Computer Science: From Abstraction to Invention

Jennifer Burg and Stan Thomas

Of the scientific disciplines taught at today's universities, computer science is the youngest, a child born of human introspection and ingenuity. Unlike the natural sciences, which seek to discover and understand the natural world around us, computer science is a unique combination of abstraction and invention. On the one hand, computers are fashioned in their essence from abstract logic, emerging from efforts to understand, formalize, and simulate human thought. On the other hand, computers are ingeniously engineered machines originally invented to solve very real problems -- to do tedious, error-prone mathematical calculations; to process volumes of census data; to compute complex ballistic tables for missile trajectories; or to crack enemy codes during World War II. Now, less than 60 years after the invention of ENIAC, EDSAC, and UNIVAC, computers have changed the way we communicate, learn, solve problems, do business, and make war. But how has computer science managed to bring together the power of abstraction and the creativity of invention? What are the fundamental ideas and advances that have led computer science to have such an impact on the modern world?

Abstraction

Glory be to God for dappled things --
For skies of couple-colour, as a brind'ed cow....
Gerard Manley Hopkins

The world is made up of a myriad of wonderful and varied *things* -- different colors, different shapes, different uses, each individual, each unique. This great variety of things "counter, original, spare, and strange" is a diversity to be celebrated, as poet Gerard Manley Hopkins rejoices in his poem "Pied Beauty." But there is another gift to be celebrated with regard to the rich variety of creation, and that is the human power of abstraction.

What if we had to deal with all these things one by one, with no way to group them into categories, no way to think of them generally rather than individually? Our minds simply could not handle the complexity. Abstraction is the process of stripping away individual characteristics and thinking about only those qualities that matter for our purposes at the moment. If I say to you, "Send me a postcard when you get to Paris!" neither one of us needs to picture a particular postcard. We both understand quite well a class of objects that are, collectively, known as postcards -- generally rectangular, with a picture on one side and space for a note and address on the other. The details are unimportant at the moment.

Abstraction is arguably the most fundamental intellectual activity in the field of computer science. Problem solving is made manageable by our ability to approach it at different *levels of abstraction*, which vary according to how closely we focus on the details. To program a computer, we give it a *procedure*, a sequence of instructions for solving a problem. As we develop this procedure, it is usually best to begin by backing away to get the big picture. How do we find the median of a list of numbers, for example? One way might be to (1) sort the list of numbers, (2) find the length of the list, and then (3) find the number in the middle of the sorted list. It's easy to think about how solve to this problem if we begin at this high level of abstraction. If instead we get bogged down from the beginning in working out the details of sorting numbers, we lose perspective on our overall task. In the initial high-level procedure for finding the median, the sorting method can temporarily remain what computer scientists call a "black box." We assume we can put a list of numbers in one end, and it comes out sorted. Later, when we have designed our overall procedure for computing the median, we can specify the inner workings of the sorting procedure. The usefulness of abstraction in managing the complexity of problem solving is even more apparent when the problem requires many more steps than our example. Furthermore, one black box may in turn have other black boxes in it; that is, one subprocedure may contain more, lower-level subprocedures whose operations remain temporarily unspecified. But by masking details until we are ready to consider them, we save ourselves from being overwhelmed by a complex problem.

In computer programming, problem solving procedures are described abstractly in levels that descend from our human way of thinking, and arrive finally at a language that directly reflects the changing states of a machine in operation. Eventually, the procedure must come down to something concrete and physical, the flipping of a switch or the toggling of a transistor. The closer the symbolism is to the physical manifestation of the computer's changing states, the lower the level of abstraction. The closer the symbolism is to human language and the concepts and processes represented, the higher the level of abstraction. It is easiest for us to think in human language. In the end, however, the machine has only the most basic of building blocks, two-state devices that know only *yes* or *no, on* or *off*, 0 or 1, *true* or *false*.

Boolean Building Blocks

When mathematician George Boole formulated his axioms for an algebra of logic in 1854, he could not possibly have envisioned their importance to the future development of computing machines. The mechanical computing devices envisioned by Blaise Pascal, Charles Babbage and other pre-20th century thinkers operated in the familiar base ten number system. Boole’s logic laid the foundation for computing devices working in a base two, binary-valued number system.

The fundamental idea of Boolean logic is simple. A variable can have only two possible values, *true* or *false*. Analogous to arithmetic operators (addition, multiplication, etc.) that operate on numbers, Boole defined operators ---- *and, or, not, implies,* etc. -- that apply to *true/false* values. The definitions of these operators tell us the result we get when we apply some operator to two particular operands. The operators are named and defined in ways consistent with innate human logic and classical logical principles. For example, say that it is true that "Spring is here." Say that it is also true that "The tulips are blooming." Then, is the compound statement "Spring is here *and* the tulips are blooming" also true? By our everyday reasoning, it is obviously true, and by Boole's definition of the operator *and* (denoted by $\wedge$), it is true as well.

What Boole did not foresee was how naturally his formal logic could lead to a computing machine. *True* and *false* are easily represented by any device that can be in exactly one of two states at any given moment -- a mechanical relay or a transistor, for example. *True* is simply "on" (i.e., the value 1) and *false* is "off" (i.e., the value 0). Boolean logic was made-to-order for computational devices. The implementers of the first computers in the 1940's discovered that two-state devices were simple, cheap, and easy to work with. Operands became inputs to the two-state devices, which were constructed to yield outputs consistent with the Boolean operations. What could be done, then, with millions of these two-state devices, connected in complex logical networks of 0's and 1's -- in digital circuits, that is?

Just about anything, it turned out. Early computers were built primarily to do numerical computations, but that was only the beginning. Later, binary encoding was adapted for storing text and other non-numeric information as well. Today, every school child knows how to manipulate sounds and images in the computer. They do not have to understand that this data, too, is encoded as patterns of 0’s and 1’s obtained through digitization. Knowingly or unknowingly, we utilize Boolean logic with each and every interaction with a modern computer. Every symbol, number, instruction, or other datum stored in a computer is, at some level of abstraction, a collection of binary digits or *bits*. Every operation carried out in a computer’s circuits can, at some level of abstraction, be viewed as an operation in Boolean logic.

Logic for Thinking Machines

Boolean logic is an abstraction that can be realized concretely in the form of digital circuits, the logical wiring from which computers are constructed. At the same time, Boolean logic is a rigorous mathematical system that attempts to mimic human thinking, specifying rules by which our reasoning operates. Logicians realized that if these rules could be made definite and sure, not varying according to the subjects on which they are applied, perhaps the reasoning process could be automated in the form of a thinking machine. In this sense, Boolean logic is a step in the direction of artificial intelligence.

Let us consider how formal logic makes artificial intelligence possible. Formal systems of logic like Boole's begin with *propositions* -- statements about the world that we know to be either true or false -- and rules for combining these to determine the truth or falsity of a group of propositions as a whole. Remarkably, the rules are so consistent and inflexible that it doesn't matter at all what the propositions actually mean in order to reason about them. How can this be so? How can we possibly be logical with no regard to the subjects of our discourse?

The secret, again, is in the power of abstraction. Consider the logical operators defined in Tables 1, 2, and 3. Each is defined in terms of two abstract propositions, P and Q. P might represent any proposition. Perhaps it is, "Spring is here," as in our previous example. Q might again be "The tulips are blooming." $P \rightarrow Q$ would then denote "If spring is here then the tulips are blooming," a true compound proposition if we know that both P and Q, individually, are true to begin with. But if we say that P denotes "Computers have eyebrows" and Q denotes "Pigs can fly" and we specify that P and Q are true, the compound proposition $P \rightarrow Q$ is just as true (i.e., "If computers have eyebrows then pigs can fly"). The point is that the truth value of $P \rightarrow Q$ can be computed without knowing what P and Q *mean*. We need only say whether P and Q, individually, are true or false.

P	**Q**	**$P \wedge Q$**
false	false	false
false	true	false
true	false	false
true	true	true

Table 1. Logical *and*.

P	**Q**	**$P \vee Q$**
false	false	false
false	true	true
true	false	true
true	true	true

Table 2. Logical *or*.

P	**Q**	**$P \rightarrow Q$**
false	false	true
false	true	true
true	false	false
true	true	true

Table 3. Logical *implies*.

The implications to machine intelligence are profound. Computers don't have much common sense. They aren't very good at looking at the world around them and determining what is true and false at a basic level -- whether pigs can fly or tulips are blooming. But they are good at representing the values *true* and *false* once the basic facts are given to them, and they can manipulate *true/false* values quite adeptly according to rules of logic. Logical computation can be automatic, mechanical, and mindless in the sense that a computing machine need have no understanding of the subject of its reasoning. But, mindless as it may be, the problem solving potential seemed enormous to the mathematicians of the 19th and early 20th centuries. What could a machine do for us? Prove theorems? Establish whole systems of mathematics like arithmetic or geometry? If we provided the basic facts, could computers stretch our understanding of the world by taking the facts to their logical limits?

These questions were posed by philosophers, mathematicians, and logicians such as Charles Babbage, Gottfried Leibnitz, Alan Turing, Kurt Godel, Bertrand Russell, and Alfred North Whitehead long before the first electronic computers were built. Answers emerged, as we shall see, as mathematical theory and computer engineering began to converge in the 1930's.

Mechanical Computation and Universal Computing Devices

The process of abstraction appears again in our efforts to understand the essence of computation itself. It is a powerful thinking tool, this ability to strip away the incidental and particular and see a concept in its naked purity. Abstraction was Alan Turing's special genius, the talent that ranks him among the founding fathers of computer science. While working on urgent message-decryption problems in World War II and helping to develop technology that would lead directly to the first electronic computers, Turing even more importantly was able to look beyond the particulars and see the great new idea.

To lay bare the notion of mechanical computation and see what Alan Turing saw, we must forget about keyboards and computer screens and integrated circuits, and broaden our idea of a machine by thinking of it entirely in the abstract. A machine is simply a mindless entity that changes from one state to another. It doesn't have to be mechanical in the physical sense, like Charles Babbage's early computational device, a contraption of gears and levers and camshafts. A machine doesn't require electromechanical relays or transistors. The only requirement is that we be able to represent and prescribe the machine's change of states. The great realization is that we need no more than a pencil and paper to "build" a machine, since it is a matter of describing the symbols it can recognize, the states it can be in, and precisely what causes it to change from one state to another.

Turing arrived at just such a definition, an abstraction we call the *Turing machine*. Turing argued that mechanical computation is completely captured by a device of the following description: Imagine that we have a "machine" that uses a fixed alphabet of symbols. 0's and 1's suffice, though other alphabets would work as well. The machine has a tape of unbounded length on which strings of these symbols are written, and a "reading" device positioned over exactly one symbol at any given moment. We consider the machine's *state* at discrete moments in time. Supplied for the machine are *rules* which tell it what to do each time it reads a symbol. These rules depend on the symbol just read and the state the machine is in at that reading. A rule can tell the machine to (1) move to the right or left, leaving the symbol unchanged or (2) move to the right or left, first rewriting the symbol to some other symbol in the alphabet. Turing maintained that, while this machine may be slow and inefficient, it can perform any computation that could be performed by any other machine, no matter how hi-tech or sophisticated.

Turing captured, in the abstract, the essence of mechanical computation. His calculating machine did not have to be constructed from wires or wheels or anything we normally associate with a machine. In fact, each individual machine of this type can be described symbolically and its activity performed entirely with pencil on paper. The device is mechanical not in a physical sense, but in the sense that all its actions are automatic and lacking spontaneity, completely prescribed by the symbols, rules, and states defined for it. Given a certain input of symbols, the Turing machine can go through only one sequence of actions. To the machine, there is no meaning associated with its actions. It is only the human creator of the machine who ascribes meaning to the input and output symbols.

Great minds had been circling around notions of mechanical computation for centuries, and in the first decades of the 20th century, mathematics, logic, and technology came together to bring the ideas to fruition. A formal model of abstract computation was finally ripe for the plucking, and Turing was not the only one to reach for it. Turing had ventured a model for a "universal" computing device, and in a doubletake, scientists of the 1930's and 40's realized that other ostensibly quite different computational models were in fact equivalent to Turing's, including Kleene's recursive functions, Church's lambda calculus, and Post's rewrite rules. This was as great a discovery as the abstract machine definition itself. As mathematician Kurt Godel later noted with amazement, the concept of computability transcended the formalism in which it was expressed.

Completeness and Computability

From the beginning, investigations into abstract computation were motivated by efforts to understand how humans think and logically solve problems. The goals were soaringly ambitious, taking abstraction to its very limits. Could we describe, in precise mathematical language, the logical relationships that we all accept in some realm of thinking -- say arithmetic, for example? Could we write these relationships in the form of rules that describe how one statement of truth can be logically transformed into another, rules that are so precise that even a mindless machine could be "programmed" to follow them? Could we then write down in our formal language all the things we know to be true in the world we are describing (e.g., *axioms*), give the whole lot to the machine, and allow the machine to deduce every possible conclusion implied by the logic and the axioms?

What an ambitious and tantalizing thought, one which tempted philosophers, logicians, and mathematicians for centuries. This was mathematician and philosopher Gottfried Wilhelm von Leibnitz's great project in the 1600's: to devise a formal and universal language of logic "in which all truths of reason would be reduced to a kind of calculus." Like Turing, Leibnitz worked in both the abstract and the concrete, building one of the first calculating machines (the "calculus ratiocinator") at the same time that he explored the limits of formal logic.

Over 150 years later, Leibnitz's goals had still not been realized, but neither had they been abandoned. In the 1840's, Charles Babbage was working tirelessly on his Analytical Engine, a massive computing machine intended to demonstrate the possibilities of mechanical computation. While Babbage worked on the concrete realization, George Boole continued in the more abstract vein, investigating "the fundamental laws of those operations of the mind by which reasoning is performed" and formulating, in the course of his inquiries, "some probable intimations concerning the nature and constitution of the human mind."

Even as late as 1910, the pursuit of the ultimate theorem-proving system continued in the research of Alfred North Whitehead and Bertrand Russell, who aspired to construct a system that could derive all of mathematics through formal logical operations on a collection of axioms. Mechanization -- the ability to prove theorems automatically -- remained a central issue of the research. Axioms could be treated by a machine purely as symbols with no meaning. Based on their form and on precise rules of logic, a machine could rewrite these axioms from one form to another, and thereby prove theorems. The machine need have no understanding whatsoever of what it is doing.

To understand how logicians went about their task of formalizing the logic of arithmetic, let us consider some of the axioms that might form the basis of a theorem proving system. We will draw examples from the domain of arithmetic over the natural numbers, a relatively simple system that logicians had hope of fully representing.

Table 4 shows the kinds of statements that we might need to "tell" the computer, in the language of predicate logic (an extension of propositional logic). Axioms and inference rules like these give us a way of making deductions and transforming one logical statement into another. They set us on the path to describing a complete system wherein all truths about arithmetic might be automatically deduced. But how many axioms like these do we need as a base? What rules and axioms are necessary and sufficient so that all other theorems can be automatically derived from them? Are there limits to the power of formal logic? Is it possible to formally describe a self-contained logical system in which everything that is true can be proved, everything that is false can be disproved, and nothing can be proved both true and false -- that is, a system that is both *complete* and *consistent*? In 1900, David Hilbert, one of the foremost mathematicians of his day, was optimistic, conjecturing in a preface to his famous twenty-three unsolved problems that if a proposition can be expressed within some formal mathematical system, then either its proof or its refutation must exist.

Type of Statement	**Axiom**	**Meaning**
axiom	$\forall x(P \rightarrow (Q \rightarrow P))$	For all natural numbers x, if P is true of x, then "Q implies P" is true of x.
axiom partially defining equality	$\forall x, x = x$	For all natural numbers x, x is equal to itself.
axiom partially defining the successor function	$\forall x, y \ (s(x) = s(y) \rightarrow x = y)$	For all natural numbers x and y, if the successor of x is equal to the successor of y, the x is equal to y.
axiom partially defining addition	$\forall x, y \ (x + s(y) = s(x+y))$	For all natural numbers x and y, x plus the successor of y is equal to the successor of x + y.
inference rule	(P and $(P \rightarrow Q)) \Rightarrow Q$	If P is true, and $P \rightarrow Q$ then Q must also be true.
Note: $\forall x$ means "for all x"; $s(x)$ is the *successor function*, referring to "the successor of x" (i.e., the number after x)		

Table 4. Axioms in the Arithmetic of Natural Numbers

For many years logicians followed the trail of Hilbert's conjecture. But in 1931, Kurt Godel brought down to earth logicians' ambitious plans to generate "truths" with the help of computing machines. In his Incompleteness Theorem, Godel showed that for any reasonably powerful, consistent system of logic or arithmetic that we might want to describe, there will always exist statements that are true, but that cannot be proved solely on the basis of the axioms and rules of the system.

Godel's argument, in sketch, is as follows. Say that in our system, we assert the following proposition:

You cannot prove that this proposition is true.

To see that a system containing such a proposition is either inconsistent or incomplete, consider the following. First assume that the proposition is true. Then, on the basis of what the proposition says, you can't prove that it is true, so the system must be incomplete. Now say that the proposition is false. Then on the basis of what the proposition says, you *can* prove that it is true, which would be inconsistent.[1]

Godel's theorem effectively defeated hopes for a complete theorem-proving system, even in the relatively simple domain of arithmetic. It did not imply, however, that automatic theorem proving was impossible -- just that it is possible to give true theorems to your system that it won't be able to prove.

Similarly, limits were found to the computing ability of Turing machines and their equivalents. It was concluded that all models for abstract computation had equivalent -- but not unlimited -- computing power. Some problems simply cannot be solved by mechanical means. The classic example is the *halting problem*, the problem of asking one Turing machine to tell us if another Turing machine will ever terminate its execution on some given input. In fact, it is impossible to write a program that can tell automatically, for any other program you might hand it, if that given program will ever stop executing on a given input. In general, the halting problem "does not compute."

The news was not entirely bad. It turns out that while, theoretically, there are more noncomputable problems than computable ones for computers, we are hard-pressed to come up with examples. Certainly, most of the computation humans are interested in doing lies in the computable realm. Generally, the problems that cannot be solved by formal or mechanical means are the ones that take us to metalevels of logic and computation, questions like "Can this proposition be proven true?" or "Will this program ever terminate if I run it on this input?" The inability of formal logic, separated from intuition, to answer metaquestions is intriguing in itself, one of those recurring roadblocks to mechanical computation that have a way of making us feel better about our human way of solving problems.

Mind or Brain?

From the beginning, the development of computers has been tied up with the hope of creating an artificial intelligence, a machine whose abilities equal or exceed those of humans. Perhaps the biggest surprise has been that those tasks which are difficult for humans—complex calculations and rote memorization—are easy for computers, whereas computers have difficulty understanding natural language, comprehending the physical world, planning, and adapting—tasks instinctive to and necessary for human survival.

Even though the nature of AI research has constantly evolved over the past fifty years, in truth there is still little consensus on issues as fundamental as whether we should seek to simulate or to duplicate intelligence. To simulate intelligence is to make a machine

[1] This informal argument captures the spirit of Godel's theorem, though a formal proof is based upon the Godel numbering system, whereby every proposition expressible in the system is given a unique integer number.

do the same thing that intelligent beings do -- that is, think -- using symbols and logical operations that mimic the workings of the *mind*. To duplicate intelligence is to make a machine function like the *brain*, with its neurons and synapses and transmission of electrical impulses.

Traditional AI is based on the symbol hypothesis of John Newell and Herbert Simon. This hypothesis states that a symbol system is all that is needed to give a machine the power of intelligent action. From this perspective, the formalisms of Boolean and propositional logic form the basis of an artificial intelligence. But we have already acknowledged that the key to the power of machine logic is to separate logical computation from meaning. While logical machine problem solving may simulate thinking, is it really *thought*?

John Searle tried to make clear the difference in his Chinese room parable. Imagine that someone is inside a closed room, equipped with a myriad of complex rules that unfailingly dictate how to turn questions that are written in Chinese into sensible answers in the same language. This person knows not one word of Chinese himself. Questions written in Chinese are passed through a hole in the door, and our translater is given whatever time is needed to turn these into answers. Does this person understand anything of the notes he is translating? Clearly he does not, any more than the computer understands the meaning we may ascribe to the abstract symbols we ask it to manipulate.

More recent AI research has turned to connectionism for a different approach to creating artificial intelligence. This approach is based on creating networks of neuron-like entities, connected in a manner akin to the physical operation of the brain itself. Experiments with these neural network computers show that they can be invested with considerable problem solving ability -- even, for example, controlling the movements of a robot as it attempts to move across a room, manuevering around the objects in its way. But once again, can we fairly say that such a robot is intelligent? How precisely do we have to model the functioning of the human brain before we can justifiably say that we have given a computer the power to think? Is there some level of exact duplication of brain functioning that would finally yield the machine's self-awareness, the ultimate test in intelligence? Whether or not machines can ever truly understand anything and be aware that they are thinking is an interesting philosophical question as deep and challenging as the human mind-body problem itself.

From Abstract to Concrete, Symbol to Meaning, Science to Technology

Ultimately, we move from the abstract to the concrete. We want our computers to do something or solve something. Show me what my yard will look like with its new landscaping. Tell me how to get to 42nd street. Calculate *pi* to the 20th digit. Find the spelling errors in my memo. Land a spacecraft on Mars.

The movement from symbol to meaning, abstract to concrete, science to technology, has brought us to the world we live in today. Once it was realized that computers are "universal" problem solvers, we no longer saw them as tedious number crunchers and realized their potential as all-purpose machines. The devices that were originally invented to calculate missile trajectory tables and handle census data could just as well process words, draw pictures, and give us virtual worlds to explore. The abstract notion of a universal computing device came to have unexpected practical ramifications -- computers for all purposes, and accessible to everyone. Computers became "personal," "user-friendly," and then "ubiquitous."

What were the great, real-world, technological advances that grew up with computer science and led us to the world we live in today? From the layperson's point of view, the advances that have had the greatest impact are those that have made computers smaller, more useful, and more accessible, bringing them into our homes and everyday lives. These advances began in hardware with the invention of the transistor in 1954, a two-state device to replace vacuum tubes and electromechanical relays. The integrated circuit, a compact collection of transistors on a silicon chip, followed in 1959. Picture the difference that 50-some years have made. In 1946, computer scientists were awed by ENIAC, a tangle of more than 17,000 interwired vacuum tubes, weighing over 30 tons and standing 100 feet long, 10 feet high, and 3 feet deep. Today, we take for granted microprocessor chips the size of one's thumbnail, on which are etched millions of transistors of far greater computational power. ENIAC had to be hard-wired for each particular task, a tedious and error-prone process. Von Neumann's stored-program concept -- the idea that the instructions for program execution should be given to the computer symbolically rather than through hardware -- was quickly recognized as essential to the general usefulness of computers, and programming languages were born.

Still, the smaller programmable computers of the 1960's were not something you would have in your home. The language of computers was arcane and the applications were generally scientific or business-oriented. People might want to use computers -- to communicate, play games, write letters, or prepare a budget -- but few had any desire to program them. Better yet, let computers look and act like things we're comfortable with -- desktops, folders, papers, and trash cans, for example. With this breakthrough idea, graphical user interfaces replaced text-based systems with their cryptic typed-in commands. Computers could be used by children, and even more surprisingly, their parents.

It was quite natural that when computers became "personal" and individuals were given full-powered computers right on their own desks, the advantages of connecting them through networks became obvious. No longer did many users connect to one big mainframe master computer. Instead, networks of computers were linked together, sharing memory, resources, and information. Computers became, perhaps most importantly for the layperson, a writing tool and a communication device.

The most striking advance in computer technology of the last four decades, the one that has accelerated us into the Information Age, is the development of the Internet. The reach of our computers now stretches from our desktops across the world, and in a matter of seconds. Understanding how this could happen requires our acceptance of one of the biggest abstractions of all -- The World Wide Web. What does it mean when we say that we are "on the Web," and how does that differ from being "on the Internet"?

By now, the story of the Internet's creation is generally known. Originally conceived by the U.S. government's Advanced Research Projects Agency as a network of computers that could communicate securely even in times of crisis, the ARPANET quickly grew under the National Science Foundation to accommodate academic researchers at universities around the world. By 1984, the term Internet was in general use, referring to a network used largely for the exchange of electronic documents and messages. But Internet use was reserved mostly for computer specialists who knew the arcane language of ftp, gopher, archie, and email. Even these specialists tired of the text-based, type-it-in manner in which files had to be located and moved from one computer to another. In the early 1990's, researchers at the CERN laboratory in Sweden worked out an even more convenient, graphical and mouse-based system for sharing information. Rather than type in cryptic text-based commands to pull a file from a remote computer to another, a computer user could simply click on a word or picture and see the document directly on his or her computer screen. Through programs called Web browsers, information could now be accessed directly in a cleanly formatted, readable form complete with color, pictures and even sound. Thus the ultimate abstraction was born: The World Wide Web, which is simply the Internet served up palatably in a multimedia format -- "pages" of information on computers all over the world, instantly accessible at your own desk, easily assimilated, and in an unending supply.

Repercussions

These technological advances have placed us in a world where we communicate more, but differently, with screens to separate us. We are closer to those who used to be far away, but too often distant from those close to us, distracted by the world seen through our computer screens. We seek uncensored expression and access to information across global channels, yet fear invasions of our privacy and assaults on our sensibilities. We have more information at our fingertips than we know what to do with. Our worlds are filled with labor-saving devices and problem solving tools, and we seem to have less and less time to do the things we think expected of us. Computers have made life easier, and they have made life harder.

We might listen to the philosophical repercussions of computer science's great ideas as well. Sometimes, the "negative" results have the greatest resonance. Two brilliant minds independently formulate complete systems of abstract computation and learn, perhaps to their dismay, that their landmark results have been discovered elsewhere. But the realization that machine computation can be carried out in vastly different forms to achieve precisely the same results is a remarkable discovery itself -- reminding us that important ideas may masquerade in different forms, yet be essentially unchanged. Kurt Godel shows that sometimes we simply cannot prove everything we might like to prove within a closed, self-contained system. To see that some things are true, it may be necessary to step outside the system and use other strategies of logic or intuition. There seems to be certain wisdom in that. Turing shows that some problems are not solvable by abstract computation, and we see the ineffable and unsolvable of human experience mirrored in mathematical theory. Perhaps most importantly, we learn from computer science that to see the essence of something, we must strip away the particulars and unimportant differences among things. This is the heart of problem solving.

The quest for a thinking machine has taken us from the limits of abstraction to marvels of invention. It has resulted in a science that continues to change our world in ways we can hardly keep up with. Computer science began with attempts to make machines that could think *for* us and *like* us, and it has ended up giving us far more to think about.

Further Reading

Boole, George. *An Investigation of the Laws of Thought, on which are founded the mathematical theories of Logic and Probabilities*. London: Macmillan, 1854.

Casti, John L. *The Cambridge Quintet: A Work of Scientific Speculation*. Reading, MA: Addison-Wesley, 1998.

Davis, Martin, ed. *The Undecidable*. Hewlett, N.Y.: Raven Press, 1965.

Dewdney, A. K. *The New Turing Omnibus: 66 Excursions in Computer Science*. New York: W. H. Freeman and Co., 1993.

Dyson, George B. *Darwin among the Machines: The Evolution of Global Intelligence*. Reading, MA: Perseus Books, 1997.

Godel, Kurt. "Uber formal unentscheidbare Satze der Principia Mathematica un verwandter Systeme I." *Monatshefte fur Mathematik und Physik* 38 (1931). Translated in Martin Davis, ed., *The Undecidable*.

Newell, Allen, and Simon, Herbert. *Human Problem Solving*. Englewood Cliffs: Prentice-Hall, 1972.

Searle, John. *Minds, Brains, and Science*. Cambridge: Harvard University Press, 1984.

Turing, Alan. Computing Machinery and Intelligence. *Mind*, 59 (October 1950), 433-460.

Von Neumann, J. *The Computer and the Brain*. New Haven, Connecticut: Yale University Press, 1958.

Whitehead, Alfred North, and Russell, Bertrand. *Principia Mathematica*. Cambridge: Cambridge University Press, 1910.