1980 Turing Award Lecture

The Emperor's Old Clothes

CHARLES ANTONY RICHARD HOARE
Oxford University, England

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland, from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960 to 1968 he held a number of positions with Elliott Brothers, Ltd., England.

Author's present address: Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science Honoris Causa *by the University of Southern California in 1979.*

The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A.M. Turing, an English mathematician who made many important contributions to the computing sciences.

The author recounts his experiences in the implementation, design, and standardization of computer programming languages, and issues a warning for the future.

My first and most pleasant duty in this lecture is to express my profound gratitude to the Association for Computing Machinery for the great honor which they have bestowed on me and for this opportunity to address you on a topic of my choice. What a difficult choice it is! My scientific achievements, so amply recognized by this award, have already been amply described in the scientific literature. Instead of repeating the abstruse technicalities of my trade, I would like to talk informally about myself, my personal experiences, my hopes and fears, my modest successes, and my rather less modest failures. I have learned more from my failures than can ever be revealed in the cold print of a scientific article and now I would like you to learn from them, too. Besides, failures are much more fun to hear about afterwards; they are not so funny at the time.

I start my story in August 1960, when I became a programmer with a small computer manufacturer, a division of Elliott Brothers (London) Ltd., where in the next eight years I was to receive my primary education in computer science. My first task was to implement for the new Elliott 803 computer, a library subroutine for a new fast method of internal sorting just invented by Shell. I greatly enjoyed the challenge of maximizing efficiency in the simple decimal-addressed machine code of those days. My boss and tutor, Pat Shackleton, was very pleased with my completed program. I then said timidly that I thought I had invented a sorting method that would usually run faster than SHELLSORT, without taking much extra store. He bet me sixpence that I had not. Although my method was very difficult to explain, he finally agreed that I had won my bet.

I wrote several other tightly coded library subroutines but after six months I was given a much more important task—that of designing a new advanced high-level programming language for the company's next computer, the Elliott 503, which was to have the same instruction code as the existing 803 but run sixty times faster. In spite of my education in classical languages, this was a task for which I was even

less qualified than those who undertake it today. By great good fortune there came into my hands a copy of the Report on the International Algorithmic Language ALGOL 60. Of course, this language was obviously too complicated for our customers. How could they ever understand all those **begins** and **ends** when even our salesmen couldn't?

Around Easter 1961, a course on ALGOL 60 was offered in Brighton, England, with Peter Naur, Edsger W. Dijkstra, and Peter Landin as tutors. I attended this course with my colleague in the language project, Jill Pym, our divisional Technical Manager, Roger Cook, and our Sales Manager, Paul King. It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

After the ALGOL course in Brighton, Roger Cook was driving me and my colleagues back to London when he suddenly asked, "Instead of designing a new language, why don't we just implement ALGOL 60?" We all instantly agreed—in retrospect, a very lucky decision for me. But we knew we did not have the skill or experience at that time to implement the whole language, so I was commissioned to design a modest subset. In that design I adopted certain basic principles which I believe to be as valid today as they were then.

(1) The first principle was *security*: The principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself. Thus no core dumps should ever be necessary. It was logically impossible for any source language program to cause the computer to run wild, either at compile time or at run time. A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not to—they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980 language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.

(2) The second principle in the design of the implementation was *brevity of the object code produced by the compiler and compactness of run time working data*. There was a clear reason for this: The size of main storage on any computer is limited and its extension involves delay and expense. A program exceeding the limit, even by one word, is impossible to run, especially since many of our customers did not intend to purchase backing stores.

This principle of compactness of object code is even more valid today, when processors are trivially cheap in comparison with the amounts of main store they can address, and backing stores are comparatively even more expensive and slower by many orders of magnitude. If as a result of care taken in implementation the available hardware remains more powerful than may seem necessary for a particular application, the applications programmer can nearly always take advantage of the extra capacity to increase the quality of his program, its simplicity, its ruggedness, and its reliability.

(3) The third principle of our design was that *the entry and exit conventions for procedures and functions should be as compact and efficient as for tightly coded machine-code subroutines.* I reasoned that procedures are one of the most powerful features of a high-level language, in that they both simplify the programming task and shorten the object code. Thus there must be no impediment to their frequent use.

(4) The fourth principle was that *the compiler should use only a single pass*. The compiler was structured as a collection of mutually recursive procedures, each capable of analyzing and translating a major syntactic unit of the language—a statement, an expression, a declaration, and so on. It was designed and documented in ALGOL 60, and then coded into decimal machine code using an explicit stack for recursion. Without the ALGOL 60 concept of recursion, at that time highly controversial, we could not have written this compiler at all.

I can still recommend single-pass top-down recursive descent both as an implementation method and as a design principle for a programming language. First, we certainly want programs to be read by *people* and people prefer to read things once in a single pass. Second, for the user of a time-sharing or personal computer system, the interval between typing in a program (or amendment) and starting to run that program is wholly unproductive. It can be minimized by the high speed of a single pass compiler. Finally, to structure a compiler according to the syntax of its input language makes a great contribution to ensuring its correctness. Unless we have absolute confidence in this, we can never have confidence in the results of any of our programs.

To observe these four principles, I selected a rather small subset of ALGOL 60. As the design and implementation progressed, I gradually

discovered methods of relaxing the restrictions without compromising any of the principles. So in the end we were able to implement nearly the full power of the whole language, including even recursion, although several features were removed and others were restricted.

In the middle of 1963, primarily as a result of the work of Jill Pym and Jeff Hillmore, the first version of our compiler was delivered. After a few months we began to wonder whether anyone was using the language or taking any notice of our occasional reissue, incorporating improved operating methods. Only when a customer had a complaint did he contact us and many of them had no complaints. Our customers have now moved on to more modern computers and more fashionable languages but many have told me of their fond memories of the Elliott ALGOL System and the fondness is not due just to nostalgia, but to the efficiency, reliability, and convenience of that early simple ALGOL System.

As a result of this work on ALGOL, in August 1962, I was invited to serve on the new Working Group 2.1 of IFIP, charged with responsibility for maintenance and development of ALGOL. The group's first main task was to design a subset of the language which would remove some of its less successful features. Even in those days and even with such a simple language, we recognized that a subset could be an improvement on the original. I greatly welcomed the chance of meeting and hearing the wisdom of many of the original language designers. I was astonished and dismayed at the heat and even rancor of their discussions. Apparently the original design of ALGOL 60 had not proceeded in that spirit of dispassionate search for truth which the quality of the language had led me to suppose.

In order to provide relief from the tedious and argumentative task of designing a subset, the working group allocated one afternoon to discussing the features that should be incorporated in the next design of the language. Each member was invited to suggest the improvement he considered most important. On October 11, 1963, my suggestion was to pass on a request of our customers to relax the ALGOL 60 rule of compulsory declaration of variable names and adopt some reasonable default convention such as that of FORTRAN. I was astonished by the polite but firm rejection of this seemingly innocent suggestion: It was pointed out that the redundancy of ALGOL 60 was the best protection against programming and coding errors which could be extremely expensive to detect in a running program and even more expensive not to. The story of the Mariner space rocket to Venus, lost because of the lack of compulsory declarations in FORTRAN, was not to be published until later. I was eventually persuaded of the need to design programming notations so as to maximize the number of errors which cannot be made, or if made, can be reliably detected at compile time. Perhaps this would make the text of programs longer. Never mind! Wouldn't you be delighted if your Fairy Godmother offered to wave her wand

over your program to remove all its errors and only made the condition that you should write out and key in your whole program three times! The way to shorten programs is to use procedures, not to omit vital declarative information.

Among the other proposals for the development of a new ALGOL was that the **switch** declaration of ALGOL 60 should be replaced by a more general feature, namely an array of label-valued variables and that a program should be able to change the values of these variables by assignment. I was very much opposed to this idea, similar to the assigned GO TO of FORTRAN, because I had found a surprising number of tricky problems in the implementation of even the simple labels and switches of ALGOL 60. I could see even more problems in the new feature including that of jumping back into a block after it had been exited. I was also beginning to suspect that programs that used a lot of labels were more difficult to understand and get correct and that programs that assigned new values to label variables would be even more difficult still.

It occurred to me that the appropriate notation to replace the ALGOL 60 switch should be based on that of the conditional expression of ALGOL 60, which selects between two alternative actions according to the value of a Boolean expression. So I suggested the notation for a "case expression" which selects between any number of alternatives according to the value of an integer expression. That was my second language design proposal. I am still most proud of it, because it raises essentially no problems either for the implementor, the programmer, or the reader of a program. Now, after more than fifteen years, there is the prospect of international standardization of a language incorporating this notation—a remarkably *short* interval compared with other branches of engineering.

Back again to my work at Elliott's. After the unexpected success of our ALGOL Compiler, our thoughts turned to a more ambitious project: To provide a range of operating system software for larger configurations of the 503 computer, with card readers, line printers, magnetic tapes, and even a core backing store which was twice as cheap and twice as large as main store, but fifteen times slower. This was to be known as the Elliott 503 Mark II software system.

It comprised:

(1) An assembler for a symbolic assembly language in which all the rest of the software was to be written.

(2) A scheme for automatic administration of code and data overlays, either from magnetic tape or from core backing store. This was to be used by the rest of the software.

(3) A scheme for automatic buffering of all input and output on any available peripheral device, —again, to be used by all the other software.

(4) A filing system on magnetic tape with facilities for editing and job control.

(5) A completely new implementation of ALGOL 60, which removed all the nonstandard restrictions which we had imposed on our first implementation.

(6) A compiler for FORTRAN as it was then.

I wrote documents which described the relevant concepts and facilities and we sent them to existing and prospective customers. Work started with a team of fifteen programmers and the deadline for delivery was set some eighteen months ahead in March 1965. After initiating the design of the Mark II software, I was suddenly promoted to the dizzying rank of Assistant Chief Engineer, responsible for advanced development and design of the company's products, both hardware and software.

Although I was still managerially responsible for the 503 Mark II software, I gave it less attention than the company's new products and almost failed to notice when the deadline for its delivery passed without event. The programmers revised their implementation schedules and a new delivery date was set some three months ahead in June 1965. Needless to say, that day also passed without event. By this time, our customers were getting angry and my managers instructed me to take personal charge of the project. I asked the senior programmers once again to draw up revised schedules, which again showed that the software could be delivered within another three months. I desperately wanted to believe it but I just could not. I disregarded the schedules and began to dig more deeply into the project.

It turned out that we had failed to make any overall plans for the allocation of our most limited resource—main storage. Each programmer expected this to be done automatically, either by the symbolic assembler or by the automatic overlay scheme. Even worse, we had failed to simply count the space used by our own software which was already filling the main store of the computer, leaving no space for our customers to run *their* programs. Hardware address length limitations prohibited adding more main storage.

Clearly, the original specifications of the software could not be met and had to be drastically curtailed. Experienced programmers and even managers were called back from other projects. We decided to concentrate first on delivery of the new compiler for ALGOL 60, which careful calculation showed would take another four months. I impressed upon all the programmers involved that this was no longer just a prediction; it was a promise; if they found they were not meeting their promise, it was their personal responsibility to find ways and means of making good.

The programmers responded magnificently to the challenge. They worked nights and days to ensure completion of all those items of software which were needed by the ALGOL compiler. To our delight,

they met the scheduled delivery date; it was the first major item of working software produced by the company over a period of two years.

Our delight was short-lived; the compiler could not be delivered. Its speed of compilation was only two characters per second which compared unfavorably with the existing version of the compiler operating at about a thousand characters per second. We soon identified the cause of the problem: It was thrashing between the main store and the extension core backing store which was fifteen times slower. It was easy to make some simple improvements, and within a week we had doubled the speed of compilation—to four characters per second. In the next two weeks of investigation and reprogramming, the speed was doubled again—to eight characters per second. We could see ways in which within a month this could be still further improved; but the amount of reprogramming required was increasing and its effectiveness was decreasing; there was an awful long way to go. The alternative of increasing the size of the main store so frequently adopted in later failures of this kind was prohibited by hardware addressing limitations.

There was no escape: The entire Elliott 503 Mark II software project had to be abandoned, and with it, over thirty man-years of programming effort, equivalent to nearly one man's active working life, and I was responsible, both as designer and as manager, for wasting it.

A meeting of all our 503 customers was called and Roger Cook, who was then manager of the computing division, explained to them that not a single word of the long-promised software would ever be delivered to them. He adopted a very quiet tone of delivery, which ensured that none of the customers could interrupt, murmur in the background, or even shuffle in their seats. I admired but could not share his calm. Over lunch our customers were kind to try to comfort me. They had realized long ago that software to the original specification could never have been delivered, and even if it had been, they would not have known how to use its sophisticated features, and anyway many such large projects get cancelled before delivery. In retrospect, I believe our customers were fortunate that hardware limitations had protected them from the arbitrary excesses of our software designs. In the present day, users of microprocessors benefit from a similar protection—but not for much longer.

At that time I was reading the early documents describing the concepts and features of the newly announced OS 360, and of a new time-sharing project called Multics. These were far more comprehensive, elaborate, and sophisticated than anything I had imagined, even in my first version of the 503 Mark II software. Clearly IBM and MIT must be possessed of some secret of successful software design and implementation whose nature I could not even begin to guess at. It was only later that they realized they could not either.

So I still could not see how I had brought such a great misfortune upon my company. At the time I was convinced that my managers were planning to dismiss me. But no, they were intending a far more severe punishment. "O.K. Tony," they said. "You got us into this mess and now you're going to get us out." "But I don't know how," I protested, but their reply was simple. "Well then, you'll have to find out." They even expressed confidence that I could do so. I did not share their confidence. I was tempted to resign. It was the luckiest of all my lucky escapes that I did not.

Of course, the company did everything they could to help me. They took away my responsibility for hardware design and reduced the size of my programming teams. Each of my managers explained carefully his own theory of what had gone wrong and all the theories were different. At last, there breezed into my office the most senior manager of all, a general manager of our parent company, Andrew St. Johnston. I was surprised that he had even heard of me. "You know what went wrong?" he shouted—he always shouted — "You let your programmers do things which you yourself do not understand." I stared in astonishment. He was obviously out of touch with present-day realities. How could one person ever understand the whole of a modern software product like the Elliott 503 Mark II software system?

I realized later that he was absolutely right; he had diagnosed the true cause of the problem and he had planted the seed of its later solution.

I still had a team of some forty programmers and we needed to retain the good will of customers for our new machine and even regain the confidence of the customers for our old one. But what should we actually plan to do when we knew only one thing—that all our previous plans had failed? I therefore called an all-day meeting of our senior programmers on October 22, 1965, to thrash out the question between us. I still have the notes of that meeting. We first listed the recent major grievances of our customers: Cancellation of products, failure to meet deadlines, excessive size of software, "... not justified by the usefulness of the facilities provided," excessively slow programs, failure to take account of customer feedback; "Earlier attention paid to quite minor requests of our customers might have paid as great dividends of goodwill as the success of our most ambitious plans."

We then listed our own grievances: Lack of machine time for program testing, unpredictability of machine time, lack of suitable peripheral equipment, unreliability of the hardware even when available, dispersion of programming staff, lack of equipment for keypunching of programs, lack of firm hardware delivery dates, lack of technical writing effort for documentation, lack of software knowledge outside of the programming group, interference from higher managers who imposed decisions, "... without a full realization of

the more intricate implications of the matter," and overoptimism in the face of pressure from customers and the Sales Department.

But we did not seek to excuse our failure by these grievances. For example, we admitted that it was the duty of programmers to educate their managers and other departments of the company by "... presenting the necessary information in a simple palatable form." The hope "... that deficiencies in original program specifications could be made up by the skill of a technical writing department ... was misguided; the design of a program and the design of its specification must be undertaken in parallel by the same person, and they must interact with each other. A lack of clarity in specification is one of the surest signs of a deficiency in the program it describes, and the two faults must be removed simultaneously before the project is embarked upon." I wish I had followed this advice in 1963; I wish we all would follow it today.

My notes of the proceedings of that day in October 1965 include a complete section devoted to failings within the software group; this section rivals the most abject self-abasement of a revisionist official in the Chinese cultural revolution. Our main failure was overambition. "The goals which we have attempted have obviously proved to be far beyond our grasp." There was also failure in prediction, in estimation of program size and speed, of effort required, in planning the coordination and interaction of programs, in providing an early warning that things were going wrong. There were faults in our control of program changes, documentation, liaison with other departments, with our management, and with our customers. We failed in giving clear and stable definitions of the responsibilities of individual programmers and project leaders, —Oh, need I go on? What was amazing was that a large team of highly intelligent programmers could labor so hard and so long on such an unpromising project. You know, you shouldn't trust us intelligent programmers. We can think up such good arguments for convincing ourselves and each other of the utterly absurd. Especially don't believe us when we promise to repeat an earlier success, only bigger and better next time.

The last section of our inquiry into the failure dealt with the criteria of quality of software. "In the recent struggle to deliver any software at all, the first casualty has been consideration of the quality of the software delivered. The quality of software is measured by a number of totally incompatible criteria, which must be carefully balanced in the design and implementation of every program." We then made a list of no less than seventeen criteria which has been published in a guest editorial in Volume 2 of the journal, *Software Practice and Experience.*

How did we recover from the catastrophe? First, we classified our 503 customers into groups, according to the nature and size of the hardware configurations which they had bought—for example, those

with magnetic tapes were all in one group. We assigned to each group of customers a small team of programmers and told the team leader to visit the customers to find out what they wanted; to select the easiest request to fulfill, and to make plans (but not promises) to implement it. In no case would we consider a request for a feature that would take more than three months to implement and deliver. The project leader would then have to convince *me* that the customers' request was reasonable, that the design of the new feature was appropriate, and that the plans and schedules for implementation were realistic. Above all, I did not allow anything to be done which I did not myself understand. It worked! The software requested began to be delivered on the promised dates. With an increase in our confidence and that of our customers, we were able to undertake fulfilling slightly more ambitious requests. Within a year we had recovered from the disaster. Within two years, we even had some moderately satisfied customers.

Thus we muddled through by common sense and compromise to something approaching success. But I was not satisfied. I did not see why the design and implementation of an operating system should be so much more difficult than that of a compiler. This is the reason why I have devoted my later research to problems of parallel programming and language constructs which would assist in clear structuring of operating systems—constructs such as monitors and communicating processes.

While I was working at Elliott's, I became very interested in techniques for formal definition of programming languages. At that time, Peter Landin and Christopher Strachey proposed to define a programming language in a simple functional notation, that specified the effect of each command on a mathematically defined abstract machine. I was not happy with this proposal because I felt that such a definition must incorporate a number of fairly arbitrary representation decisions and would not be much simpler in principle than an implementation of the language for a real machine. As an alternative, I proposed that a programming language definition should be formalized as a set of axioms, describing the desired properties of programs written in the language. I felt that carefully formulated axioms would leave an implementation the necessary freedom to implement the language efficiently on different machines and enable the programmer to prove the correctness of his programs. But I did not see how to actually do it. I thought that it would need lengthy research to develop and apply the necessary techniques and that a university would be a better place to conduct such research than industry. So I applied for a chair in Computer Science at the Queen's University of Belfast where I was to spend nine happy and productive years. In October 1968, as I unpacked my papers in my new home in Belfast, I came across an obscure preprint of an article by Bob Floyd entitled, "Assigning Meanings to Programs." What a stroke of luck! At last I could see a

way to achieve my hopes for my research. Thus I wrote my first paper on the axiomatic approach to computer programming, published in the *Communications of the ACM* in October 1969.

Just recently, I have discovered that an early advocate of the assertional method of program proving was none other than Alan Turing himself. On June 24, 1950, at a conference in Cambridge, he gave a short talk entitled, "Checking a Large Routine," which explains the idea with great clarity. "How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite *assertions* which can be checked individually, and from which the correctness of the whole program easily follows."

Consider the analogy of checking an addition. If the sum is given [just as a column of figures with the answer below] one must check the whole at one sitting. But if the totals for the various columns are given [with the carries added in separately], the checker's work is much easier, being split up into the checking of the various assertions [that each column is correctly added] and the small addition [of the carries to the total]. This principle can be applied to the checking of a large routine but we will illustrate the method by means of a small routine viz. one to obtain n factorial without the use of a multiplier. Unfortunately there is no coding system sufficiently generally known to justify giving this routine in full, but a flow diagram will be sufficient for illustration. That brings me back to the main theme of my talk, the design of programming languages.

During the period August 1962 to October 1966, I attended every meeting of the IFIP ALGOL working group. After completing our labors on the IFIP ALGOL subset, we started on the design of ALGOL X, the intended successor to ALGOL 60. More suggestions for new features were made and in May 1965, Niklaus Wirth was commissioned to collate them into a single language design. I was delighted by his draft design which avoided all the known defects of ALGOL 60 and included several new features, all of which could be simply and efficiently implemented, and safely and conveniently used.

The description of the language was not yet complete. I worked hard on making suggestions for its improvement and so did many other members of our group. By the time of the next meeting in St. Pierre de Chartreuse, France, in October 1965, we had a draft of an excellent and realistic language design which was published in June 1966 as "A Contribution to the Development of ALGOL" in the *Communications of the ACM*. It was implemented on the IBM 360 and given the title ALGOL W by its many happy users. It was not only a worthy successor of ALGOL 60, it was even a worthy predecessor of PASCAL.

At the same meeting, the ALGOL committee had placed before it, a short, incomplete, and rather incomprehensible document, describing a different, more ambitious and, to me, a far less attractive language.

I was astonished when the working group, consisting of all the best known international experts of programming languages, resolved to lay aside the commissioned draft on which we had all been working and swallow a line with such an unattractive bait.

This happened just one week after our inquest on the 503 Mark II software project. I gave desperate warnings against the obscurity, the complexity, and overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.

The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature. It also requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late.

So it was with the ALGOL committee. Clearly the draft which it preferred was not yet perfect. So a new and final draft of the new ALGOL language design was promised in three months' time; it was to be submitted to the scrutiny of a subgroup of four members including myself. Three months came and went, without a word of the new draft. After six months, the subgroup met in the Netherlands. We had before us a longer and thicker document, full of errors corrected at the last minute, describing yet another but to me, equally unattractive language. Niklaus Wirth and I spent some time trying to get removed some of the deficiencies in the design and in the description, but in vain. The completed final draft of the language was promised for the next meeting of the full ALGOL committee in three months time.

Three months came and went—not a word of the new draft appeared. After six months, in October 1966, the ALGOL working group met in Warsaw. It had before it an even longer and thicker document, full of errors corrected at the last minute, describing equally obscurely yet another different, and to me, equally unattractive language. The experts in the group could not see the defects of the design and they firmly resolved to adopt the draft, believing it would be completed in three months. In vain, I told them it would not. In vain, I urged them to remove some of the technical mistakes of the language, the predominance of references, the default type conversions. Far from wishing to simplify the language, the working group actually asked the authors to include even more complex features like overloading of operators and concurrency.

When any new language design project is nearing completion, there is always a mad rush to get new features added before standardization. The rush is mad indeed, because it leads into a trap from which there

is no escape. A feature which is omitted can always be added later, when its design and its implications are well understood. A feature which is included before it is fully understood can never be removed later.

At last, in December 1968, in a mood of black depression, I attended the meeting in Munich at which our long-gestated monster was to come to birth and receive the name ALGOL 68. By this time, a number of other members of the group had become disillusioned, but too late: the committee was now packed with supporters of the language, which was sent up for promulgation by the higher committees of IFIP. The best we could do was to send with it a minority report, stating our considered view that, "... as a tool for the *reliable creation* of sophisticated programs, the language was a failure." This report was later suppressed by IFIP, an act which reminds me of the lines of Hilaire Belloc,

> But scientists, who ought to know / Assure us that it must be so./
> Oh, Let us never, never doubt / What nobody is sure about.

I did not attend any further meetings of that working group. I am pleased to report that the group soon came to realize that there was something wrong with their language and with its description; they labored hard for six more years to produce a revised description of the language. It is a great improvement but I'm afraid, that in my view, it does not remove the basic technical flaws in the design, nor does it begin to address the problem of its overwhelming complexity.

Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of the large number of conflicting objectives for each of our programming projects. If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.

Now let me tell you about yet another overambitious language project. Between 1965 and 1970 I was a member and even chairman of the Technical Committee No. 10 of the European Computer Manufacturers Association. We were charged first with a watching brief and then with the standardization of a language to end all languages, designed to meet the needs of all computer applications, both commercial and scientific, by the greatest computer manufacturer of all time. I had studied with interest and amazement, even a touch of amusement, the four initial documents describing a language called NPL, which appeared between March 1 and November 30, 1964. Each was more ambitious and absurd than the last in its wishful speculations. Then the language began to be implemented and a new series of documents began to appear at six-monthly intervals, each describing the final frozen version of the language, under its final frozen name PL/I.

But to me, each revision of the document simply showed how far the initial *F*-level implementation had progressed. Those parts of the language that were not yet implemented were still described in free-flowing flowery prose giving promise of unalloyed delight. In the parts that *had* been implemented, the flowers had withered; they were choked by an undergrowth of explanatory footnotes, placing arbitrary and unpleasant restrictions on the use of each feature and loading upon a programmer the responsibility for controlling the complex and unexpected side-effects and interaction effects with all the other features of the language.

At last, March 11, 1968, the language description was nobly presented to the waiting world as a worthy candidate for standardization. But it was not. It had already undergone some seven thousand corrections and modifications at the hand of its original designers. Another twelve editions were needed before it was finally published as a standard in 1976. I fear that this was not because everybody concerned was satisfied with its design, but because they were thoroughly bored and disillusioned.

For as long as I was involved in this project, I urged that the language be simplified, if necessary by subsetting, so that the professional programmer would be able to understand it and able to take responsibility for the correctness and cost-effectiveness of his programs. I urged that the dangerous features such as defaults and **ON**- conditions be removed. I knew that it would be impossible to write a wholly reliable compiler for a language of this complexity and impossible to write a wholly reliable program when the correctness of each part of the program depends on checking that every other part of the program has avoided all the traps and pitfalls of the language.

At first I hoped that such a technically unsound project would collapse but I soon realized it was doomed to success. Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way — and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.

All this happened a long time ago. Can it be regarded as relevant in a conference dedicated to a preview of the Computer Age that lies ahead? It is my gravest fear that it can. The mistakes which we have made in the last twenty years are being repeated today on an even grander scale. I refer to a language design project which has generated documents entitled *strawman, woodenmen, tinman, ironman, steelman, green* and finally now ADA. This project has been initiated and sponsored by one of the world's most powerful organizations, the United States Department of Defense. Thus it is ensured of an influence and attention quite independent of its technical merits and its faults

and deficiencies threaten us with far greater dangers. For none of the evidence we have so far can inspire confidence that this language has avoided any of the problems that have afflicted other complex language projects of the past.

I have been giving the best of my advice to this project since 1975. At first I was extremely hopeful. The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity. Gradually these objectives have been sacrificed in favor of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns of safety and economy.

It is not too late! I believe that by careful pruning of the ADA language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use. The sponsors of the language have declared unequivocally, however, that there shall be no subsets. This is the strangest paradox of the whole strange project. If you want a language with no subsets, you must make it *small*.

You include only those features which you know to be needed for *every* single application of the language and which you know to be appropriate for *every* single hardware configuration on which the language is implemented. Then extensions can be specially designed where necessary for particular hardware devices and for particular applications. That is the great strength of PASCAL, that there are so few unnecessary features and almost no need for subsets. That is why the language is strong enough to support specialized extensions—Concurrent PASCAL for real time work, PASCAL PLUS for discrete event simulation, UCSD PASCAL for microprocessor work stations. If only we could learn the right lessons from the successes of the past, we would not need to learn from our failures.

And so, the best of my advice to the originators and designers of ADA has been ignored. In this last resort, I appeal to you, representatives of the programming profession in the United States, and citizens concerned with the welfare and safety of your own country and of mankind: Do not allow this language in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, antiballistic missile defense systems. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce the risk, not to increase it.

Let me not end on this somber note. To have our best advice ignored is the common fate of all who take on the role of consultant, ever since Cassandra pointed out the dangers of bringing a wooden horse within the walls of Troy. That reminds me of a story I used to hear in my childhood. As far as I recall, its title was:

The Emperor's Old Clothes

Many years ago, there was an Emperor who was so excessively fond of clothes that he spent all his money on dress. He did not trouble himself with soldiers, attend banquets, or give judgement in court. Of any other king or emperor one might say, "He is sitting in council," but it was always said of him, "The emperor is sitting in his wardrobe." And so he was. On one unfortunate occasion, he had been tricked into going forth naked to his chagrin and the glee of his subjects. He resolved never to leave his throne, and to avoid nakedness, he ordered that each of his many new suits of clothes should be simply draped on top of the old.

Time passed away merrily in the large town that was his capital. Ministers and courtiers, weavers and tailors, visitors and subjects, seamstresses and embroiderers, went in and out of the throne room about their various tasks, and they all exclaimed, "How magnificent is the attire of our Emperor."

One day the Emperor's oldest and most faithful Minister heard tell of a most distinguished tailor who taught at an ancient institute of higher stitchcraft, and who had developed a new art of abstract embroidery using stitches so refined that no one could tell whether they were actually there at all. "These must indeed be splendid stitches," thought the minister. "If we can but engage this tailor to advise us, we will bring the adornment of our Emperor to such heights of ostentation that all the world will acknowledge him as the greatest Emperor there has ever been."

So the honest old Minister engaged the master tailor at vast expense. The tailor was brought to the throne room where he made obeisance to the heap of fine clothes which now completely covered the throne. All the courtiers waited eagerly for his advice. Imagine their astonishment when his advice was not to add sophistication and more intricate embroidery to that which already existed, but rather to remove layers of the finery, and strive for simplicity and elegance in place of extravagant elaboration. "This tailor is not the expert that he claims," they muttered. "His wits have been addled by long contemplation in his ivory tower and he no longer understands the sartorial needs of a modern Emperor." The tailor argued loud and long for the good sense of his advice but could not make himself heard. Finally, he accepted his fee and returned to his ivory tower.

Never to this very day has the full truth of this story been told: That one fine morning, when the Emperor felt hot and bored, he extricated himself carefully from under his mountain of clothes and

is now living happily as a swineherd in another story. The tailor is canonized as the patron saint of all consultants, because in spite of the enormous fees that he extracted, he was never able to convince his clients of his dawning realization that their clothes have no Emperor.

References

1. Arvind, and Gostelow, K. P. A new interpreter for data flow schemas and its implications for computer architecture. Tech. Rep. No. 72, Dept. Comptr. Sci., U. of California, Irvine, Oct. 1975.
2. Backus, J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71–86.
3. Berkling, K. J. Reduction languages for reduction machines. Interner Bericht ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, Sept. 1976.
4. Burge, W. H. *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass, 1975.
5. Church, A. *The Calculi of Lambda-Conversion.* Princeton U. Press, Princeton, N.J., 1941.
6. Curry, H. B., and Feys, R. *Combinatory Logic, Vol. 1.* North Holland Pub. Co., Amsterdam, 1958.
7. Dennis, J. B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., May 1973.
8. Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
9. Friedman, D. P., and Wise, D. S. CONS should not evaluate its arguments. In *Automata, Languages and Programming,* S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257–284.
10. Henderson, P., and Morris, J. H., Jr. A lazy evaluator. Conf. Record Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, pp. 95–103.
11. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (Oct. 1969), 576–583.
12. Iverson, K. *A Programming Language.* Wiley, New York, 1962.
13. Kosinski, P. A data flow programming language. Rep. RC 4264, IBM T. J. Watson Research Ctr., Yorktown Heights, N.Y., March 1973.
14. Landin, P. J. The mechanical evaluation of expressions. *Computer J. 6,* 4 (1964), 308–320.
15. Mago, G. A. A network of microprocessors to execute reduction languages. *Int. J. Comptr. and Inform. Sci.*
16. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM 16,* 8 (Aug. 1973), 491–502.
17. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Pt. 1. *Comm. ACM 3,* 4 (April 1960), 184–195.
18. McJones, P. A Church-Rosser property of closed applicative languages. Rep. RJ 1589, IBM Res. Lab., San Jose, Calif., May 1975.
19. Reynolds, J. C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM 13,* 5 (May 1970), 308–318.

20. Reynolds, J. C. Notes on a lattice-theoretic approach to the theory of computation. Dept. Syst. and Inform. Sci., Syracuse U., Syracuse, N.Y., 1972.
21. Scott, D. Outline of a mathematical theory of computation. Proc. 4th Princeton Conf. on Inform. Sci. and Syst., 1970.
22. Scott, D. Lattice-theoretic models for various type-free calculi. Proc. Fourth Int. Congress for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972.
23. Scott, D., and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symp. on Comptrs. and Automata, Polytechnic Inst. of Brooklyn, 1971.

Categories and Subject Descriptors:

D.3.2 [**Programming Languages**]: Language Classifications; D.3.4 [**Programming Languages**]: Processors—*compilers*; D.4.1 [**Operating Systems**]: Process Management

General Terms:

Design, Languages, Reliability, Security

Additional Key Words and Phrases:

Ada, Algol 60, Algol 68, Algol W, Elliott 503 Mark II Software System, PL/I