Epigrams on Programming

Alan J. Perlis (1922-1990)

Carnegie Mellon is extremely grateful to Alan for his fundamental contribution to the University — starting the Computer Science Department in 1965. He was one of the three founders, together with Herb Simon and Allen Newell. Although computer science at Carnegie Mellon actually came out of the business school, it was Alan Perlis' doing to create an independent department and give it a place close to mathematics and the sciences.

Alan is vividly remembered by the senior faculty, and remembered very fondly for his unique contribution not only as the first department head and organizer, but even more for the team spirit he managed to establish. He was the one who lived by the reasonable person principle — creating a supportive and constructive environment lacking much of the frustrating bureaucracy and aggravating competitiveness that one finds in many other places. Alan was the one who started the policy of sharing research funding in such a way that incoming Ph.D. students could choose an advisor or a research topic mainly depending on their interests and not primarily on the temporary funding situation of individual faculty.

Alan will also be remembered as an inspired teacher and as an innovative educator...

— A. Nico Habermann, Alan J. Perlis Professor of Computer Science

Definition 1: epigram — a concise poem dealing pointedly and often satirically with a single thought or event and often ending with an ingenious turn of thought; — a terse, sage, or witty and often paradoxical saying.

Epigrams on Programming

The phenomena surrounding computers are diverse and yield a surprisingly rich base for launching metaphors at individual and group activities. Conversely, classical human endeavors provide an inexhaustible source of metaphor for those of us who are in labor within computation. Such relationships between society and device are not new, but the incredible growth of the computer's influence (both real and implied) lends this symbiotic dependency a vitality like a gangly youth growing out of his clothes within an endless puberty.

The epigrams that follow attempt to capture some of the dimensions of this traffic in imagery that sharpens, focuses, clarifies, enlarges and beclouds our view of this most remarkable of all mans' artifacts, the computer.

— Alan J. Perlis, while at Yale University

1. One man's constant is another man's variable.
2. Functions delay binding: data structures induce binding.
   Moral: Structure data late in the programming process.
3. Syntactic sugar causes cancer of the semi-colons.
4. Every program is a part of some other program and rarely fits.
5. If a program manipulates a large amount of data, it does so in a small number of ways.
6. Symmetry is a complexity reducing concept (co-routines include sub-routines); seek it everywhere.
7. It is easier to write an incorrect program than understand a correct one.
8. A programming language is low level when its programs require attention to the irrelevant.
9. It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.
10. Get into a rut early: Do the same processes the same way. Accumulate idioms. Standardize. The only difference(!) between Shakespeare and you was the size of his idiom list — not the size of his vocabulary.
11. If you have a procedure with 10 parameters, you probably missed some.
12. Recursion is the root of computation since it trades description for time.
13. If two people write exactly the same program, each should be put in microcode and then they certainly won't be the same.
14. In the long run every program becomes rococo — then rubble.
15. Everything should be built top-down, except the first time.
16. Every program has (at least) two purposes: the one for which it was written and another for which it wasn't.
17. If a listener nods his head when you're explaining your program, wake him up.
18. A program without a loop and a structured variable isn't worth writing.
19. A language that doesn't affect the way you think about programming is not worth knowing.
20. Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.
21. Optimization hinders evolution.
22. A good system can't have a weak command language.
23. To understand a program you must become both the machine and the program.
24. Perhaps if we wrote programs from childhood on, as adults we'd be able to read them.
25. One can only display complex information in the mind. Like seeing, movement of flow or alteration of view is more important than the static picture, no matter how lovely.

26. There will always be things we wish to say in our programs that in all known languages can only be said poorly.

27. Once you understand how to write a program get someone else to write it.

28. Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?

29. For systems, the analogue of a face-lift is to add to the control graph an edge that creates a cycle, not just an additional node.

30. In programming, everything we do is a special case of something more general — and often we know it too quickly.

31. Simplicity does not precede complexity, but follows it.

32. Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis.

33. The 11th commandment was "Thou Shalt Compute" or "Thou Shalt Not Compute" — I forget which.

34. The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information.

35. Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers.

36. The use of a program to prove the 4-color theorem will not change mathematics — it merely demonstrates that the theorem, a challenge for a century, is probably not important to mathematics.

37. The most important computer is the one that rages in our skulls and ever seeks that satisfactory external emulator. The standardization of real computers would be a disaster — and so it probably won't happen.

38. Structured Programming supports the law of the excluded muddle.

39. Re: graphics: A picture is worth 10K words — but only those to describe the picture. Hardly any sets of 10K words can be adequately described with pictures.

40. There are two ways to write error-free programs; only the third one works.

41. Some programming languages manage to absorb change, but withstand progress.

42. You can measure a programmer's perspective by noting his attitude on the continuing vitality of FORTRAN.

43. In software systems it is often the early bird that makes the worm.

44. Sometimes I think the only universal in the computing field is the fetch-execute cycle.

45. The goal of computation is the emulation of our synthetic abilities, not the understanding of our analytic ones.

46. Like punning, programming is a play on words.

47. As Will Rogers would have said, "There is no such thing as a free variable."

48. The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

49. Giving up on assembly language was the apple in our Garden of Eden: Languages whose use squanders machine cycles are sinful. The LISP machine now permits LISP programmers to abandon bra and fig-leaf.

50. When we understand knowledge-based systems, it will be as before — except our finger-tips will have been singed.

51. Bringing computers into the home won't change either one, but may revitalize the corner saloon.

52. Systems have sub-systems and sub-systems have sub-systems and so on ad finitum — which is why we're always starting over.

53. So many good ideas are never heard from again once they embark in a voyage on the semantic gulf.

54. Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

55. A LISP programmer knows the value of everything, but the cost of nothing.

56. Software is under a constant tension. Being symbolic it is arbitrarily perfectible; but also it is arbitrarily changeable.

57. It is easier to change the specification to fit the program than vice versa.

58. Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

59. In English every word can be verbed. Would that it were so in our programming languages.

60. Dana Scott is the Church of the Lattice-Way Saints.

61. In programming, as in everything else, to be in error is to be reborn.

62. In computing, invariants are ephemeral.

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

64. Often it is means that justify ends: Goals advance technique and technique survives even when goal structures crumble.

65. Make no mistake about it: Computers process numbers — not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity.

66. Making something variable is easy. Controlling duration of constancy is the trick.

67. Think of all the psychic energy expended in seeking a fundamental distinction between "algorithm" and "program."

68. If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?

69. In a 5 year period we get one superb programming language. Only we can't control when the 5 year period will begin.

70. Over the centuries the Indians developed sign language for communicating phenomena of interest. Programmers from different tribes (FORTRAN, LISP, ALGOL, SNOBOL, etc.) could use one that doesn't require them to carry a blackboard on their ponies.

71. Documentation is like term insurance: It satisfies because almost no one who subscribes to it depends on its benefits.

72. An adequate bootstrap is a contradiction in terms.

73. It is not a language's weaknesses but its strengths that control the gradient of its change: Alas, a language never escapes its embryonic sac.

74. Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to always see it as a soap bubble?

75. Because of its vitality, the computing field is always in desperate need of new cliches: Banality soothes our nerves.

76. It is the user who should parametrize procedures, not their creators.

77. The cybernetic exchange between man, computer and algorithm is like a game of musical chairs: The frantic search for balance always leaves one of the three standing ill at ease.

78. If your computer speaks English it was probably made in Japan.

79. A year spent in artificial intelligence is enough to make one believe in God.

80. Prolonged contact with the computer turns mathematicians into clerks and vice versa.

81. In computing, turning the obvious into the useful is a living definition of the word "frustration."

82. We are on the verge: Today our program proved Fermat's next-to-last theorem!

83. What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak.

84. Motto for a research laboratory: What we work on today, others will first think of tomorrow.

85. Though the Chinese should adore APL, it's FORTRAN they put their money on.

86. We kid ourselves if we think that the ratio of procedure to data in an active data-base system can be made arbitrarily small or even kept small.

87. We have the mini and the micro computer. In what semantic niche would the pico computer fall?

88. It is not the computer's fault that Maxwell's equations are not adequate to design the electric motor.

89. One does not learn computing by using a hand calculator, but one can forget arithmetic.

90. Computation has made the tree flower.

91. The computer reminds one of Lon Chaney — it is the machine of a thousand faces.

92. The computer is the ultimate polluter: Its feces are indistinguishable from the food it produces.

93. When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.

94. Interfaces keep things tidy, but don't accelerate growth: Functions do.

95. Don't have good ideas if you aren't willing to be responsible for them.

96. Computers don't introduce order anywhere as much as they expose opportunities.

97. When a professor insists computer science is X but not Y, have compassion for his graduate students.

98. In computing, the mean time to failure keeps getting shorter.

99. In man-machine symbiosis, it is man who must adjust: The machine can't.

100. We will never run out of things to program as long as there is a single program around.

101. Dealing with failure is easy: Work hard to improve. Success is also easy to handle: You've solved the wrong problem. Work hard to improve.

102. One can't proceed from the informal to the formal by formal means.

103. Purely applicative languages are poorly applicable.

104. The proof of a system's value is its existence.

105. You can't communicate complexity, only an awareness of it.

106. It's difficult to extract sense from strings, but they're the only communication coin we can count on.

107. The debate rages on: Is PL/I Bactrian or Dromedary?

108. Whenever two programmers meet to criticize their programs, both are silent.

109. Think of it! With VLSI we can pack 100 ENIACs in 1 sq. cm.

110. Editing is a rewording activity.

111. Why did the Roman Empire collapse? What is the Latin for office automation?

112. Computer Science is embarrassed by the computer.

113. The only constructive theory connecting neuroscience and psychology will arise from the study of software.

114. Within a computer natural language is unnatural.

115. Most people find the concept of programming obvious, but the doing impossible.

116. You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.

117. It goes against the grain of modern education to teach children to program. What fun is there in making plans, acquiring discipline in organizing thoughts, devoting attention to detail and learning to be self-critical?

118. If you can imagine a society in which the computer-robot is the only menial, you can imagine anything.

119. Programming is an unnatural act.

120. Adapting old programs to fit new machines usually means adapting new machines to behave like old ones.

121. In seeking the unattainable, simplicity only gets in the way. If there are epigrams, there must be meta-epigrams.

122. Epigrams are interfaces across which appreciation and insight flow.

123. Epigrams parametrize auras.

124. Epigrams are macros, since they are executed at read time.

125. Epigrams crystallize incongruities.

126. Epigrams retrieve deep semantics from a data base that is all procedure.

127. Epigrams scorn detail and make a point: They are a superb high-level documentation.

128. Epigrams are more like vitamins than protein.

129. Epigrams have extremely low entropy.

130. The last epigram? Neither eat nor drink them, snuff epigrams.

Who Was Alan J. Perlis?

Reknowned as an educator and as a programming system and language developer, Alan Perlis was the first Editor-in-Chief of the Communications of ACM (1958-1962), President of ACM from 1962 to 1964, and the first recipient, in 1966, of ACM's Turing Award.

He was born in Pittsburgh, PA on April 1, 1922. He received a B.S. in Chemistry in 1942 from Carnegie Institute of Technology (now Carnegie Mellon University). After serving in USAAF from 1942 to 1945 (1'st Lt.-ETO), he did post-graduate work at the California Institute of Technology in 1946-1947, receiving an M.S. in Mathematics. He received his Ph.D. in Mathematics from the Massachusetts Institute of Technology in 1950 and spent 1951 as a Research Mathematician in the Multi-Machine Computing Laboratory of the Ballistic Research Laboratories at Aberdeen Proving Grounds. He then returned to the Massachusetts Institute of Technology to work at Project Whirlwind, developing programs throughout 1952.

From September 1952 through 1956 he was Assistant Professor of Mathematics at Purdue University and director of a computer center consisting of an I.B.M. CPC (installed October 1952) that was replaced by a Datatron 205 in the spring of 1954. In 1955 he headed a group at Purdue that defined a language IT (for "Internal Translator") and developed a compiler for it. This work was continued on the IBM 650 when he took the position of Associate Professor of Mathematics and Director of the Computation Center at Carnegie Tech in 1956. A version of IT was running at the center in November 1956, and within a year it was in wide use on 650's at a number of university computing centers. (The one for Purdue's Datatron was also running in 1957; the succession of algebraic language compilers and assemblers were designed and built for the 650 by Perlis, Joseph Smith, Harold Van Zoeren, and Arthur Evans).

"Research in programming languages dominated my life from then on" said Perlis at the ACM SIGPLAN History of Programming Languages Conference in Los Angeles, California, June 1-3, 1978[1].

In 1957, ACM President John W. Carr, III, appointed Perlis chairman of a programming language committee and head of a subcommittee to meet with a similar subcommittee of GAMM (Gesellschaft fur angewandte Mathematik und Mechanik) in Europe to design a "universal algebraic language". The group of eight met in Zurich and specified what was finally called ALGOL-58. The report on Algol-58 by Perlis and Samelson was the basis of a formal specification of Algol-60 by thirteen international scientists (including Perlis, of course).

Meanwhile, at Carnegie Tech a course on programming, distinct from numerical analysis and available to all undergraduate students, was introduced. Perlis was Professor and Chairman of the Mathematics Department at Carnegie Tech while continuing as Director of the Computation Center from 1960 to 1964, and in 1962 he became Co-Chairman of an interdisciplinary doctoral program in Systems and Communications Sciences. This led to the establishment of an ARPA supported program in computer science, which, in turn, led in 1965 to a graduate department of Computer Science. He was its first chairman, remaining so until 1971. More than a dozen of its graduates received their degrees from him; these include well-known leaders in teaching and research from coast to coast and a few abroad.

During the 1960's he developed such extensions to Algol as Formula Algol, an Algol manipulator of formal mathematical expressions, and LCC, a form of Algol adapted to interactive incremental programming.

After a tour as visiting scientist at the Mathematische Centrum in Amsterdam, Netherlands (1965-1966), he held the title of University Professor of Computer Science and Mathematics at Carnegie Tech.

In 1971 he joined the new Computer Science Department at Yale University as Eugene Higgins Professor of Computer Science, and played a leading role in developing that department. He was its chairman for the 1976-1977 year and from 1978-1980, serving again as acting chairman in 1987. During these years he developed new computer courses at both the graduate and undergraduate levels. From 1977-1978 he was Gordon and Betty Moore Professor of Computer Science at the California Institute of Technology. More than a dozen graduate students at Yale received their doctorates under his supervision.

He received honorary degrees from Davis and Elkins College, Purdue University, Waterloo University, and Sacred Heart University. Beside dozens of papers, written alone and with co-workers, on programming languages, systems and developments, he was the author of *Introduction to Computer Science*, Harper and Row (1972,1975); and, with B.A. Galler, of *A View of Programming Languages*, Addison-Wesley, 1970.

Throughout his career he continued to serve on committees and boards, national and international, for medical research, natural language processing and translation, and, for the last twenty years, on software engineering.

Perlis will be remembered as much for his personal warmth and pervasive joy as for specific technical achievements. He believed that computing should be fun; this contagious enthusiasm set the tone of both his research and his teaching.

— Courtesy Communications of the ACM, 1990

---

[1]See *History of Programming Languages*, ed. R.L. Wexelblatt, Academic Press, 1981, p. 171.