NIKLAUS WIRTH
1984 ACM A.M. TURING AWARD RECIPIENT

Niklaus Wirth

Niklaus Wirth of the Swiss Federal Institute of Technology (ETH) was presented the 1984 ACM A.M. Turing Award at the Association's Annual Conference in San Francisco in October in recognition of his outstanding work in developing a sequence of innovative computer languages: Euler, ALGOL-W, Modula, and Pascal. Pascal, in particular, has become significant pedagogically and has established a foundation for future research in the areas of computer language, systems, and architecture. The hallmarks of a Wirth language are its simplicity, economy of design, and high-quality engineering, which result in a language whose notation appears to be a natural extension of algorithmic thinking rather than an extraneous formalism.

Wirth's ability in language design is complemented by a masterful writing ability. In the April 1971 issue of *Communications of the ACM*, Wirth published a seminal paper on Structured Programming ("Program Development by Stepwise Refinement") that recommended top-down structuring of programs (i.e., successively refining program stubs until the program is fully elaborated). The resulting elegant and powerful method of exposition remains interesting reading today even after the furor over Structured Programming has subsided. Two later papers, "Toward a Discipline of Real-Time Programming" and "What Can We Do About the Unnecessary Diversity of Notation" (published in *CACM* in August and November 1974, respectively), speak to Wirth's consistent and dedicated search for an adequate language formalism.

The Turing Award, the Association's highest recognition of technical contributions to the computing community, honors Alan M. Turing, the English mathematician who defined the computer prototype Turing machine and helped break German ciphers during World War II.

Wirth received his Ph.D. from the University of California at Berkeley in 1963 and was Assistant Professor at Stanford University until 1967. He is Professor at the ETH Zurich since 1968; from 1982 until 1984 he was Chairman of the Division of Computer Science (Informatik) at ETH. Wirth's recent work includes the design and development of the personal computer Lilith in conjunction with the Modula-2 language. In his lecture, Wirth presents a short history of his major projects, drawing conclusions and highlighting the principles that have guided his work.

FROM PROGRAMMING LANGUAGE DESIGN TO COMPUTER CONSTRUCTION

From NELIAC (via ALGOL 60) to Euler and ALGOL W, to Pascal and Modula-2, and ultimately Lilith, Wirth's search for an appropriate formalism for systems programming yields intriguing insights and surprising results.

NIKLAUS WIRTH

It is a great pleasure to receive the Turing Award, and both gratifying and encouraging to receive appreciation for work done over so many years. I wish to thank ACM for bestowing upon me this prestigious award. It is particularly fitting that I receive it in San Francisco, where my professional career began.

Soon after I received notice of the award, my feeling of joy was tempered somewhat by the awareness of having to deliver the Turing lecture. For someone who is an engineer rather than an orator or preacher, this obligation causes some noticeable anxiety. Foremost among the questions it poses is the following: What do people expect from such a lecture? Some will wish to gain technical insight about one's work, or expect an assessment of its relevance or impact. Others will wish to hear how the ideas behind it emerged. Still others expect a statement from the expert about future trends, events, and products. And some hope for a frank assessment of the present engulfing us, either glorifying the monumental advance of our technology or lamenting its cancerous side effects and exaggerations.

In a period of indecision, I consulted some previous Turing lectures and saw that a condensed report about the history of one's work would be quite acceptable. In order to be not just entertaining, I shall try to summarize what I believe I have learned from the past. This choice, frankly, suits me quite well, because neither do I pretend to know more about the future than most others, nor do I like to be proven wrong afterwards. Also, the art of preaching about current achievements and misdeeds is not my primary strength. This does not imply that I observe the present computing scene without concern, particularly its tumultuous hassle with commercialism.

Certainly, when I entered the computing field in 1960, it was neither so much in the commercial limelight nor in academic curricula. During my studies at the Swiss Federal Institute of Technology (ETH), the only mention I heard of computers was in an elective course given by Ambros P. Speiser, who later became the president of IFIP. The computer ERMETH developed by him was hardly accessible to ordinary students, and so my initiation to the computing field was delayed until I took a course in numerical analysis at Laval University in Canada. But alas, the Alvac III E machinery was out of order most of the time, and exercises in programming remained on paper in the form of untested sequences of hexadecimal codes.

My next attempt was somewhat more successful: At Berkeley, I was confronted with Harry Huskey's pet machine, the Bendix G-15 computer. Although the Bendix G-15 provided some feeling of success by producing results, the gist of the programming art appeared to be the clever allocation of instructions on the drum. If you ignored the art, your programs could well run slower by a factor of one hundred. But the educational benefit was clear: You could not afford to ignore the least little detail. There was no way to cover up deficiencies in your design by simply buying more memory. In retrospect, the most attractive feature was that every detail of the machine was visible and could be

understood. Nothing was hidden in complex circuitry, silicon, or a magic operating system.

On the other hand, it was obvious that computers of the future had to be more effectively programmable. I therefore gave up the idea of studying how to design hardware in favor of studying how to use it more elegantly. It was my luck to join a research group that was engaged in the development—or perhaps rather improvement—of a compiler and its use on an IBM 704. The language was called *NELIAC*, a dialect of ALGOL 58. The benefits of such a "language" were quickly obvious, and the task of automatically translating programs into machine code posed challenging problems. This is precisely what one is looking for when engaged in the pursuit of a Doctorate. The compiler, itself written in NELIAC, was a most intricate mess. The subject seemed to consist of 1 percent science and 99 percent sorcery, and this tilt had to be changed. Evidently, programs should be designed according to the same principles as electronic circuits, that is, clearly subdivided into parts with only a few wires going across the boundaries. Only by understanding one part at a time would there be hope of finally understanding the whole.

This attempt received a vigorous starting impulse from the appearance of the report on *ALGOL 60*. ALGOL 60 was the first language defined with clarity; its syntax was even specified in a rigorous formalism. The lesson was that a clear specification is a necessary but not sufficient condition for a reliable and effective implementation. Contact with Aadrian van Wijngaarden, one of ALGOL's codesigners, brought out the central theme more distinctly: Could ALGOL's principles be condensed and crystallized even further?

Thus began my adventures in programming languages. The first experiment led to a dissertation and the language *Euler*—a trip with the bush knife through the jungle of language features and facilities. The result was academic elegance, but not much of practical utility—almost an antithesis of the later data-typed and structured programming languages. But it did create a basis for the systematic design of compilers that, so was the hope, could be extended without loss of clarity to accommodate further facilities.

Euler caught the attention of the IFIP Working Group that was engaged in planning the future of ALGOL. The language ALGOL 60, designed by and for numerical mathematicians, had a systematic structure and a concise definition that were appreciated by mathematically trained people but lacked compilers and support by industry. To gain acceptance, its range of application had to be widened. The Working Group assumed the task of proposing a successor and soon split into two camps. On one side were the ambitious who wanted to erect another milestone in language design, and, on the other, those who felt that time was pressing and that an adequately extended ALGOL 60 would be a productive endeavor. I belonged to this second party and submitted a proposal that lost the election. Thereafter, the proposal was improved with contributions from Tony Hoare (a member of the same group) and implemented on Stanford University's first IBM 360. The language later became known as *ALGOL W* and was used in several universities for teaching purposes.

A small interlude in this sizable implementation effort is worth mentioning. The new IBM 360 offered only assembler code and, of course, FORTRAN. Neither particularly were loved, either by me or my graduate students, as a tool for designing a compiler. Hence, I mustered the courage to define yet another language in which the ALGOL compiler would be described: A compromise between ALGOL and the facilities offered by the assembler, it would be a machine language with ALGOL-like statement structures and declarations. Notably, the language was defined in a couple of weeks; I wrote the cross compiler on the Burroughs B-5000 computer within four months, and a diligent student transported it to the IBM 360 within an equal period of time. This preparative interlude helped speed up the ALGOL effort considerably. Although envisaged as serving our own immediate needs and to be discarded thereafter, it quickly acquired its own momentum. *PL360* became an effective tool in many places and inspired similar developments for other machines.

Ironically, the success of PL360 was also an indication of ALGOL W's failure. ALGOL's range of application had been widened, but as a tool for systems programming, it still had evident deficiencies. The difficulty of resolving many demands with a single language had emerged, and the goal itself became questionable. PL/1, released around this time, provided further evidence to support this contention. The *Swiss army knife* idea has its merits, but if driven to excess, the knife becomes a millstone. Also, the size of the ALGOL-W compiler grew beyond the limits within which one could rest comfortably with the feeling of having a grasp, a mental understanding, of the whole program. The desire for a more concise yet more appropriate formalism for systems programming had not been fulfilled. Systems programming requires an efficient compiler generating efficient code that operates without a fixed, hidden, and large so-called run-time package. This goal had been missed by both ALGOL-W and PL/1, both because the languages were complex and the target computers inadequate.

In the fall of 1967, I returned to Switzerland. A year later, I was able to establish a team with three assistants to implement the language that later became known as *Pascal*. Freed from the constraints of obtaining a committee consensus, I was able to concentrate on including the features I myself deemed essential and excluding those whose implementation effort I judged to be incommensurate with the ultimate benefit. The constraint of severely limited manpower is sometimes an advantage.

Occasionally, it has been claimed that Pascal was designed as a language for teaching. Although this is correct, its use in teaching was not the only goal. In

fact, I do not believe in using tools and formalisms in teaching that are inadequate for any practical task. By today's standards, Pascal has obvious deficiencies for programming large systems, but 15 years ago it represented a sensible compromise between what was desirable and what was effective. At ETH, we introduced Pascal in programming classes in 1972, in fact against considerable opposition. It turned out to be a success because it allowed the teacher to concentrate more heavily on structures and concepts than features and peculiarities, that is, on principles rather than techniques.

Our first Pascal compiler was implemented for the CDC 6000 computer family. It was written in Pascal itself. No PL6000 was necessary, and I considered this a substantial step forward. Nonetheless, the code generated was definitely inferior to that generated by FORTRAN compilers for corresponding programs. Speed is an essential and easily measurable criterion, and we believed the validity of the high-level language concept would be accepted in industry only if the performance penalty were to vanish or at least diminish. With this in mind, a second effort—essentially a one-man effort—was launched to produce a high-quality compiler. The goal was achieved in 1974 by Urs Ammann, and the compiler was thereafter widely distributed and is being used today in many universities and industries. Yet the price was high; the effort to generate good (i.e., not even optimal) code is proportional to the mismatch between language and machine, and the CDC 6000 had certainly not been designed with high-level languages in mind.

Ironically again, the principal benefit turned up where we had least expected it. After the existence of Pascal became known, several people asked us for assistance in implementing Pascal on various other machines, emphasizing that they intended to use it for teaching and that speed was not of overwhelming importance. Thereupon, we decided to provide a compiler version that would generate code for a machine of our own design. This code later became known as *P-code*. The P-code version was easy to construct because the new compiler was developed as a substantial exercise in structured programming by stepwise refinement and therefore the first few refinement steps could be adopted unchanged. Pascal-P proved enormously successful in spreading the language among many users. Had we possessed the wisdom to foresee the dimensions of this movement, we would have put more effort and care into designing and documenting P-code. As it was, it remained a side effort to honor the requests in one concentrated stride. This shows that even with the best intentions one may choose one's goals wrongly.

But Pascal gained truly widespread recognition only after Ken Bowles in San Diego recognized that the P-system could well be implemented on the novel microcomputers. His efforts to develop a suitable environment with integrated compiler, filer, editor, and debugger caused a breakthrough: Pascal became available to thousands of new computer users who were not burdened with acquired habits or stifled by the urge to stay compatible with software of the past.

In the meantime, I terminated work on Pascal and decided to investigate the enticing new subject of multiprogramming, where Hoare had laid respectable foundations and Brinch Hansen had led the way with his *Concurrent Pascal*. The attempt to distill concrete rules for a multiprogramming discipline quickly led me to formulate them in terms of a small set of programming facilities. In order to put the rules to a genuine test, I embedded them in a fragmentary language, whose name was coined after my principal aim: modularity in program systems. The module later turned out to be the principal asset of this language; it gave the abstract concept of information hiding a concrete form and incorporated a method as significant in uniprogramming as in multiprogramming. Also, *Modula* contained facilities to express concurrent processes and their synchronization.

By 1976, I had become somewhat weary of programming languages and the frustrating task of constructing good compilers for existing computers that were designed for old-fashioned "by-hand" coding. Fortunately, I was given the opportunity to spend a sabbatical year at the research laboratory of Xerox Corporation in Palo Alto, where the concept of the powerful personal workstation had not only originated but was also put into practice. Instead of sharing a large, monolithic computer with many others and fighting for a share via a wire with a 3KHz bandwidth, I now used my own computer placed under my desk over a 15MHz channel. The influence of a 5000-fold increase in anything is not foreseeable; it is overwhelming. The most elating sensation was that after 16 years of working for computers, the computer now seemed to work for me. For the first time, I did my daily correspondence and report writing with the aid of a computer, instead of planning new languages, compilers, and programs for others to use. The other revelation was that a compiler for the language *Mesa*, whose complexity was far beyond that of Pascal, could be implemented on such a workstation. These new working conditions were so many orders of magnitude above what I had experienced at home that I decided to try to establish such an environment there as well.

I finally decided to dig into hardware design. This decision was reinforced by my old disgust with existing computer architectures that made life miserable for a compiler designer with a bent toward systematic simplicity. The idea of designing and building an entire computer system consisting of hardware, microcode, compiler, operating system, and program utilities quickly took shape in my imagination—a design that would be free from any constraint to be compatible with a PDP-11 or an IBM 360, or FORTRAN, Pascal, UNIX, or whatever other current fad or committee standard there might be.

But a sensation of liberation is not enough to succeed

in a technical project. Hard work, determination, a sensitive feeling of what is essential and what ephemeral, and a portion of luck are indispensable. The first lucky accident was a telephone call from a hardware designer enquiring about the possibility of coming to our university to learn about software techniques and acquire a Ph.D. Why not teach him about software and let him teach us about hardware? It didn't take long before the two of us became a functioning team, and Richard Ohran soon became so excited about the new design that he almost totally forgot both software and Ph.D. That didn't disturb me too much, for I was amply occupied with the design of hardware parts; with specifying the micro- and macrocodes, and by programming the latter's interpreter; with planning the overall software system; and in particular with programming a text editor and a diagram editor, both making use of the new high-resolution bit-mapped display and the small miracle called *Mouse* as a pointing device. This exercise in programming highly interactive utility programs required the study and application of techniques quite foreign to conventional compiler and operating system design.

The total project was so diversified and complex that it seemed irresponsible to start it, particularly in view of the small number of part-time assistants available to us, who averaged around seven. The major threat was that it would take too long to keep the enthusiastic two of us persisting and to let the others, who had not yet experienced the power of the workstation idea, become equally enthusiastic. To keep the project within reasonable dimensions, I stuck to three dogmas: Aim for a *single-processor* computer to be operated by a *single user* and programmed in a *single language.* Notably, these cornerstones were diametrically opposed to the trends of the time, which favored research in multiprocessor configurations, time-sharing multiuser operating systems, and as many languages as you could muster.

Under the constraints of a single language, I faced a difficult choice whose effects would be wide ranging, namely, that of selecting a language. Of existing languages, none seemed attractive. Neither could they satisfy all the requirements, nor were they particularly appealing to the compiler designer who knows the task has to be accomplished in a reasonable time span. In particular, the language had to accommodate all our wishes with regard to structuring facilities, based on 10 years' experience with Pascal, and it had to cater to problems so far only handled by coding with an assembler. To cut a long story short, the choice was to design an offspring of both proven Pascal and experimental Modula, that is, *Modula-2.* The module is the key to bringing under one hat the contradictory requirements of high-level abstraction for security through redundancy checking and low-level facilities that allow access to individual features of a particular computer. It lets the programmer encapsulate the use of low-level facilities in a few small parts of the system, thus protecting him from falling into their traps in unexpected places.

The *Lilith* project proved that it is not only possible but advantageous to design a single-language system. Everything from device drivers to text and graphics editors is written in the same language. There is no distinction between modules belonging to the operating system and those belonging to the user's program. In fact, that distinction almost vanishes and with it the burden of a monolithic, bulky resident block of code, which no one wants but everyone has to accept. Moreover, the Lilith project proved the benefits of a well-matched hardware/software design. These benefits can be measured in terms of speed: Comparisons of execution times of Modula programs revealed that Lilith is often superior to a VAX 750 whose complexity and cost are a multiple of those of Lilith. They can also be measured in terms of space: The code of Modula programs for Lilith is shorter than the code for PDP-11, VAX, or 68000 by factors of 2 to 3, and shorter than that of the NS 32000 by a factor of 1.5 to 2. In addition, the code generating parts of compilers for these microprocessors are considerably more intricate than they are in Lilith due to their ill-matched instruction sets. This length factor has to be multiplied by the inferior density factor, which casts a dark shadow over the much advertised high-level language suitability of modern microprocessors and reveals these claims to be exaggerated. The prospect that these designs will be reproduced millions of times is rather depressing, for by their mere number they become our standard building blocks. Unfortunately, advances in semiconductor technology have been so rapid that architectural advances are overshadowed and have become seemingly less relevant. Competition forces manufacturers to freeze new designs into silicon long before they have proved their effectiveness. And whereas bulky software can at least be modified and at best be replaced, nowadays complexity has descended into the very chips. And there is little hope that we have a better mastery of complexity when we apply it to hardware rather than software.

On both sides of this fence, complexity has and will maintain a strong fascination for many people. It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for *elegant* solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are *harder* to find than complex ones, and they require more time, which we too often believe to be unaffordable.

Before closing, let me try to distill some of the common characteristics of the projects that were mentioned. A very important technique that is seldom used as effectively as in computing is the *bootstrap.* We used it in virtually every project. When developing a tool, be it a programming language, a compiler, or a computer, I designed it in such a way that it was beneficial in the very next step: PL360 was developed to implement ALGOL W; Pascal to implement Pascal; Modula-2 to implement the whole workstation software; and Lilith

to provide a suitable environment for all our future work, ranging from programming to circuit documentation and development, from report preparation to font design. Bootstrapping is the most effective way of profiting from one's own efforts as well as suffering from one's mistakes.

This makes it mandatory to *distinguish early between what is essential and what ephemeral.* I have always tried to identify and focus in on what is essential and yields unquestionable benefits. For example, the inclusion of a coherent and consistent scheme of data type declarations in a programming language I consider essential, whereas the details of varieties of for-statements, or whether the compiler distinguishes between upper- and lowercase letters, are ephemeral questions. In computer design, I consider the choice of addressing modes and the provision of complete and consistent sets of (signed and unsigned) arithmetic instructions including proper traps on overflow to be crucial; in contrast, the details of a multichannel prioritized interrupt mechanism are rather peripheral. Even more important is ensuring that the ephemeral never impinge on the systematic, structured design of the central facilities. Rather, the ephemeral must be added fittingly to the existing, well-structured framework.

Rejecting pressures to include all kinds of facilities that "might also be nice to have" is sometimes hard. The danger that one's desire to please will interfere with the goal of consistent design is very real. I have always tried to weigh the gains against the cost. For example, when considering the inclusion of either a language feature or the compiler's special treatment of a reasonably frequent construct, one must weigh the benefits against the added cost of its implementation and its mere presence, which results in a larger system. Language designers often fail in this respect. I gladly admit that certain features of *Ada* that have no counterparts in Modula-2 may be nice to have occasionally, but at the same time, I question whether they are worth the price. The price is considerable: First, although the design of both languages started in 1977, Ada compilers have only now begun to emerge, whereas we have been *using* Modula since 1979. Second, Ada compilers are rumored to be gigantic programs consisting of several hundred thousand lines of code, whereas our newest Modula compiler measures some five thousand lines only. I confess secretly that this Modula compiler is already at the limits of comprehensible complexity, and I would feel utterly incapable of constructing a good compiler for Ada. But even if the effort of building unnecessarily large systems and the cost of memory to contain their code could be ignored, the real cost is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them and use them effectively.

Another common characteristic of the projects sketched was the *choice of tools.* It is my belief that a tool should be commensurate with the product; it must be as simple as possible, but no simpler. A tool is in fact counterproductive when a large part of the entire project is taken up by mastering the tool. Within the Euler, ALGOL W, and PL360 projects, much consideration was given to the development of table-driven, bottom-up syntax analysis techniques. Later, I switched back to the simple recursive-descent, top-down method, which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen. In the development of the Lilith hardware, we restricted ourselves to a good oscilloscope; only rarely was a logic state analyzer needed. This was possible due to a relatively systematic, trick-free concept for the processor.

Every single project was primarily a *learning experiment.* One learns best when inventing. Only by actually *doing* a development project can I gain enough familiarity with the intrinsic difficulties and enough confidence that the inherent details can be mastered. I never could separate the design of a language from its implementation, for a rigid definition without the feedback from the construction of its compiler would seem to me presumptuous and unprofessional. Thus, I participated in the construction of compilers, circuity, and text and graphics editors, and this entailed microprogramming, much high-level programming, circuit design, board layout, and even wire wrapping. This may seem odd, but I simply like hands-on experience much better than team management. I have also learned that researchers accept leadership from a factual, in-touch team member much more readily than from an organization expert, be he a manager in industry or a university professor. I try to keep in mind that teaching by setting a good example is often the most effective method and sometimes the only one available.

Lastly, each of these projects was carried through by the enthusiasm and the desire to succeed in the knowledge that the endeavor was worthwhile. This is perhaps the most essential but also the most elusive and subtle prerequisite. I was lucky to have team members who let themselves be infected with enthusiasm, and here is my chance to thank all of them for their valuable contributions. My sincere thanks go to all who participated, be it in the direct form of working in a team, or in the indirect forms of testing our results and providing feedback, of contributing ideas through criticism or encouragement, or of forming user societies. Without them, neither ALOGL W, nor Pascal, nor Modula-2, nor Lilith would have become what they are. This Turing Award also honors their contributions.

Author's Present Address: Niklaus Wirth, Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.