machine translates the genetic code into protein

reason that the subtractive notation for Roman numerals cannot be handled by a finite-state machine. If the expression XI is read and the machine interprets it as 11, it cannot go back to revise the value when the next character turns out to be V. Many other functions are ruled out by the same limitation. For example, it is not possible to build a finite-state machine that reads a sequence of binary digits and determines whether the number of 1's is equal to the number of 0's. Similarly, although a finite-state machine can add binary numbers, it cannot multiply them; I leave it to the reader to deduce why.

Beyond finite-state machines and regular languages there extends a hierarchy of more powerful machines and more general languages. It is called the Chomsky hierarchy, after the linguist Noam Chomsky, who investigated the various formal languages as possible models of natural language. The more general languages are created by relaxing constraints on the grammatical rules of regular sets; the machines are built by adding memory elements to the basic finite-state model.

The next machine in the series is called the pushdown automaton. It consists of a finite-state machine with the addition of a memory array that has an infinite capacity but a peculiar organization. The memory takes the form of a stack, like a counterweighted stack of cafeteria trays. An item of information can be stored only by putting it on top of the stack; when the information is retrieved, any overlying items must first be removed. Thus the last item in is the first one out.

The language recognized by a pushdown automaton is called a context-free language. In parsing its statements the acceptability of a symbol can depend both on the symbol immediately to the left and on the one immediately to the right. This bidirectional dependency is permissible because any symbols whose interpretation cannot be decided immediately can be stored on the stack until the ambiguity is resolved. Hence a pushdown automaton can work with subtractive Roman numerals, and it can identify expressions with equal numbers of 1's and 0's (or other symbols, such as left and right parentheses). On the other hand, it cannot detect sentences with equal numbers of three symbols, such as 0's, 1's and 2's. Most programming languages are context-free, and the parser of a compiler is generally a pushdown automaton. Many computers include hardware facilities for organizing a part of the memory capacity as a pushdown stack. One programming language, Forth, makes a stack the primary memory structure. Of course, the stack in any real machine cannot have infinite depth.

The context-free languages merit their name because the parsing of any symbol can be influenced directly only by the symbol's two immediate neighbors, not by the wider context in which it is found. Removing this constraint gives rise to a context-sensitive language and once again increases the difficulty of interpretation. Now widely separated symbols can interact; in the worst case it is not possible to interpret the first symbol in an expression until the last one has been read. In exchange for the added complexity somewhat greater capability is gained. A machine based on a context-sensitive language can determine whether an expression includes equal numbers of three symbols.

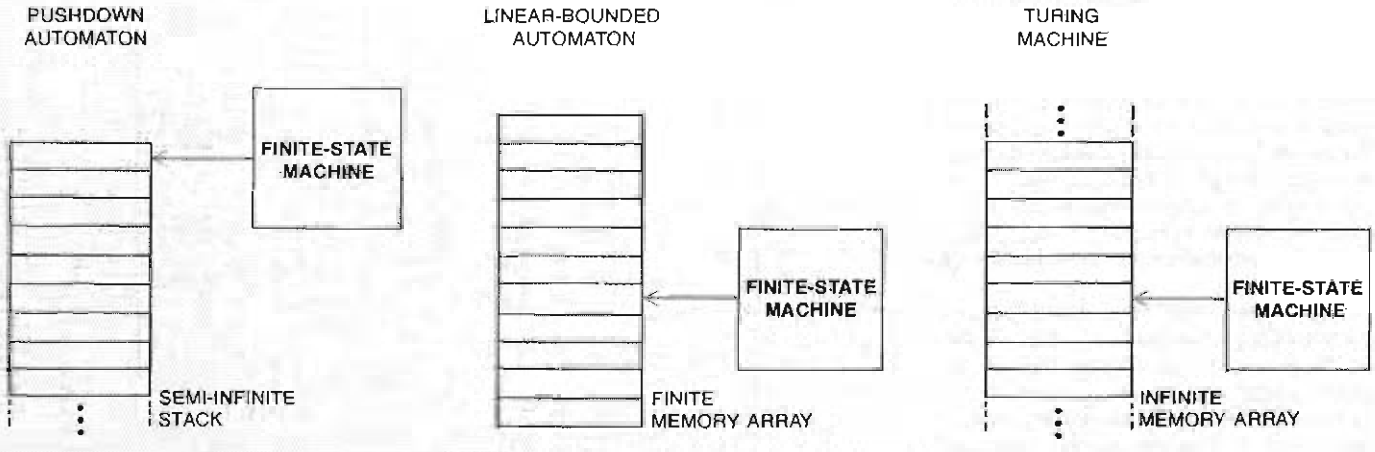
The machine that can recognize a context-sensitive language is a linear-bounded automaton. In addition to the usual finite-state apparatus it has a memory organized in such a way that any storage location can be reached at any time; it is a random-access machine. The memory is only finite in capacity, but it is assumed to be large enough to hold any input the machine receives. The linear-bounded automaton seems a good approximation of the von Neumann model of a digital computer. Oddly, though, the corresponding context-sensitive programming languages seem to be rare; evidently the simpler context-free structure almost always has sufficient expressive power.

All the languages described above have a property in common: they are said to be recursive. What this designation amounts to is that one can imagine a procedure for generating all possible "utterances" in the language in order of increasing length. It follows that there is a guaranteed method of deciding whether any given statement of finite length is a member of the language: simply generate all the statements up to that length and compare them.

There are languages that cannot meet even this minimal standard of tractability. For them there is only one possible recognizing machine: it is the computer of last resort, the Turing machine, a finite-state automaton allowed to roam freely through an unbounded memory. In the description given by Turing the memory is a tape, infinite in both directions and marked off into cells, which the finite-state apparatus can write on, read or erase.

Looking down from the elevated perspective of the Turing machine, the relations of the lesser computing devices become clearer. The linear-bounded automaton is simply a Turing machine with a finite tape. The pushdown automaton has a tape that is infinite in one direction, but the "head" for reading and writing on the tape always remains fixed over the last nonblank cell. The finite-state machine is a Turing machine with no tape at all.

Brand-name-conscious readers, eager to parse nonrecursive languages, may already be out shopping for a Turing machine. They should be warned that the ultimate computer also has its weaknesses. There are languages with grammars so preposterous that even a Turing machine cannot be counted on to recognize their statements in a finite amount of time. So far such languages have found little use in the world of computing machines, but people somehow manage to speak them.



The Chomsky hierarchy of finite and infinite machines