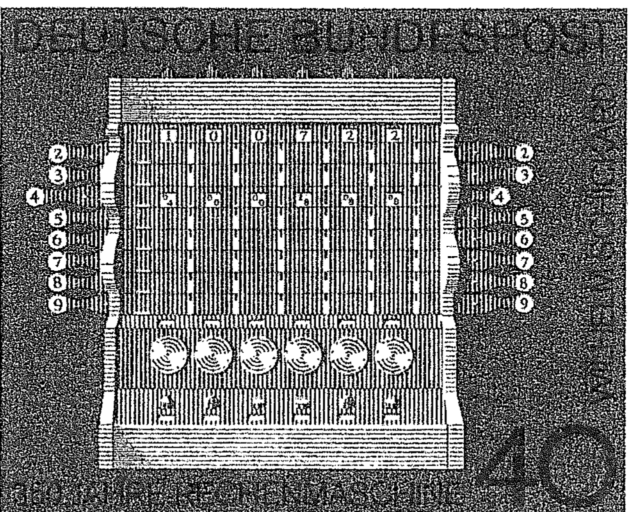


Postage stamp depicting Schickard's calculator on its 350th anniversary

From Artificial Intelligence by J. Hangeland. (c) 1985 by MIT Press. Permission to reprint granted by the publisher.

4 Computer Architecture

#1

Babbage and His Analytical Engine

People too often assume that computers are all essentially alike and then they can imagine only one basic design for Artificial Intelligence (or the mind). But computers differ dramatically from one another, even in their fundamental structure and organization. In this chapter we will sample several different general *architectures*, savoring just enough of each to bring out its own special flavor. The point is not to canvass candidate psychological arrangements—we're not yet contending for that lofty honor—but rather to illustrate how much machines can vary and to see what some of the theory is like.

As far as we know, the automatic digital calculator was invented around 1623 by an obscure German astronomer named Wilhelm Schickard (1592–1635). I say "as far as we know" because nobody knew about Schickard's invention until recently, and who knows what else may still be lost. The first publicized calculator was built some twenty years later by the French philosopher Blaise Pascal (1623–1662), after whom a popular programming language is now named. His device caused quite a stir, even though it could only add and subtract, and only clumsily at that. Predictable improvements followed in due course—most notably multiplication and division, introduced in the 1670s by the German philosopher Gottfried Leibnitz (1646–1716)—but nothing very exciting happened for a couple of centuries.[1]

England's Charles Babbage (1792–1871), history's first computer scientist, was also a classic eccentric genius—notorious, among other things, for an obsessive hatred of organ grinders. Trained as a mathematician, and for nine years Lucasian professor of

mathematics (Newton's chair at Cambridge), Babbage never actually taught a course, or even moved to the university from London. Intellectually, he was as versatile as he was pioneering, making original contributions not only in mathematics, but also in precision metalworking (his shop was among the finest in Europe) and in what we now call operations research (he wrote one of its earliest systematic treatments). But, above all, Babbage is remembered for his calculating engines.

In fact, he designed two quite different machines. The first (ca. 1823–1833), called the *Difference Engine,* is not terrifically important, though it was certainly innovative and potentially useful. Many components and even prototypes were built, but no Difference Engine was ever actually completed, for several reasons. First, critical tolerances made construction very expensive (hence the fine machine shop); second, Babbage kept improving the design, requiring constant modification and rebuilding; and third, sometime around 1833 his imagination was seized by a more elegant and far grander scheme.

The new *Analytical Engine* occupied Babbage for the rest of his life and was his crowning achievement. Its design incorporates *two* utterly profound and unprecedented ideas, which together are the foundation of all computer science:

1. its operations are fully *programmable;* and
2. the programs can contain *conditional branches.*

Like the Difference Engine (and for similar reasons), the Analytical Engine was never finished, although nearly a thousand technical drawings, timing diagrams, and flow charts were prepared, along with six or seven thousand pages of explanatory notes. Moreover, quite a number of parts were made, and a substantial model was under construction when Babbage died (at age seventy-nine).[2]

For many years, however, the only published account of the new Engine was a little "memoir" with an engaging history of its own. In 1842 L.F. Menabrea, an Italian engineer, published (in French, but in a Swiss journal) a summary of some lectures

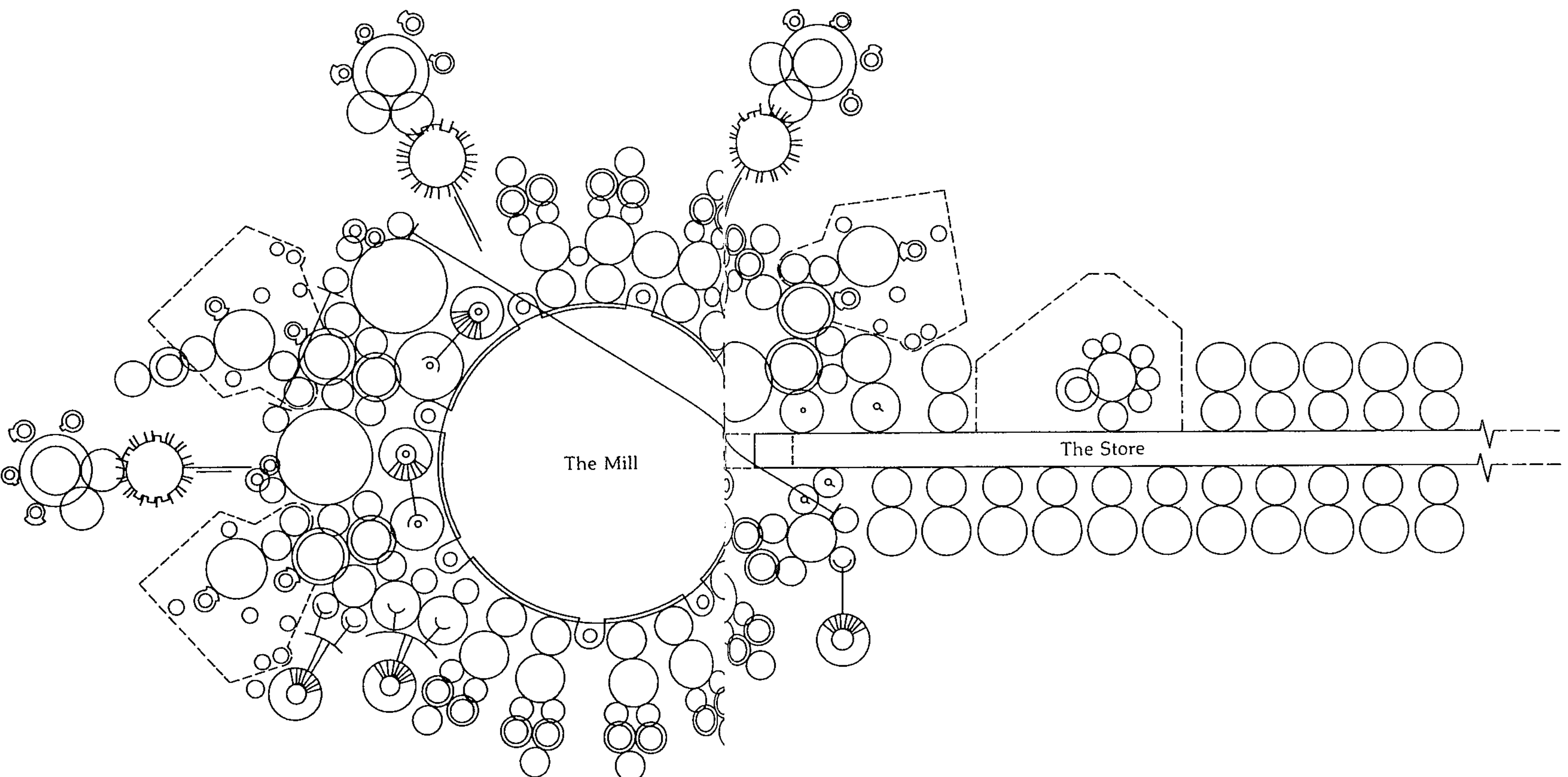
Babbage had given privately in Naples. The following year Ada Augusta, a gifted young protégé of Babbage's (and later Countess of Lovelace), translated it into English—adding, with the master's encouragement, a few explanatory footnotes. These "notes" turn out to be three times longer and considerably more sophisticated than the original article. Though Babbage probably supplied some of her material, it seems certain that Lady Lovelace understood his work better than did anyone else in his lifetime; the powerful new language *Ada* has been named in her honor.[3]

The Analytical Engine has three major components: the *mill* (an arithmetic unit), the *store* (data memory); and a sequencer/controller unit, for which Babbage provided no name. The tokens manipulated in the system are signed numerals, and the playing field is essentially the store. The mill can perform the four standard arithmetic operations, using any indicated locations in the store for operands and results. Thus, using our earlier terms, we can think of the mill as a team of four rather limited "inner players," one for each primitive ability. And the controller, then, is the inner referee, never manipulating tokens itself, but only telling the players when and where to exercise their respective talents; that is, it calls on them, one at a time, saying which numerals to work with and where to put the answer. Every move is fully determined.

But what "game" does the Analytical Engine play? What are the rules? What determines the instructions that the referee issues to the players? These are all the same question, and the answer is the key to the whole brilliant concept. Within certain limits, the Analytical Engine will play whatever game you tell it to. That is, you specify the rules (in a particular manner), and the referee will see that the players abide by them. So just by handing the referee different "definitions," you can make the Analytical Engine be any automatic formal system you want (within its limits).

This remarkable exercise, *making* a specific automatic system, simply *by describing* it appropriately for a general purpose system, is called *programming.* It is one of the most powerful ideas in the history of technology, and Babbage invented it all by himself.

Compare an ordinary calculator. It can perform the same four operations, but you have to enter each command manually, each



Analytical Engine, top view

step of the way. If you want to evaluate $3(x + y) - 5$, for given x and y, you must first add the two data, then multiply their sum by three, and finally subtract five from the product. If you need the value for various givens, you must go through the whole sequence, step by step, every time. A calculator is essentially just four independent special-purpose devices, each of which can do only one thing; any combination or repetition of operations must be done by hand. One could, of course, build a more complicated special-purpose machine to compute not the sum or product of two numbers, but (say) five less than triple their sum; but that would be weird and ugly.

A much better plan would be a machine for which you could prespecify *any arbitrary sequence* of basic operations (on any specified variables). Such a machine could evaluate any formula you like, given the appropriate specifications; and, given those specifications, it would act exactly like a special-purpose machine custom made for that formula. For all intents and purposes, it would *be* that special machine—until the specifications were changed. Needless to say, this is how the Analytical Engine works: the "specifications" are the program.

The second great invention in the Analytical Engine is the *conditional branch*: a directive to the referee to move to another part of the program, depending on the result of some test. We have already seen the importance of conditional branches, in our key/lock algorithms; in fact, Babbage's programs are essentially branched schedules, just like those algorithms. Such branches are required for many numerical calculations, as Babbage was well aware (see box 1 for an example). Conditional control is crucial also in other general symbol manipulators, including those that matter in Artificial Intelligence.

Inasmuch as the Analytical Engine is an automatic formal system, it is strictly medium independent. But 150 years ago physical realization was a major obstacle, and Babbage's amazing design is worth a brief description. Numerals are forty decimal (not binary) digits, plus a sign. (He seems to have had several different ideas for keeping track of fractions, but none settled.) Each digit is represented by the rotation of a brass gear, roughly the size of a

Box 1
Sample Program for the Analytical Engine

The user programming facility is the least well defined aspect of Babbage's design. The following is a rough compromise between his informal notation and the format we used for algorithms in chapter 2.

Locations in the store have names: V1, V2, V3, etc. The instruction: "V1 + V2 → V3" means "get the numbers stored in V1 and V2, add them, and put the sum in V3." You can think of it as the referee instructing the addition specialist to "do his thing" using these locations. Directives to the referee are printed in capital letters. The conditionals (IF . . .) test whether a certain number is greater than zero: if so, the referee branches; if not, she just continues to the next instruction. (Note that the branch directions don't tell the referee what line to go to, but rather how many lines to move from where she is.)

The sample program computes the value of: $ar^m - r^n$.

Before starting, load storage locations with initial values as follows:

V1 with *a*, V2 with *r*, V3 with *m*, V4 with *n*, and V5, V6, and V7 all with the constant *1*.

Program	Explanation
START	
V5 x V2 → V5	Multiply V5 by *r* (to be repeated *m* times)
V3 − V7 → V3	Subtract 1 from *m* (i.e., count loops)
IF V3 > 0, GO BACK 2 LINES	Repeat above ("loop") *m* times—compute: r^m
V1 × V5 → V5	Multiply by coefficient *a*

V6 × V2 → V6	Compute: r^n (by repeated multiplying again)
V4 − V7 → V4	
IF V4 > 0, GO BACK 2 LINES	
V5 + V6 → V8	Add the two terms
PRINT V8	Print the final results
STOP	

doughnut, with the forty gears for each numeral stacked a few inches apart on a brass rod about ten feet high.

The mill consists of more than a hundred such columns, arranged in a circular pattern four or five feet across. The store contains another hundred or so columns in a double row extending out to one side of the mill. All of this, of course, is riddled with levers and ratchets, pinions and interlocks, making sure everything turns when (and only when) it should—including sophisticated precautions against wear and backlash. Recent analyses suggest that the parts being manufactured in Babbage's shop were both accurate and strong enough for a working Engine. Completed, the whole thing would have been about the size and weight of a small railway locomotive, though far more complicated and expensive.

Programs for the Analytical Engine are not kept in the store but are encoded on sequences of punched cards, strung together with ribbons. The ribbons enable the machine to back up and repeat a group of instructions over and over, which is essential for implementing loops. This basic arrangement was inspired by a system used to control the brocaded patterns woven by Jacquard looms, and that fact, in turn, inspired Lady Lovelace to a famous simile:

> We may say most aptly, that the Analytical Engine *weaves algebraic patterns* just as the Jacquard-loom weaves flowers and leaves.

And just so does a mind weave general symbolic patterns—if Artificial Intelligence is right about us.

Turing Machines and Universality

Alan Turing (1912–1954), like Babbage an English mathematician (but not so cranky), is prominent in the history of computing on several counts. We have already reviewed the famous "Turing test," which he proposed in 1950 in an influential discussion of machine intelligence. Prior to that, during and immediately after the second world war, he made major contributions to the pioneering British development of electronic computers.[4] And most important, before the war (while still a graduate student, in fact), Turing produced the first mathematically sophisticated theory of computation, including some astonishing and profound discoveries about machine capabilities.[5]

Crucial to these theoretical findings was the invention of a new kind of computer—a new basic architecture—comprising what are now called *Turing machines.* There are infinitely many (possible) specific Turing machines, no one of which is terribly significant by itself; what matters are the overall category of such devices and various general theorems that Turing proved about them.

Before proceeding, we might mention that, like his predecessor, Turing never actually built any of his machines; but the reasons are entirely different. The Analytical Engine is extraordinarily large and complicated, near the limits of what could be understood, let alone constructed, in its time. Turing machines, on the other hand, are amazingly simple; moreover, though they can be arbitrarily large, they are typically fairly small. Beneath this surface contrast lies a deep difference in essential goals. Whereas Babbage intended his Engines to be useful for practical purposes, Turing was interested only in abstract theoretical questions. Therefore Turing's designs are elegant to describe and convenient to prove things about, but utterly impractical to use. Hence building one, though easy, would be pointless.

A Turing machine consists of two parts: a *head* and a *tape* (figure 1). The tape is just a passive storage medium: it is divided along its length into squares, each of which can hold one token (from some prespecified finite alphabet). We assume that the number of tape-squares available is unlimited but that only a finite segment of them is occupied at any one time; that is, all the rest are empty

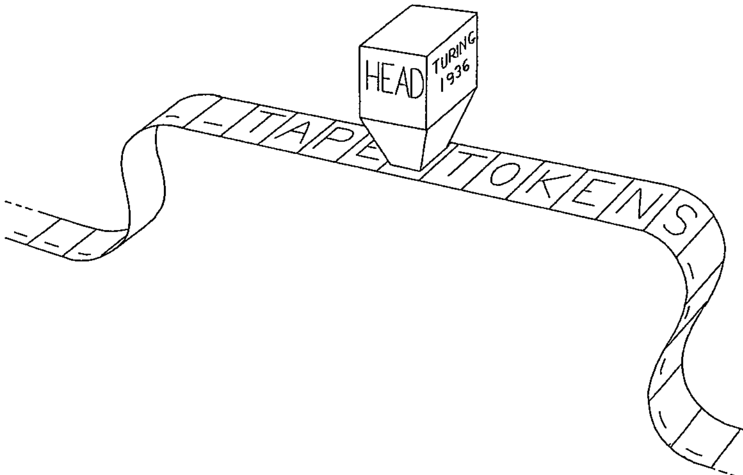


Figure 1 Diagram of a Turing Machine

or contain only a special blank token. The tape is also used for input and output, by writing tokens onto it before the machine is started and then reading what remains after it halts.

The head is the active part of the machine, hopping back and forth along the tape (or pulling the tape back and forth through it) one square at a time, reading and writing tokens as it goes. At any given step, the head is "scanning" some particular square on the tape, and that's the only square it can read from or write to until it steps to another one. Also at each step, the head itself is in some particular *internal state* (from a prespecified finite repertoire of states). This state typically changes, from one step to the next; but under certain conditions, the head will enter a special "halt" state, in which case the machine stops—leaving its output on the tape.

What the head does at any step is fully determined by two factors:

1. the token it finds in the square it is scanning; and
2. the internal state it is currently in.

And these two factors determine each of three consequences:

1. what token to write on the present square (replacing whatever was previously there);
2. what square to scan next (the same one, or the one immediately to the right or left); and
3. what internal state to be in for the next step (or whether to halt).

So the entire functioning of a Turing machine (i.e., of the head) can be specified in a single, two-dimensional chart, with one row for each token that the head might find and one column for each state that it might be in. Each entry in the chart simply specifies the above three actions, as determined by that token and that state. (See box 2 for examples.)

Turing machines are easy to describe as "automatic games." The tape, clearly, is the playing field, and the tokens on it are the game tokens. The various internal states correspond to inner players, each of whom is very narrow. A player can read the current token, and, depending on what she finds, replace that token with another (of the same or a different type), and then tell the referee which way to move the tape, and which player to call on next. The inner referee is even dimmer: he merely starts with player 1 and then moves the tape and calls on players as they themselves request—until somebody says HALT.

With the idea of a Turing machine, we can describe a milestone of mathematical history—a proposal put forward in various ways by Turing and others around 1936. Recall the problem of finite playability: in a formal system, the rules must be "followable by players that are indisputably finite." Unfortunately this condition is somewhat vague; so it's not perfectly clear just which systems or rules satisfy it. In the mid-thirties a number of mathematicians were worried about this question, and several quite different criteria were suggested. Each was gratifyingly rigorous and intuitively plausible; moreover, it was soon established that they are all theoretically equivalent—hence the plausibility of each augments that of all the others. Moreover, every plausible criterion since proposed has also been proven equivalent.

As a result, essentially all mathematicians now accept *Turing's thesis* that these mutually supporting criteria do rigorously capture

Box 2
Two Simple Turing Machines

In the charts below, the states are numbered across the top, and the alphabet is listed down the left. Corresponding to each state and token is either a three-character entry in the chart or the word "HALT." The first character is the token to be written; the second character is an L or an R, for move left or move right; and the third character is the number of the next state.

Example 1 ARTHUR:

	1	2
_	_R2	HALT
A	AL1	BR2
B	BL1	AR2
C	CL1	CR2

ARTHUR has three types in his alphabet (besides the blank, which is indicated by an underscore: _). We assume that he starts in state 1, scanning some square in the midst of a string of A's, B's, and C's. First, he just moves to the left, not changing anything on the tape (state 1). But when he gets to the left end of the string (the first blank), he switches to state 2, and starts moving right, converting all A's to B's and B's to A's (while leaving C's alone). At the right end of the string, he stops.

Example 2 BERTHA:

	1	2	3	4	5	6	7	8	9
_	HALT	HALT	_L5	AR7	_R1	_R1	AR8	AL9	_R1
A	AR2	AR3	AR4	AR3	*L6	AL5	AR8	AL9	AL9
*	*R1	*R2	*R3	*R4	*L5	*L6	AR8	AR9	*L9

BERTHA is more complicated but also more fun to figure out. We assume she starts in state 1, at the left end of a string of A's and asterisks. First, she moves to the right end of the string, not changing anything, but keeping track of A's. If she found one or no A's along the way (states 1 and 2), she halts. Otherwise, if she found an even number of A's (finishing in state 3), she goes back to the beginning, converting every second A to an asterisk (states 5 and 6), and starts over; but if she found an odd number of A's (finishing in state 4), she puts three more A's on the right end of the string (states 4, 7, and 8), before returning and starting over (state 9).

Now check this: if the number of A's in the initial string happens *not* to be a multiple of 3, BERTHA will methodically eliminate all but one of them and halt gracefully; otherwise she will land helplessly in an infinite loop, and never rest.

the intuitive idea of finite playability.[6] There are, of course, as many ways to spell this out as there are explicit criteria; we sketched one in our discussion of algorithms. Another (and more rigorous) formulation is as follows:

TURING'S THESIS: For any deterministic automatic formal system whatever, there exists a formally equivalent Turing machine.

In other words, no automatic formal system can do anything (nonrandom) that Turing machines can't do; in principle, Turing machines are the only automatic systems we ever need. Note that this is called a "thesis" (and not a "theorem") because it is not amenable to proof; strictly, it is a proposal about what "deterministic automatic formal system"—especially the "finite playability" clause—really means.

So far we have not mentioned programming, but some Turing machines can be programmed so as to mimic others. The gist of the idea is to encode a description of the machine being imitated and use that as (part of) the input for the imitator. The latter will

read this description, do a lot of computing, and, from time to time, make a move (in a reserved part of its tape) exactly equivalent to the next move that the former would make. (The equivalence may involve some transliteration, of course, if the two machines don't happen to use the same alphabet.) The encoded description of the first machine is then functioning as a program for the second.

Suppose, for instance, that the tape of some machine, say MOM, has squares tinted alternately pink and light blue. (These are only for our convenience; MOM herself is color-blind.) We then encode the description of some other machine—call it BABY—into a sequence of pink squares only. This description is essentially just BABY's two-dimensional chart (as discussed above), stretched out into a single line, with a little punctuation thrown in to avoid confusion. So exactly what BABY would do for any given state and scanned token is *explicitly listed* somewhere in this sequence on MOM's pink squares. The blue squares are reserved as a surrogate for BABY's own tape. Now all MOM has to do is keep track of BABY's successive tape positions and internal states (she uses some extra pink squares as a scratch pad for this purpose), read the corresponding blue square, look up BABY's response in her list, and then go do it.

Of course, we have yet to show that any machines can really be such MOMs. Turing, however, did even better: he proved that some Turing machines are *SUPER-MOMs*—or, in his more decorous terms, *universal machines*:

TURING'S PROOF: There exist universal Turing machines that can mimic, move for move, any other Turing machine whatever.

If you have just one universal Turing machine (and divine patience), then, in principle, you never need any other Turing machine for any other purpose. For, suppose there were some other Turing machine you needed: you could just take its standard description (chart), transliterate the alphabet, code it up for your SUPER-MOM, and then pay attention to only her blue squares. The moves being made on those squares are *exactly* the moves

(as transliterated) that the other machine would have made on its tape—so, in effect, you have that other machine already.

Needless to say, Turing's thesis and Turing's proof complement one another splendidly. According to the former, Turing machines are the only automatic formal systems we ever need; and, according to the latter, we only ever need one Turing machine—a universal one. Put these ideas together, and our one SUPER-MOM can do the work of all possible (deterministic) automatic formal systems. You might suppose that such powerful, general-purpose systems would have to be quite complicated; but that's not true (see box 3

Box 3
Minsky's Universal Turing Machine

"Most" Turing machines (including, e.g., those in box 2 are *not* universal. Still, there remain indefinitely many designs that are universal, and the question arises: Which design is "simplest"? Defining 'simple' is, of course, hard; but for Turing machines a widely accepted measure is the product of the number of internal states times the number of token types in the alphabet. (This is because there are easy technical tricks for reducing either, at the expense of increasing the other.) By this measure, the simplest universal Turing machine known was discovered by an American computer scientist, Marvin Minsky (1927–). It has four symbols (including the blank) and seven states—for a winning product of twenty-eight. Here it is.

	1	2	3	4	5	6	7
Y	_L1	_L1	YL3	YL4	YR5	YR6	_R7
_	_L1	YR2	HALT	YR5	YL3	AL3	YR6
1	1L2	AR2	AL3	1L7	AR5	AR6	1R7
A	1L1	YR6	1L4	1L4	1R5	1R6	_R2

Not surprisingly, the proof that this astonishingly simple chart defines a truly universal machine is itself fairly difficult and lengthy; the reader is referred to Minsky (1967, section 14.8) for the details. I present it here purely as a work of art.

for a specimen). What they are is *slow*. The encoded description of a Turing machine (e.g., BABY) is really just a transcription of a number of primitive rules into a very regimented form. So all MOM needs is a few basic abilities enabling her to decode and follow arbitrary rules in this specific form—and lots of time.

Since Turing's original proof, it has been discovered that there are many other possible architectures for universal formal systems, including, in fact, Babbage's. In the remainder of this chapter, we will look at three more universal programmable architectures, each quite different from the others (and from Turing machines) and each widely used in practice today.

The Ubiquitous von Neumann Machine

Although John von Neumann (1903–1957) was born and educated in Hungary, he moved to the United States as a young man, which nicely symbolizes several transitions in our story. So far our main characters have been Europeans, but from now on they'll be Americans; moreover, von Neumann's own interest in computers blossomed at about the same time as U.S. domination of the field, basically after World War II. This period also saw the earliest *practical* general purpose computers, a development in which von Neumann (and the U.S.) played a major role. Finally, those were the days when computer science itself began to mushroom, with dozens and soon hundreds of people all working more or less together. As a result, associating particular ideas with particular individuals becomes more tenuous and difficult. Babbage, for instance, was truly a lonesome hero, whose ideas are his alone; and Turing machines are wholly the invention of one man, even though several contemporaneous systems turned out to be mathematically equivalent. What is now called "the von Neumann architecture," on the other hand, actually evolved over some years, with numerous individual contributions that are not always clearly separable. This trend will become more evident as we proceed.

Von Neumann, like most computer pioneers, was trained in mathematics; but unlike the others, he always remained primarily a mathematician. (Though he did manage to squeeze in some brilliant, original work in quantum mechanics and economics—the famous "theory of games"—along the way.) His involvement

with computers began almost accidentally, as a consultant for the ENIAC project, which, at the time, was easily the largest computer in the world. Reflection, by von Neumann and others, on ENIAC's weaknesses culminated in the IAS (Institute for Advanced Study) computer, which, though smaller, was more powerful and much easier to program and which proved to be the basis of many subsequent designs.

Crucial to a von Neumann machine is a large memory that can be accessed in either of two different ways: relatively or absolutely. *Accessing* memory is finding a particular location in it, so that any token there can be read and/or replaced. The two modes of access are analogous to two ways of specifying a particular building in a city. *Relative access* is specifying a new location in terms of its relation to the current one. Thus, in giving directions to a stranger, we often say things like: "Go down two blocks, make a left, and it's the second door on your right." That tells the person where the place is from here, that is, relatively. Similarly, in computer memory, you can specify the next location to use (read or write) in terms of its relation to one you've just finished using. For instance, a Turing machine tape is accessible (only) relatively: the next square to be scanned is always specified relative to the present square (it's the same or right next door). Notice that, for relative access, the memory locations must all be organized in some regular way (such as the linear ordering of a Turing tape), and at each moment the machine must be "at" some particular location (e.g., the square now being scanned).

Absolute access (also called *random access*) is specifying a particular location with a unique name (or number). The post office, for example, identifies buildings by unique addresses (1234 Main Street or Cathedral of Learning, Pittsburgh); similarly, phone calls are routed absolutely, with unique phone numbers. Absolute access does not require any organized structure or "current" location, as long as each possible target can be reached reliably by name. The store of Babbage's Analytical Engine, for instance, is accessible (only) absolutely; the variable columns all have individual names (which happen to be numerals), but they are not related to one another in any usable manner.

The main memory of a von Neumann machine is not only accessible in both these ways, but is also used for two quite different jobs; and this is surprisingly important. First, the memory holds the tokens to be operated upon: initial data (if any), intermediate values, and then the final results, ready for output. Thus if the system were programmed to sort lists alphabetically, the memory would first be loaded with the original list, subsequently contain partially alphabetized lists, and end up holding the final list. This is just like Babbage's store, except that the latter was designed primarily for numerical data and didn't allow relative access. Relative access is useful if the data itself is structured (like a list or an array): the machine can go (absolutely) to the beginning of the data and then move through it item by item (relatively).

The other use of the same main memory is to hold the program that the machine is executing. Von Neumann programs are branched schedules, basically like Babbage programs, except that the latter were encoded on strings of punched cards rather than held in the store with the variables. Those cards could be accessed (only) relatively; that is, the string could be advanced or backed up by so many cards (e.g., for repetitive loops), but there was no way to access an arbitrary card by name. A von Neumann machine also generally accesses its program relatively, either step by step in a straight schedule or back and forth by specified numbers of steps (in loops).

Where the von Neumann architecture really shines, however, is in the use of absolute access to implement subroutines. A *subroutine* is a "mini-program" that a larger program can invoke at any point *as if it were a single step*. Suppose, for instance, that your main program (a big record-keeping system, say) needs to alphabetize various lists at various times. One possibility is to insert a complete alphabetizing routine into the program at each point where it might be needed; but that's clearly wasteful. A much better strategy is to include only one copy of the routine, at some agreed location where it can always be found. Whenever the main program needs to alphabetize something, it just branches over to this handy subroutine, which then branches back when it's done.

The trick, of course, lies in these two branches, especially the second. Since the subroutine itself remains at a fixed (absolute) location but gets *called* (branched to) from all different parts of the program, it is most easily called by name (address). Just as important, when the routine is finished, it must *return* (branch back) to wherever it was called from so that execution can resume where it left off. But this raises a delicate question: how does the subroutine know where to branch back to, given that it might have been called from anywhere? The answer is to use another prearranged location to store the return address. Then whenever the program calls a subroutine, it puts its own current address in this special place before branching. When the subroutine is finished, it automatically gets the program address from that prearranged place and uses it to branch back (absolutely). (See box 4 for further details.)

Babbage's Analytical Engine cannot use this clever subroutine technique, for two different reasons. First, it can access portions of its program (the strings of cards) only relatively; but the above subroutine branches use absolute access. Second, the return addresses have to be different on different occasions (because the subroutines can be called from different places in the program). That means they have to be alterable (rewritable) by the machine during execution of the program. But Babbage's cards were permanently punched; programs could be changed only by punching new cards and stringing them together again manually. Other early systems (e.g., the ENIAC) were like Babbage's in this regard. But a von Neumann machine can read, write, and use *addresses* (not just data) that are stored in main memory and that, therefore, are easily altered by the machine itself while the program is running. Consequently, absolute subroutine calls (and returns) are feasible and convenient; in fact this is the primary architectural advance of von Neumann's design.[7]

When Babbage and Turing machines are described as automatic formal games, the role of the inner referee is fairly trivial. But von Neumann machines have a more elaborate structure, and the referee therefore has more to do. His job is to tell the inner players whose turn it is and which tokens they can play with—that is, which primitive operations to carry out when and what they

Box 4
The Return-Address Stack

The return-from-subroutine problem is subtler than it looks. In the first place, if there is only one prearranged location to store the return address, then only one subroutine can be called at a time; in particular, one subroutine couldn't in turn call a further "sub-subroutine," which would often be very convenient. A partial solution (and the standard approach in von Neumann's day) is to set aside a separate location for the return address for each subroutine; then each can find its way back independently of the others.

This is still limiting, however, because any given subroutine can still be called only once at a time; that is, a subroutine cannot call *itself* as a sub-subroutine. (That may sound like a bizarre idea, but it's actually quite feasible and useful; compare the discussion of *recursion* in the next section.) A general solution of the problem (invented in the mid-fifties by Allen Newell and Cliff Shaw) is to use a special kind of memory, called a *stack*. This is a "last-in-first-out" memory, like a stack of letters on your desk that you access only from the top. Thus every time you get a letter that needs to be ansered, you throw it on top of the stack; and every time you get a chance to answer letters, you take them from the top of the stack. For correspondence this may not be a wonderful method, because the poor guy who wrote you first gets answered last. But it has an important advantage for subroutine returns.

Here's how it works. Whenever a portion of your program calls a subroutine, it first puts its return address on the top of the stack and then branches. Whenever any subroutine finishes its job, it removes whatever address is on top of the stack and branches back there. The result is that any subroutine always returns automatically to the most recent call that has not yet been answered. Since each return address is removed from the stack as it is used, the return address that remains on top of the stack (if any) is automatically the one put there previously for the subroutine that called the subroutine that just finished. Thus subroutine calls can be *nested* (subroutines calling sub-subroutines and so on) arbitrarily and without restriction. (Notice that the stack memory itself uses a particular sort of relative access.)

should operate on. Since branching directives (including subroutine calls and returns) control the sequence of execution, they are the province of the referee. In other words, branches are not operations (token manipulations) that players perform, but shifts that the referee makes in determining which player to call on next. In these terms, then, it is the referee's business to keep track of all subroutine calls and returns and to see that execution resumes where it left off; hence the special return-address storage (the "stack") is actually a private little memory belonging exclusively to the referee.

Subroutines are important not only because they are efficient, but also because they allow programs to be more *modular.* That is, once a subroutine for a particular task has been written and "debugged" (corrected and made to work right), then it can be used at any time without further ado. As a result, the problem of writing a large program, composed mainly of lots of different subroutines, can be broken down into a number of smaller and more manageable subproblems—namely, getting each of the various subroutines to do what it's supposed to do separately. Furthermore, once useful subroutines have been written and debugged for one program, they can then be borrowed wholesale for use in other programs, without the headache of writing and debugging them all over again.

This idea is in turn the foundation of another, which is perhaps the backbone of all modern programming: *higher-level machines* (so-called "programming languages"). A general selection of widely useful subroutines can be collected together to form a sort of "library," which any programmer can then draw upon whenever necessary; it saves a lot of work. Then all you need is an automatic librarian to take care of the calls and other miscellaneous bookkeeping, and the subroutines themselves are tantamount to new

primitive operations, any of which can be invoked with a single instruction (they become new "players" in a fancier game). In effect, you have a new and more powerful computer. This is the essence of such familiar systems as BASIC, Pascal, FORTRAN, COBOL, etc.

These particular *virtual machines* ("languages") all happen to be roughly von Neumann in architecture: their programs are all branched schedules, with loops, named subroutines, and so on. But that's just a coincidence. It's equally possible to write subroutines and librarian/referees that give the new machine quite a different architecture. For instance, one could easily write a von Neumann machine program that exactly simulated a Babbage or Turing machine. Then, for all practical purposes, that new machine would be the one you worked with—writing programs for it, etc.—if, for some odd reason, you wanted to. This is precisely the situation we encountered earlier with SUPER-MOM Turing machines; in fact, von Neumann machines (with unlimited memory) are also universal machines, in exactly the sense that (some) Turing machines are.

Von Neumann universality, however, is far more important (in the real world) than Turing universality. For one thing, von Neumann machines are much easier to program; for another, they are much faster—they don't have to search back and forth along the tape, one square at a time. And it turns out that von Neumann architectures are easy to build out of basic electronic components. This last point is less a comparison with Turing machines (which are almost trivial to build) than with certain other universal architectures, which, though easy to program and fast enough, are hard to build. The upshot is that virtually all (programmable) computers in current production have a basically von Neumann architecture at the hardware level.

In the next two sections, we will look at two quite different architectures that are widely used in Artificial Intelligence work. Theoretically, of course, it doesn't matter how they are built or implemented as long as they always make legal moves in legal positions (medium independence again); but it gives some pause to remember that (at present anyway) they are all actually being simulated on von Neumann SUPER-MOMs.

McCarthy's LISP

In our survey of computer architects, John McCarthy (1927–) is the first native American and the last trained mathematician. Professionally, however, his interests soon focused on computer automata; and by 1956 he had coined the term "Artificial Intelligence" and had organized the fledgling field's very first scientific conference.[8] In 1958 he and Marvin Minsky founded the AI Lab at MIT and in 1963 he set up another out at Stanford. Both are now world centers of AI research. In the meantime, he also developed the first "time-sharing" operating systems—the ingenious (and now common) arrangement by which one central computer can simultaneously support many independent users.

Our topic, however, is McCarthy's invention, in 1959, of a whole new kind of computer, named *LISP*.[9] McCarthy machines—as LISP systems might also be called—have a simple and elegant structure that is remarkably flexible and at the same time remarkably different from either Turing or von Neumann machines. Compared to these predecessors, LISP is distinctive in two main respects: its *memory organization* and its *control structure*. Though neither development is difficult to understand, each is peculiar enough to merit some explanation.

Memory Imagine a long line of tin cups strung on a chain and a plentiful supply of atomic tokens to put in the cups, one token per cup. Obviously it's a "memory" device. It works just like a Turing machine tape, if we assume that cups can be specified only by relative position and that the head can move down the chain only one cup at a time. On the other hand, if each cup also has a unique label ("address") and if the processor can jump any distance in a single step, then we have a von Neumann memory.[10] Notice that the two memories are organized just alike and differ only in the additional access mode of the von Neumann machine; in particular, the structure of each is:

1. LINEAR: the chain has no forks or junctions;
2. UNITARY: there is only one chain; and
3. FIXED in advance: the chain is always the same.

Are these structural features essential to computer memory? Not at all.

Imagine, in place of the tin cups, a huge stock of Y-connectors, piled loose in a heap. A *Y-connector* is a Y-shaped device with a base or plug on one end and two sockets set at angles on the other. One common variety allows you to put two light bulbs in a single fixture; and there are different kinds for joining garden hoses, electric cords, and so on. The important point is that the base of one Y-connector can be plugged or screwed into a socket on another; and this process can continue as long as you please, building up bizarre Y-connector *trees*. Each such tree will have one exposed base at the bottom (its *root*)[11] and any number of empty sockets around the top and nestled among the limbs. So just imagine labels on the roots of all the trees and atomic tokens plugged into all their empty sockets (like "leaves"), and you have the basic idea of LISP memory.

The Y-connectors (joints) in a LISP tree are called *nodes*. The atomic symbols in a tree's terminal sockets are also nodes (specifically *terminal* nodes); in fact, an atomic token all by itself can be a single-node tree (like a seedling, with one leaf and no branches). No socket in a LISP tree can ever be empty; but there is a special atomic token, *NIL*, that serves as a "blank" or "nothing" when necessary. One critical deviation from our analogy is that LISP nodes are not symmetrical: the branches (sockets) are explicitly designated "left" and "right," respectively. Hence any node in any tree can be specified by a combination of absolute and relative access: name the tree and then give a path—a sequence of left and right turns—to that node from the root.

Distinguishing left and right branches also makes it possible to define *lists* as special trees built as follows: take as many Y-connectors as there are items to be listed, line them up in a row and plug each into the right socket of its predecessor, plug the list *elements* into all the left sockets (in order, starting from the base), and plug NIL into the last right socket. (See figure 2.)

The contrasts with cup-chain memories are striking. LISP trees clearly aren't linear (though lists can do duty for linear structures, whenever necessary). Moreover, since many trees can be alive at once, the overall structure isn't unitary either. But, most important,

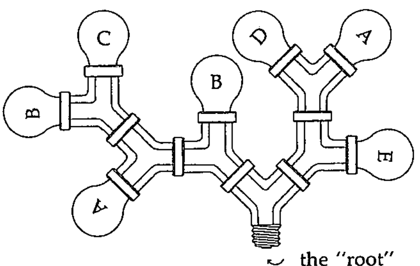
the "shapes" of LISP trees aren't fixed in advance: they are built to order as needed (and thrown back on the heap when no longer needed). This has two subtle but significant consequences. First, since trees are not all the same shape but vary from case to case, the shapes themselves must be part of what the system "remembers." Thus cup-chain memories store only the contents of the cups; but LISP trees store not only their terminal tokens, but also the particular connections among all their nodes. Hence they are especially convenient for keeping track of complex structural relationships. Second, since the trees are built to order as the program runs, there's no need to know ahead of time exactly what they'll be like or even how big they'll grow; such details can be resolved on the spot, as the situation develops. Consequently, LISP programs can be extraordinarily flexible in response to widely differing conditions.

Control To explain the distinctive control structure of LISP, we must shift analogies. *Control* is determining who does what when. Consider serving me breakfast: chilled orange juice and two fried eggs, please. Basic cooking skills ("primitive chefs") are not the issue here, but rather how they might be organized. In other words, we're mainly interested in the executive chef, the kitchen referee. In von Neumann's kitchen, the referee calls on specialists according to a schedule:

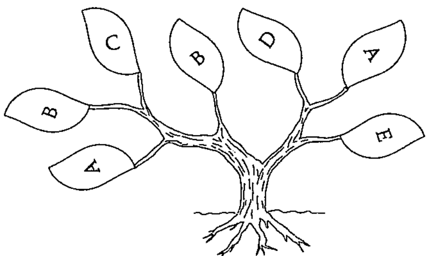
ORANGE
JUICE that orange;
CHILL that juiced stuff;
EGG
EGG
CRACK those eggs;
SKILLET
FRY those cracked things in that skillet;
SERVE that chilled stuff and those fried things.

Capital letters indicate which specialist to call on: the first line activates ORANGE, the orange-getting specialist; the second line then tells JUICE, the juicing specialist, to do his thing (on whatever

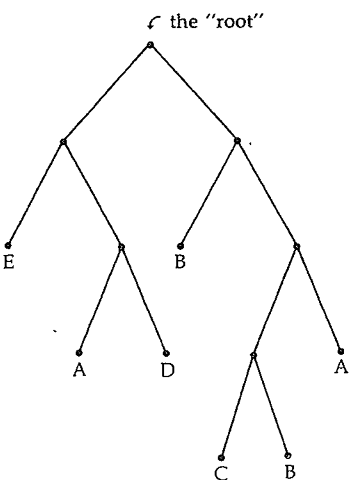
Figure 2 Examples of LISP Trees



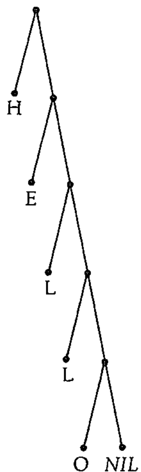
This is a simple Y-connector tree.



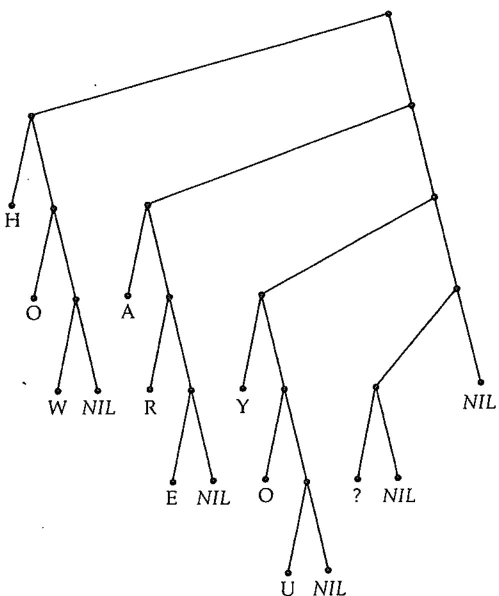
Here's the same tree, looking somewhat more botanical.



Here it is again, as a computer scientist would draw it. (For dark reasons of their own, computer scientists always draw trees upside down.)



This is a special sort of tree, called a "list." (The NIL just plugs up the last right socket; it's not one of the items listed.)



This is also a list; but the items listed happen to be lists themselves—so we have a list of lists.

ORANGE got); and so on. Notice that the routine is basically linear and that its order reflects priority in time.

What happens in McCarthy's kitchen? The same primitive actions are performed in the same order; but they are arranged, so to speak, from the opposite end. The McCarthy referee always starts from a required result and works back to prerequisites. Thus she begins by telling SERVE to serve chilled orange juice and two fried eggs and then waits to see if he needs anything. Predictably, he does: namely, orange juice from CHILL and two eggs from FRY. So she accepts these as new (subsidiary) requirements and calls in turn on CHILL and FRY—waiting now to see what they might need (orange from JUICE, eggs from CRACK, etc.).

So far everybody's just passing the buck, and nothing is actually getting done. But when (at JUICE's behest) ORANGE finally gets called on, the buck stops: ORANGE is a simple stockboy who requires no preparations; he just gets an orange and hands it over. When that happens, however, the referee can start fulfilling prior requests: thus, remembering who wanted the orange, she gives it to JUICE, who can now perform and return the orange juice. Then, remembering who wanted that, she passes it to CHILL, and so on, until, eventually, SERVE has everything he requested and can bring ME breakfast. This is the essence of LISP control.

Notice that the structure is not basically linear but hierarchical and that the primary ordering is not by time but by results and prerequisites: when a specialist is engaged, he can't deliver his product without first engaging others to prepare his materials. Two dimensions show it best:

```
        ( CHILL  {  JUICE    {  ORANGE
SERVE  <
        |        (  CRACK    {  EGG
        ( FRY   <               EGG
                 (  SKILLET
```

The order of engaging is left to right, and the order of delivery is then right to left; vertical columns just list the engaged preparers, when there's more than one.

In mathematical terminology, specialists who interact in this way are called *functions*, their prerequisites (if any) are called *arguments*, and the results they return are called *values*. Functions that require no arguments are *constants*; they always return the same value. Mathematicians allow the arguments and values of functions to be anything at all, as long as the following essential condition is met:

FUNCTIONS: For each allowable argument (or combination of arguments), there is *one and only one* (permanently assigned) value.

For instance, MOTHER[x] is a perfectly acceptable function (taking people as arguments) because each person has exactly one (permanent) mother. But UNCLE[x] is not an acceptable function because some people have no uncles and others have many. LISP primitives (and programs) are all functions, but they differ from their mathematical counterparts in two basic respects: they are active players who automatically evaluate themselves when called upon, and their (uninterpreted) arguments and values must always be LISP trees.

Whenever a LISP function gets an argument, it assumes that the argument is actually specified via another function, which therefore ought to be evaluated first. In the kitchen we called this "passing the buck"; but a symbolic example will illustrate the real point:

THIRD LETTER OF [SECOND WORD OF [THIS BOOK'S TITLE]]

What is the value of that function for that argument? Well, the argument is:

SECOND WORD OF [THIS BOOK'S TITLE]

Is its third letter *C*? No, because the actual argument is not that very expression but rather what it specifies: namely, the *value* of the function *SECOND WORD OF*[ ], when applied to the argument:

THIS BOOK'S TITLE

So is the intended word *BOOK'S* (with third letter *O*)? No, for the same reason: *THIS BOOK'S TITLE* is also a function, specifying a value. Fortunately, however, it's a *constant* function, with no arguments. Therefore, unlike the first two functions, this one can be evaluated straightaway, without evaluating anything else first. (The third letter of the second word of its value is *T*.)

How do you program a LISP machine? Certainly not by writing out a sequential schedule, as for a von Neumann machine. Instead, you program it by *defining* new functions in terms of other functions that are already defined. The general idea is familiar from algebra; for instance, the square of a number can be defined in terms of multiplication:

SQUARE[x] = def TIMES[x,x]

Or, back in the kitchen, a BREAKFAST function could be defined in terms of SERVE, FRY, SKILLET, etc. Defined functions are "management" specialists: like all functions, they return values for given arguments, but they do it entirely by telling the referee who else to call on, making no (other) contribution of their own.

Obviously not all functions can be defined in terms of others; there have to be some primitive functions to start with. Real-world LISP systems provide lots of convenient primitives, including many (naturally interpreted as) standard arithmetic functions, etc. But quintessential LISP, out of which all the rest can be constructed, comprises only six primitive functions:

LEFT[x]	RIGHT[x]	JOIN[x,y]
EQUAL[x,y]	ATOM[x]	IF[x,y,z]

Each of these functions takes only LISP trees as arguments and returns only LISP trees as values (remember that single atomic tokens count as "seedling" trees). With these primitives, it is possible to manipulate (build, modify, destroy) LISP trees of arbitrary complexity, in any algorithmically specifiable way; hence it is possible to define a single complex function that will return any such transform as its value, for any input. LISP is therefore universal.

The first three primitives are used for assembling and disassembling arbitrary trees. The value of LEFT[x] is whatever is screwed into the left socket of the Y-connector at the very base of (the value of) *x*; in effect, it returns the left half of its argument. If the argument doesn't have a left half (because it's only an atomic seedling), then LEFT[x] reports an error and quits. Needless to say, RIGHT[x] is just the same, except that it returns the right half. JOIN[x,y] takes two arguments, and, not surprisingly, puts them together; more specifically, it gets a new Y-connector, plugs (the values of) its first and second arguments into the left and right sockets, respectively, and returns the new tree thus constructed.

The other three functions provide LISP's alternative to testing and branching. EQUAL[x,y] returns the special token TRUE if (the values of) its two arguments are the same; otherwise it returns FALSE. ATOM[x] returns TRUE if (the value of) its argument is a single atomic token; otherwise FALSE. IF[x,y,z] might be clearer if we wrote it as IF[x, (THEN)y, (ELSE)z], because what it does is return the value of either its second or its third argument, depending on whether (the value of) its first argument is TRUE or FALSE. Thus IF[__, __, __] does not "branch" LISP around to different parts of a schedule of instructions (there is no such schedule to branch around in). Rather, it determines which functions actually get evaluated by asking the referee for the value of either its second or its third argument (but not both).

LISP is distinctive in its dynamic, tree-like memory organization and its function-application control scheme. Moreover, these are related in a beautiful inner unity: the hierarchical structure of functions applied to arguments which are themselves functions, etc. is essentially a *tree*. More specifically, LISP represents functions with their arguments as *lists*, with the function name as the first element of the list and the arguments following in order. Since those arguments in turn are function calls, they too will be represented as lists, and so on—making the whole structure a list of lists of lists . . . which is to say, a tree. In other words, LISP programs and LISP memory units ("data structures") have exactly the same underlying form.

Box 5
Recursion and Universality

Inasmuch as LISP machines have the power of *universal* Turing machines—the power, that is, to mimic any other computer—they must be able to repeat a specified process over and over (an arbitrary number of times) until some condition is met. Von Neumann machines accomplish this with loops and conditional branches. McCarthy machines rely instead on *recursive definitions*: in defining a function, that function itself can be used, as long as there's an escape clause that eventually stops the regress. To put it somewhat paradoxically, it's okay for a definition to be "circular" as long as it's not circular forever.[12]

Suppose, for example, you needed to count the atoms in LISP trees. This task is naturally conceived recursively: the atom count for any tree is either 1 (if the whole tree is just a single atom) or else the sum of the atom counts for its left and right halves. So we can define a simple function:

```
COUNTATOMS[x] = def
IF[ ATOM[x],
(THEN) 1,
(ELSE) COUNTATOMS[LEFT[x]] + COUNTATOMS[RIGHT[x]] ]
```

This definition is recursive, because the function COUNTATOMS[—] is not only being defined, but also being used in that very definition.

But notice how it works: if the given tree is atomic, then the function returns 1 immediately; otherwise it calls (repeats) itself for each half of the argument. Moreover, it can keep recalling itself (for ever smaller halves of halves, etc.), an arbitrary number of times. Sooner or later, however, there won't be any smaller halves: every branch ultimately ends with terminal atoms. Then COUNTATOMS[—] does not call itself again but simply returns 1 to its predecessor; and when the predecessors start getting numbers back, they can start returning sums to their own predecessors, until, at last, the original call to COUNTATOMS[—] gets back a value.

What keeps this "circular" definition from being vicious? Two things: first, every time COUNTATOMS[—] turns around and calls itself, it does so with a simpler argument; and second, the arguments eventually get so simple that COUNTATOMS[—] can handle them directly, without calling itself again (that's the necessary escape clause).

Newell's Production Systems

Allen Newell (1927–) considered a career in mathematics, but reconsidered. The Rand Corporation, a high-flying "think tank" out of California, seemed a lot more exciting than graduate school in New Jersey; so off he went.[13] There he met Cliff Shaw, a farsighted computer pioneer (who had joined Rand the same year, 1950), and Herbert Simon, a rising star in economics and management science (who visited from Carnegie Tech in the summer of 1952). By 1955 the three of them had more or less invented general symbol manipulators, heuristic search, and Artificial Intelligence.[14] It was not until the 1960s, however, after Simon had lured him back to academia, that Newell's thoughts turned to *production systems*—the distinctive computers that might fairly be called "Newell machines."[15]

The active players in a production system—the chefs in Newell's kitchen—are called *productions*. Like their Turing and von Neumann counterparts (but unlike LISP functions), productions all act on a common linear memory, called the *workspace*. But they differ from their colleagues in determining who works where when. Each Newell chef constantly surveys the entire workspace, looking for a certain *pattern* of prerequisite ingredients; whenever he finds his particular pattern, he goes there and performs his specialty (with those ingredients). For instance, one modest expert might look exclusively for a pint of whipping cream, flanked by a whisk and a cold copper bowl. Unless and until that appears, he does nothing; but as soon as it does, he rushes over, whips the cream in the bowl, and discards the whisk. Most such actions

modify the patterns in the workspace, which may then "satisfy" some other specialist, such as the cake decorator, who has been waiting patiently for a bowl of whipped cream. And thus the processing continues.

Two things are remarkable about this procedure. In the first place, locations in the workspace are not specified either absolutely (by "name") or relatively (how to get there from "here"); rather, they are identified by their *contents*, that is, in terms of whatever is currently stored there. So if the paradigm of absolute access is a city address (1234 Main Street) and that of relative access is directions (go north six blocks, and east two), then the paradigm of content-based access would be a description of the destination: find a yellow brick split-level with pink curtains, wherever it may be. In a production system, of course, these described contents are nothing other than the patterns for which the various specialists are looking.

Second, nobody tells productions when to act; they wait until conditions are ripe and then *activate themselves*. By contrast, chefs in the other kitchens merely follow orders: Turing units are nominated by their predecessors, von Neumann operations are all prescheduled, and LISP functions are invoked by other functions. Production system teamwork is more laissez-faire: each production acts on its own, when and where its private conditions are satisfied. There is no central control, and individual productions never directly interact. All communication and influence is via patterns in the common workspace—like anonymous "to whom it may concern" notices on a public bulletin board. Accordingly, overall direction emerges only as the net effect of many independent decisions—reminiscent of an open market, guided by Adam Smith's "invisible hand."

Fastidious readers may have noticed a couple of loose ends. What happens, for instance, if two productions find their conditions satisfied at the same time? Assuming they can't both act at once, some arbitration procedure (referee) is needed. The simplest solution is to rank order the players and then always select the petitioner with highest priority. Imagine them all standing in line: the first one checks for his pattern and acts if he can; otherwise the second one gets to check; and so on. After any production is

satisfied and acts, they start over again at the head of the line. In a similar vein, what happens if some chef's pattern actually occurs twice in the workspace? Which one should he use? Again, the easiest solution is a prior ordering: for example, the productions could always scan the workspace left-to-right, stopping at the first satisfactory pattern.[16]

How do you "program" a production system? By *constructing* (defining) productions; the productions themselves are the program. An unprogrammed production system is just an empty workspace, a referee, and a kit of primitives, out of which productions can be built. The programmer's job, then, is to put together a set of productions—each able to perform a certain action in certain circumstances—whose collective behavior will be that of the desired system. The "primitives" in the kit correspond roughly to von Neumann primitive operations or LISP primitive functions, but the way they're assembled into a composite system is entirely different.

Every production has two basic parts: a *condition* and an *action*. The definition of a production is essentially a rule (imperative) that says:

WHENEVER ⟨condition⟩ IS SATISFIED, PERFORM ⟨action⟩

or, more schematically:

⟨condition⟩ → ⟨action⟩

The condition is a pattern that might appear in the workspace, and the action is usually some modification of that pattern (but sometimes it's external output). Thus the primitive abilities needed are for various kinds of pattern recognition and then for modifying the patterns recognized.

For example, a system to simplify algebraic equations might contain a rule like the following:

"A + B = C + B" → replace with: "A = C"

Here the quote marks indicate what's in the workspace itself, with the capital letters (A, B, and C) understood as placeholders for arbitrary strings of actual workspace tokens; hence the equation

$$2x + 4y(z - 14) = w - 3 + 4y(z - 14)$$

would reduce to

$$2x = w - 3$$

in a single application of the rule. This example presupposes some fairly classy recognition and matching capabilities—for instance, the ability to match the complex expression "4y(z − 14)" with "B" on both sides of the equation. But such facilities are common in production systems; and with them, many useful productions (including other algebraic simplifications) are easy to design and easy to understand.

In one way, Newell machines are throwbacks to Turing machines. For unlike LISP or von Neumann machines, Turing and Newell machines use no explicit "IF . . ."; rather, they have conditionalization built right into the architecture: *every* step is conditional. Thus which successor a Turing unit nominates always depends on which token it found on the tape; likewise, a production will self-activate only on condition that it finds its special pattern in the workspace. The point can also be put in terms of sequencing: von Neumann and McCarthy machines don't *need* to make a decision at each step because they have a "natural" order of execution "built in." A von Neumann machine automatically follows its schedule, and LISP automatically evaluates the arguments of each function before evaluating the function itself. Hence conditionals are needed only occasionally, to redirect the flow at critical junctures. Turing and Newell machines, by contrast, have no predisposed flow, so an explicit decision must be made for each step.

But Newell machines are far from just Turing machines warmed over: the basic spirit is entirely different. Turing's intent was to make the machine itself as simple as possible (just a few very

Box 6
Repetition in Production Systems

The production architecture, like the others we've discussed, is universal: for any symbolic algorithm, some production system can execute it. Hence production systems must be able to repeat arbitrarily, until some condition is met. In fact, repetition comes naturally to production systems; the hard part is stopping or avoiding it. Consider, for example, a system with just one production:

any integer, N → increment: N

(Incrementing a number means increasing it by one.) If this system ever finds an integer to get started on, it will just sit and increment till the cows come home. Yet there is nothing (no branch instruction, loop, recursive definition, etc.) that tells the system to repeat; it just does. On the other hand, if you want it ever to stop (say, when it gets to 100), then you have to tell it, by adding another production, with higher priority:

any integer greater than 99 → stop

(Alternatively, the test could be incorporated into the condition clause of the original production.)

As another illustration, imagine a workspace that can hold LISP-like trees, and a kit of primitives allowing tree manipulations. Then the following simple production system does the same job as the COUNTATOMS[__] function from box 5:

any atomic tree, A, and integer, N	→ delete A and increment N
any tree, T	→ replace with: LEFT[T] & RIGHT[T]
anything at all	→ stop

The workspace initially contains the tree to be counted and

the numeral 0. The middle production repeatedly cuts trees in half, until it reduces them to their terminal atoms; meanwhile, the first production constantly sweeps up and discards those atoms, incrementing the counter each time. Notice again that the "recursion" or "looping" just takes care of itself—the "invisible hand." The last production never gets a chance until the first two have completely chopped up the original tree, counted the pieces, and swept them all away; then it stops the system.

primitive units, operating only on atomic tokens) and then let its programs be numbingly large and tedious. Production systems, by constrast, tend to have very powerful and sophisticated "primitives"—such as automatic finding and matching of complex patterns (with variable constituents)—which go naturally with pattern-based access and self-activating players. Accordingly, Newell programs are comparatively brief and elegant: each line does a lot.

Production systems promote a degree of "modularity" unrivaled by other architectures. A *module* is an independent subsystem that performs a well-defined task and interacts with the rest of the system only in certain narrowly defined ways. Thus the tape deck, preamp, etc. of a "component" stereo are modules. In LISP, user-defined functions serve as separate program modules; but they are not as independent as individual productions.[17] For any LISP function to evaluate, it must be called explicitly; and the caller must know not only when to call that function, but also its name, what arguments to provide, and what sort of value to expect back. Or rather, the programmer must keep track of all this for each situation where any particular function could (or should) be invoked.

A production, on the other hand, will activate itself when and where conditions are right, regardless of what any other production knows or decides. The point can be illustrated with a fanciful example. Suppose we're designing a robot taxi driver and everything is in place, except that we forgot about sirens. So now we

have to add a feature: whenever the robot hears a siren, it should pull over until the danger is past. Depending on the architecture, that would involve a new subroutine, function, or production that checks memory for a SIREN report (from the "ears") and outputs a PULL OVER signal when it finds one.

How can this new unit be integrated into the rest of the system? Presumably the driver should respond to a siren (if any) before entering an intersection, before pulling out from a parking place, before changing lanes, and so on, as well as while driving along the road. Hence in a von Neumann or McCarthy machine, the routines or functions that perform all these activities would *each* have to be modified, so that they call on the new SIREN unit at appropriate moments. In a Newell machine, on the other hand, the SIREN production merely has to be added to the lineup, with a suitably high priority; no existing productions need be touched. The new one will just sit there quietly, doing nothing and bothering no one, until a siren comes along; then it will issue the PULL OVER signal all by itself.

Counterbalancing the obvious advantages of production modularity, however, are two limitations. In the first place, not every task is conveniently divisible into components that can be handled autonomously; that is, sometimes the subtasks must be explicitly coordinated, with particular information or instructions being directed to particular units at particular times. Such organization, while not impossible in production systems, is distinctly awkward compared with "centralized" subroutine or function calls. Second, production modularity is single-level; that is, in a production system, you don't build high-level modules out of lower-level modules, built out of still lower ones, and so on. All the productions are on a par, except for priority. LISP definitions, by contrast, are inherently hierarchical, and von Neumann programs often are.

Our primary goal in this chapter has been a broader view of what computers can be. Too often people unwittingly assume that all computers are essentially like BASIC, FORTRAN, or even Turing machines. Once it is appreciated that equally powerful (i.e., universal) architectures can be deeply and dramatically different, then suddenly the sky's the limit. Artificial Intelligence in no way claims that the mind is a Turing machine or a BASIC

program—or any other kind of machine we've considered. It is true that, for reasons of convenience and flexibility, most AI programs happen to be written in LISP. But the virtual machines thus concocted are not LISP machines; in principle, they could be anything. It is also true that some workers suspect the mind itself may be organized rather like a production system, though one much more elaborate and sophisticated than the elementary structure reviewed here. The important point, however, is this: the mind could have a computational architecture all its own. In other words, from the perspective of AI, *mental architecture* itself becomes a new theoretical "variable," to be investigated and spelled out by actual cognitive science research.