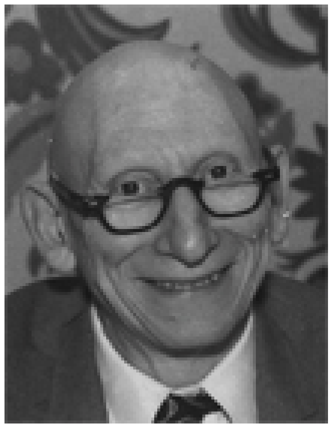
Alan J. Perlis

Born April 1, 1922, Pittsburgh, Pa.; died February 7, 1990, New Haven, Conn.; computer pioneer; leader in establishing computer science as a legitimate discipline; longtime developer of programming languages and programming techniques; author of classic one-liners.



Education: BS, chemistry, Carnegie Institute of Technology (now Carnegie Mellon University), 1943; MS, mathematics, MIT, 1949; PhD, mathematics, MIT, 1950.

Professional Experience: research mathematician, Aberdeen Proving Grounds, 1951; Project Whirlwind, MIT, 1952; assistant professor, mathematics, Purdue University, 1952-1956; Carnegie Mellon University: associate professor of mathematics, 1956-1960, director of the Computation Center, 1956-1971, professor and chairman, Mathematics Department, 1960-1965, co-director, Graduate Program in Systems and Communications Science, 1962-1965, head, Graduate Department of Computer Science, 1965-1971; Eugene Higgins Chair in Computer Science, Yale University, 1971-1990.

Honors and Awards: ACM Turing Award, 1966;[1]" AFIPS Education Award, 1984; IEEE Pioneer Award, 1985; member, National Research Council Assembly of Engineering, 1978-1990; member, National Academy of Engineering, 1976-1990; member, American Academy of Arts and Sciences; honorary doctorates: Davis and Elkins College, University of Waterloo, Purdue University, and Sacred Heart University; editor-in-chief, *Communications of the ACM,* 1958-1962, president, ACM, 1962-1964.

In 1951 Perlis was employed as a research mathematician at Aberdeen Proving Grounds, and in 1952 at Project Whirlwind. Later that year Perlis became an assistant professor of mathematics at Purdue and was responsible for forming the university's digital computer laboratory. In 1955 he organized an effort to build the IT compiler for the Datatron 205 at Purdue and this work continued on the IBM-650 when he took the position of associate professor of mathematics and director of the computation center at Carnegie in 1956. Along with Joseph Smith, Harold van Zoernen, and Arthur Evans, he designed and built a succession of algebraic language compilers and assemblers for the IBM-650. A course on programming, distinct from numerical analysis, was instituted in 1958 and made accessible to undergraduates at all levels. Also in 1958, Professor Perlis became one of 8 people to define the programming language Algol 58, and in 1960 he was one of 13 international scientists involved in the definition of Algol 60.

Alan Perlis was named professor and chairman of the Mathematics Department at Carnegie in 1960 and he also remained as director of its computation center. Then, in 1962, he became co-director of a graduate program in systems and communications science, whose goal was the development of a graduate program in what has now become computer science. In 1965, the graduate department of computer science was founded at Carnegie, and he became its first head.

[1] The first ACM Turing Award.

Throughout the 1960s he was involved in the definition and extensions of Algol, that is, Formula Algol, a version for manipulating formal mathematics expressions, and LCC, a version adapted to interactive incremental programming. He joined the new Computer Science Department at Yale University as Eugene Higgins Professor of Computer Science in 1971, a position he held until his death.

His research was primarily in the area of programming language design and the development of programming techniques. He was the author or coauthor of many published articles, and two books, *Introduction to Computer Science* and *A View of Programming Languages*. He was the founding editor of the *Communications of the ACM* and served as president of the association from 1962 to 1964.

In 1981 Perlis was asked to give a talk on "Computing in the Fifties" at the ACM National Conference in Nashville, Tenn. The transcript of his talk follows:[1]

One has to understand that computing after World War II was largely concentrated in a few locales under the support of agencies of the federal government. Whirlwind at MIT and the Mark I, II, III computers at Harvard were both supported, by and large, by the Office of Naval Research. The Ordnance Department of the US Army supported a large activity at the University of Illinois. A few other smaller efforts were under way in other places. Within universities as such, however, there was no computing, other than some small punched card calculations being done in various statistical departments around the country. Almost all computing was essentially funded by the federal government and aimed at developing computers that would be directly used within the defense effort.

In 1952 I was fortunate to be invited to Purdue University, where the director of the Statistical Laboratory, an unsung hero of our profession named Carl Cossack, decided that an engineering school should have a computer that would be used by its students and faculty as part of its educational and research program. I accepted a position at Purdue and went out there to start a computing center. There were no computer science departments in those days; everybody who worked around computers was either an electrical engineer or a numerical analyst. The courses on the computer usually were of two kinds: courses in circuitry taught within the engineering department, or courses in numerical analysis taught under the aegis of the mathematics department.

At Purdue we searched for one year for a computer-the university having given us $5,000 and a Purdue automobile to travel through the eastern and middle-western part of the country. We saw some very strange computers. One I will never forget was being developed by Facsimile in New York City in a loft, in the Lower East Side, on the third floor that one reached by going up on a rickety freight elevator. The machine was called the Circle Computer. It had one important attribute: there was a socket on the top of the computer into which one could plug an electric coffee pot. The Circle Computer showed me that machines really do have a personality and a spectrum of emotions over which they run; the Circle Computer would only run when one of its designers was in the same room. As soon as he happened to leave the room the computer would stop. National Cash Register, for some strange reason, bought this computer and used it as the basis of their incursion into the computer field, starting out on the West Coast in Hawthorne, California.

We found a small computer firm in Minneapolis that was only distinguished by the fact that the president of the firm had a clock in back of him with the face reversed. There was a mirror on the other wall. He was very fond

[1] Perlis had submitted this transcript for consideration for publication in the *Annals of the History of Computing*. It was being edited at the time of his death.

of saying to people when they came in that a good executive never turns his back on his work. He looked at the mirror to find out what the time was. Needless to say, his computers weren't any good. I did try to get a clock like that later on, however.

Ultimately we found a computer for Purdue in California called the Datatron 205. It was manufactured by a firm called Consolidated Electrodynamics. The computer cost about $150,000, which in those days was an awful lot of money.

Behind every computer development at a university, one finds an administration, either an administration that objects to and rebels against and postpones all growth and development, or an administration that fosters. Generally, we don't find them being neutral. At Purdue, the president at that time was Frederick Hutde, an ex-All American football player from the University of Minnesota, who unfortunately was ahead of his time; otherwise he might be running for president today. In any event, I walked in with the three other members of our committee to see the president and told him the sad news: the computer that Purdue needed would cost $125,000 (we had gotten a small discount). I expected the president to throw up his hands and say, “Impossible! No one spends that kind of money on a piece of equipment that no one really understands the benefits of.” Instead, he said, “Give me two days.” In two days he called me on the phone and said, “You have $125,000 from the Purdue Research Foundation. Get the machine.”

From that point on, my attitude has always been that if you are expert, if you are right, the administrators will accede to your wishes. I haven't been disappointed. It's a principle by which I think all of us should operate. By and large, administrators are always looking for people to tell them what they ought to be doing, rather than being confronted with a decision that they have to make on which they have no information with which to make that decision, so the natural technique is to postpone or form another committee. Instead, one really ought to go to them and say, “Do this because it's right.”

The computer arrived at Purdue in 1952, and we began to teach computing in a university to the students and the faculty. But the education then was quite primitive: numerical analysis and electronic circuitry. There was no such subject as programming, and programming languages did not exist as such. We knew of Grace Murray Hopper's work at Univac, but programming languages as such were for the future.

I was interested in a problem in numerical analysis in those days: [Chebychev] approximation. I came up with some potential algorithms that required a great deal of programming experimentation and decided that it was impossible to write these programs in machine language. So we began at Purdue the development of a compiler, which I continued when I moved on to Carnegie in 1956.

In the meantime, IBM decided in marketing the IBM 650 computer to offer a resounding 60% discount for that machine, which made it extraordinarily attractive. The initial budget at Carnegie-and I think the budgets for 650s were similar at that time in other universities-for operating the 650, for supplies, for personnel, and for rental of the machine was $50,000 per year. With that $50,000 we were able to develop a computer center. We continued work on the compiler, which became known as “IT” (Interpretative Translator). I will never forget the miracle of seeing the first program compiled, when an eight-line Gaussian elimination program was expanded into 120 lines of assembly language, which we took as evidence of the power of the compiler, rather than its weakness. It was with absolute amazement that we witnessed this process of generation.

Shortly after that there began a very fruitful exchange between Carnegie and Case and Michigan. The three universities collaborated on the development of a whole sequence of languages based on “IT,” all of the successors being far superior to the original. These successors came cascading out over a period of about a year and a half Don Knuth developed a series at Case Institute; Bernie Galler, Bruce Arden, and Bob Graham developed a series at Michigan.

In 1957 we taught for the first time (although I won't claim at all that there was no previous course offered in the US) a course in programming, independent of numerical analysis and independent of circuitry. The chief issue of this course was: how do you program? In the course we used one of our compiler languages, and we also taught assembly language. We developed what I called structured programming. What we meant by structure was: if you were to write a program in assembly language, write out the structure in a high-level language and then use templates or macros (although we didn't call them that then) to reduce the program to machine language. We were able to reduce the errors tremendously in the resulting programs.

In teaching a course on programming in those days, one of our major problems was: what do you do in such a course and what kind of exercises do you give? There is little point in a course in programming to give exercises in doing square roots by Newton-Raphson or Gaussian elimination or interpolation. They do not reveal the issues in programming. We developed a series of exercises that were based on geometry, where the problems given to students were of a very simple kind that had the blessed virtue that the students did not have to know any deep mathematics at all to understand the intent of the program or the problem and the goal they were to reach. Problems were very simple, such as: if we give you the coordinates of the endpoints of n line segments, how do you tell if these line segments form the consecutive sides of a polygon? The students understood exactly what they were supposed to do, and yet nowhere in their education had they ever been supplied with the techniques, or algorithms as we now call them, by which these tasks could be accomplished. So they were therefore forced to think in what we might call pure programmatic terms. These courses were very successful, and now, of course, such courses are taught everywhere.

The way one used the computer in those days was to line up on Monday morning to sign up for time in blocks half an hour to an hour or even two hours after midnight. A cot was always kept next to the computer, and in the air-conditioning unit of the computer there was a small chamber where one kept beer and Coca-Cola. I lived across the street from Carnegie at that time. One night I received a call, as I often did, at 2 o'clock in the morning, saying, “Come quick! The machine is broken down, and I'm in the middle of the last calculations of my PhD thesis.” I walked over to the Computing Center, which was in the basement of the Graduate School of Industrial Administration building. The hall was dark, but I knew my way along. Suddenly I fell over something. I heard a curse in the dark. I remembered where the light switch was and turned it on. There, to my amazement, were stretched out 20 yards down one corridor and two yards at a right angle, one body after another lying on the floor-including women! I said, “What's going on here?” “Oh, we're waiting to sign up for time on Monday morning.” I recognized a graduate student's wife. I said, 'What are you doing here?” 'Well, my husband's at home working on the thesis and taking care of the kid. I'm holding his place in line.”

I was appalled. It turns out, as is always the case, that the people in charge are the last to know when things are going bad. The next day I went to see the president and said we needed a new computer. The president said, “Fine, as long as it doesn't cost too much.”

In those days, how did one decide what kind of computer to get? The choices were rather sparse. Nevertheless, in looking around, we decided there was really no computer around that was worth getting as a second-generation machine to replace the 650. At an ACM meeting at MIT at that time, I met a man named Dave

Evans, who is now the president of Evans and Sutherland. At that time he was the chief engineer of Bendix. We were walking along the Charles River, and he said, 'We are building a new machine. Let me describe it to you." He did, and it was the G-20, which was based on the idea of a collection of processors tied together in a communications net—works we now hear a great deal of—all the processors operating independently, being controlled by interrupt sequences.

I listened to him, and I said, "We'll take it." Indeed, Carnegie then received delivery of the first G-20 shipped from California, and that machine worked quite well for about five years. When the machine arrived at Carnegie, it came with a full load of software: one binary load routine used by the engineers for testing memory. That was the sole extent of the software. Back in California, they were working on Algol compilers, Fortran compilers, assembly systems—all kinds of things—but none of it was available for us. So we had to build a piece of software for the machine.

At that time we were using at Carnegie a compiler language that was an extension of one developed at Michigan called GAT; we called it GATE (GAT Extended). That came over several steps from "IT." This excellent language predated BASIC in a way and was much better than BASIC: about the same size, about the same speed.

How could one build a compiler quickly? We were fortunate at that time to have with us a visitor from Denmark, Jorn Jensen, who was with the Regnecentralen. Jorn was a magnificent programmer. We conceived of the idea of building a piece of software as though we were carrying out the execution of a program. We put into one room the three programmers who had built the system, with their desks at the vertices of a triangle, if you will. Jensen sat at one desk; he was building the assembler. Arthur Evans sat at another desk; he was building the parser. Harold Van Zoren sat at the third desk; he was building the code generator. All three were being defined simultaneously.

The method of construction worked as follows. Jensen would decide that a certain construction ought to be in assembly language, and he would broadcast to the other two. He would then start independent processes to determine how and where these ideas would be embedded in the code they were writing. When they decided how they would work or what changes would be required, working in good code-team fashion, they suspended and broadcast back to Jensen their proposals. Jensen would drop what he was doing and start another independent process. The amount of code that each wrote turned out to be of the order of about 2,000-3,000 machine language instructions. It turned out that at that size of code, such a technique worked magnificently. Each of the programmers could keep two to four processes in the air simultaneously, and changes progressed very fast. All three parts were completed at the same time. Jensen debugged the assembly language on the computer, simultaneously with the debugging of the parser and the code generator. We had here an example of the construction of a piece of software as though we were executing a real-time multiprocessor program.

About that time in the late 1950s, Algol came into being. Algol has never turned out to be as successful a language in the US as we would have liked at that time. Maybe that's a blessing in disguise. In any event, the Algol process in the 1950s was extraordinarily important in that it brought together computer scientists (although we didn't go by that name then) from the US and Europe who worked closely together to produce something that would be, it was hoped, an international standard. We now know more about the difficulties of getting people to accept progamming languages and programming systems. Saul Rosen has amply demonstrated the slowness with which people are going to change what they have been doing. Algol as such never flew in the US; it did in Europe, however, and a great deal of Europe's success in computing is a consequence of the Algol effort in the 1950s.

One of the things about the 1950s that was pronounced was the sparseness of people. In almost every university you could find one—and if they were fortunate two—persons who knew anything at all about computing. They had to carry the burden of teaching everyone else at the university the benefits of computing in those days. Now everybody knows about computing. I used to use as a yardstick the conversations I would hear on airplanes. In the late 1950s, when you would take a flight, you would never hear anyone on the airplane speaking to anyone else about Fortran or Cobol, about software, or anything of that sort. Nowadays, 50% of the people on airplanes are writing programs, or reading computing books, or traveling to a computing meeting, or coming back from one. Another yardstick we used to use in those days was an estimate of the largest population of a city in the US that had no computers in it. It was about 15,000 in the late 1950s. Today, I doubt if that number would even reach 300. We really are in the computer age.

In the 1950s we learned, I think, a great deal. At the time we weren't aware that we were learning it, but the things we learned in those days have become the basis of computer science as we now know it, the basis of the use of computers in industry as we now see it. It is too bad that we didn't know what we were learning at the time, because we might have saved everyone a lot of grief and prevented people from going down the wrong path-except that in computing there is no wrong path. One of the things we learned about computing in the 1950s was that there are no bounds to the subject. It cannot be put into a tidy receptacle. Everywhere that computing has been embedded in some other discipline, it has not flowered. Computing is not part of electrical engineering; it is not part of mathematics; it is not part of industrial administration. Computing belongs by itself. The reason this is the computer age is precisely because of that.

One of the things we have learned since the 1950s is that computer science is really nothing more or less than the study of all the phenomena that arise around computers. These phenomena are generated from preceding phenomena, and one cannot predict what they will be. I have been collecting a number of epigrams about computing. One of them is that if a professor tells you that computer science is x but not y, have compassion for his students.

Herb Simon, when he said that there would be a chess champion in the 1960s, had no idea that this would not occur. He believed that it would. Since that time we have learned that there is a tremendous difference between the ease with which we can master syntax and the difficulty with which we take care of semantics. I am fond of saying that many a good idea is never heard from again once it embarks on a voyage in the semantic gulf.

Certainly, one of our goals in developing chess programs-programs that understand natural language, programs that play any kind of game-is to write programs that learn. What we have found is this: when we write programs that learn, it turns out that we do and they don't.

A previous speaker mentioned VLSI, the great new hope. All that comes to my mind when I hear about VLSI is, “Isn't it wonderful we can now pack 100 ENIACs in one square centimeter?” VLSI is going to turn out to be of enormous value in special-purpose computers. We all know that computers are ubiquitous, and soon they will sink beneath notice, like the electric motor. How many of you know how many electric motors you have in your kitchen? In five years the same question will be asked: how many computers do you have in your home?

Almost all of them you won't see. But the general-purpose computers are going to measure the future of computing, not the special purpose ones. VLSI places a tremendous burden on those of us who are concerned with imaginative uses of computing to find out how to control and to utilize the great benefit that comes from micro circuitry.

In the 1950s we often asked ourselves, “What are our goals?” As computers were applied in new areas, as new computers kept coming out, as we saw that there were waves of students who wanted to learn about computers, we visualized a world in which everything would be done by computers, everyone would be literate in computers, but what is the goal? I don't really know whether a science depends on having a major problem. Physicists are very fortunate in that they can create their major problems quite easily; all they do is say that there must be some new fundamental particle. Society, because of nuclear energy, has cheerfully paid for this search. Biologists, of course, have suddenly come up with a major problem: genetic engineering.

What is the major problem in computer science? Is there one? Can there ever be any? In thinking about it, I have come to the conclusion that if there is any major problem, it is the problem of bridging the gap between neuroscience and psychology. If one looks at neuroscientists, one finds that they work very deep inside the nervous system at an extremely primitive level; they work on neurons. It is as though computer scientists spent all their time looking at gates. On the other end of the spectrum is the psychologist, who sees man as an entity, as a totality, trying to understand what makes him tick. The psychologist's weapons are gross and very, very macroscopic. How can we create a bridge between neuroscience and psychology? My conclusion is that computer science is the discipline that will provide this bridge, and the bridge will come from the study of software. *There is* a major problem worth sinking your teeth into. Software-collections of symbols that execute processes-is our external way of trying to model the way thinking goes on. Of course, we are very primitive. Nevertheless, it seems to us that is the clue: computer science—if any discipline will—will ultimately provide us the bridge that tells us how the neurons, working as gates, pass signals by the billions in such a way that our brain causes us to behave in a way that psychologists can say, “Ah! I understand.”

We've learned some other things, too. Some of them are so trivial and so obvious that we've ignored them. For instance, syntax is extraordinarily expensive. Nothing slows programming down as a plethora of syntax. When one looks at a new language such as Ada, one observes that the disease is in its final stages there. It's a language that is redolent with syntax. Why is syntax so expensive? Well, when you build systems that include other systems in which you wish to generate programs, every time you work with heavy syntax you must build heavy parsers and all the software that has to be carried along. Everything slows to a crawl. It is obvious why a language like LISP has proved so fruitful in artificial intelligence. It has no syntax. Parsing is automatic, and when one wishes to build up languages within, systems within, they all look the same. The same thing that has happened in the human body-that has worked so well in the development of all complex organisms-happens there. Why is it that we have not extended the same point of view toward other programming languages? The reason, of course, is that in these other programming languages, we always think of having built systems at the design table, that once having been built they will stay that way forever. All of our bitter experience tells us they do not. I think we learned some of those lessons in the 1950s, and we neglected our role as teachers in not passing them along.

I would like to close with one last remark. We live in a country in which computers have flowered. When I ask people, “Why is it that computers have flowered in the United States?” the answer is usually “Lots of bucks; the Defense Department; IBM”—as if these are natural phenomena that existed all the time. In point of fact, I think there is another reason. I think computers have flowered in this country because of our national style for accomplishing things. This country has always supported entrepreneurial activities-people who have ideas and are willing to sweat to bring them about. Computers flower in such an environment because on a computer everything is possible, although we have since learned that nothing is easy. Compare that with, for example, the Soviet Union. The Soviet Union, having a large centralized society, needs computers much worse than we do. Yet they are totally unable to produce them, totally unable to apply them in anywhere near the profusion that we find here. They have the wrong kind of society for the instrument they most badly need. Tony Hoare, in his Turing Award lecture, also talked of a style of life. The European style has always worked very hard at producing elegant constructions—devices that fit. Holland, for example, is a very small country, and when going to Holland one finds that the instruments they use are also small. Russia is an enormous country; they use large instruments. So it never surprised me that the Russians decided to build a compiler once that would simultaneously compile PL/1, Cobol, and Algol. Holland gave us Edsger Dijkstra, with structured programming and natural Dutch construction.

Computers really are an explanation of what we are. Our goal, of course, ought to be to use this tool of our age-as Saul Rosen has said, the computer age-to help us understand why we are the way we are, today and forever.

OUOTATIONS[1]

"Any noun can be verbed."

"People in charge are the last to know when things go bad."

"It is easier to get forgiveness than permission."

"The goal of computation is the emulation of our synthetic abilities, not the understanding of our analytical ones."

"The best is the enemy of the good." (Perlis 1981)

"Fools ignore complexity; Pragmatists suffer it; Some can avoid it; Geniuses remove it." (Perlis 1982)

"Both knowledge and wisdom extend man's reach. Knowledge led to computers, wisdom to chopsticks."

"If a professor tells you that computer science is *x* but not *y*, have compassion for his students."

"When we write programs that learn, it turns out that we do and they don't."

BIBLIOGRAPHY

Biographical

Anon., "Alan J. Perlis, 1922-90," *Comm. ACM*, Vol. 33, No. 5, May 1990, pp. 604-605.

[1] Perlis made a habit of adding pithy sayings to his talks, lectures, and presentations, and thus may be one of the most quoted computer pioneers-along with Richard Hamming.

Perlis, Alan J., “The American Side of the Development of Algol,” in Wexelblat, Richard L., ed., *History of Programming Languages,* Academic Press, New York, 1981.

Perlis, Alan J., “Two Thousand Words and Two Thousand Ideas—The 650 at Carnegie,” *Ann. Hist. Comp.*, Vol. 8, No. 1, Jan. 1986, pp. 62-65.

Significant Publications

Carr, John W., III, and Alan J. Perlis, “A Comparison of Large-Scale Calculators,” *Control Engineering,* Vol. 3, Feb. 1956, pp. 84-96.

Perlis, Alan J., et al., *Software Metrics,* MIT Press, Cambridge, Mass., 1981.

Perlis, Alan J., “Epigrams on Programming,” *ACM SIGPLAN Notices,* Vol. 17, No. 9, Sept. 1982, pp. *7-13.*

UPDATES

Portrait changed (MRW, 2013)