Infotech State-of-the-Art Report on High-level Languages 1972

C A R HOARE has been Professor of Computer Science at the Queen's University Belfast since 1968. Before this, he worked on general purpose software construction with Elliott Automation, where he became Chief Scientist in the Computing Research Laboratory. He was the designer of the first successful commercially available compiler for ALGOL 60, which is still in use on the Elliott 803 and 503 computers, and has participated in the IFIP working group for ALGOL (2.1). He is noted for his contribution to the development of the language. His current research interests are concerned with the development of techniques for controlling and reducing the phenomenon of programming error and in the discovery of practical and useful techniques for the implementation of operating systems.

PROSPECTS FOR A BETTER PROGRAMMING LANGUAGE

In this presentation, I shall address myself to the question, can one hope for a better programming language. I shall attempt to show that there are indeed good grounds for hope of advance in this field, that there are even now certain well established results in computing science of both a theoretical and practical nature that, if taken into account in the design and development of a programming language, would result in certain benefits to the user of that language. However, I shall also argue that at the present time there is very little demand for a better programming language and it is likely, even in our supposedly fast-moving field, that known improvements in the design of programming languages will take many years, or perhaps decades, before they achieve widespread recognition.

Before embarking on our search for a better programming language, I shall take stock of our present position. The most widely used high level programming language is undoubtedly COBOL. This is a language designed in 1960, before the IBM 1401 came into vogue. The 1401 is now obsolescent, perhaps even obsolete, but COBOL maintains its dominant position on the machines that superseded it. Another very popular language is FORTRAN, which dates from about 1956 and came into widespread use at about the same time as the IBM 704. Today there is a new generation of computer men who have hardly heard of the 704 but they have been taught FORTRAN and perhaps use no other language. Can it be that our rate of progress in the design of software tools does not match the rate of development of hardware? As our component technology proceeds gradually through the generations, first, second, third, and perhaps just round the corner, a fourth, our programming languages remain fixed in the era of thermionic valves and Williams tubes. If we are to make any progress in this field, we must first obtain a thorough understanding of the reason for this amazing longevity of old software designs. Why is it that third generation hardware components are universally welcomed as an improvement in both reliability and speed whereas the epithet third generation when applied to software has come to mean "neither very reliable nor very fast"? Having posed this question, I fear that I shall not be able to answer it in general terms; however, I shall try to answer it in terms of our search for a better programming language.

MANUFACTURER LANGUAGES

The first and most obvious place to start our search is with the computer manufacturers. Let us give pride of place to the largest of them. There is little doubt that IBM regards the problem of a better programming language as at least to some extent solved; all that is required is a decision to switch programming to PL/1. It is no part of my task here to denigrate the commercially saleable products of any company, but I must declare that I do not regard PL/1 as the end of our search. I cannot explain at this point my reasons for rejecting PL/1, although I hope that they will become apparent during the remainder of my presentation. However, I would like to comment on the rather curious manner in which the language was designed.

The original concept in the middle of the last decade was that a committee of users of existing languages, mainly FORTRAN and COBOL, could within a few months lay down the outline of a new and superior language for a new range of computers. It was considered that for this purpose there was no requirement for knowledge or experience of any of the work that had been progressing in the field over the previous 5 years. There was no need for any knowledge of the theory of programming or of the practice of implementing programming languages. Obviously, the design of a programming language was such a simple task that there was no need to call in the experts: a committee of informed laymen was quite good enough and could complete the work quite quickly.

The error of this belief was soon recognized and control of the design passed to the computer manufacturer. However, it was obviously still a very easy task to design a programming language;

the experts would soon have it completed and the final frozen version of the language would be available to meet a three-month deadline. After that, there would be no need to consider any proposal for further change. However, at the end of three months, something was still wrong and the deadline for a frozen version had to be extended yet again. I believe that this process of repeated freezing and unfreezing lasted several years, until the first cutomer version of the language was delivered. However, even then the rate of change did not slacken. The need for compatibility merely made the changes more arbitrary and complicated. In 1967, a version of the language was submitted for standardization. Even this had little effect on the rate of change; it just increased the number of committees engaged in making the changes. This process has been continuing for six years and it would be a rash man who supposed it was likely to stop now. This is why PL/1 has been called "7000 after-thoughts in search of a language".

Of course, a long development period does not argue against the quality of a language, and certainly the current versions of PL/1 are a great improvement on the designs published in 1965. Nevertheless, there are certain features of PL/1 that are generally agreed to owe their place more to historical accident than to technical merit. Furthermore, one must conclude that the design of a language is not a task that should be undertaken against a three-month deadline, except by people who have a considerably greater degree of theoretical knowledge and practical experience than were available to the designers of PL/1 during the early years of its development.

Turning to other computer manufacturers, it is not possible to find any more promising development. The main concern of the larger manufacturers is how they are going to catch up with the five years' start and multimillion dollar investment that IBM has made in PL/1. No other manufacturer can give thought to any other language until they have at least made PL/1 available to their customers. I guess that it will be five years before PL/1 is in general use by non-IBM customers; the only ground for hope is that when other manufacturers have caught up, IBM will produce another and possibly better language to retain its competitive advantage.

So, unless PL/1 is regarded as the end of our search for a better language, there is little hope of finding our objective with a computer manufacturer. This is by no means an attack on the computer manufacturing industry. Their task is to provide what the customer wants, or what the customer thinks he wants, or perhaps, best of all, what he has been persuaded to think he wants. And I regret to say that few customers at the present time show any inclination to want a better programming language.

ALGOL 68

Let us turn our attention to the work of the experts. I refer, of course, to a working group set up under the auspices of the International Federation of Information Processing Society, and popularly known as the ALGOL Committee. This is a committee of scientists of acknowledged distinction, commissioned with the task of developing programming languages in the tradition of ALGOL 60. In 1965, they decided to start on the design of a successor to ALGOL and charged one of their members to produce a draft for the next meeting in six months' time. At the next meeting, they charged a sub-committee of four members to produce a new draft, this time in three months' time. The next meeting was held a year later, in 1966, but the draft was not yet complete; another few months would be required. Six months later, a further delay was reported. Finally, in late December 1968, a final report was produced. Since then, I have stopped counting how many changes have been made to it.

GOOD LANGUAGES

The next place to turn in our search is the universities. Here, at first sight, the picture that greets us is quite different. Instead of universal recognition of a single language as a goal towards which all must strive, we see each individual scientist setting up an entirely new language of his own design that he propounds, and even implements, as the solution to some or all programming problems. However, I regret that on closer inspection the situation is depressingly similar to that found in the design of PL/1. The academic designer of a programming language rarely gives himself the time to think deeply about the theoretical basis of his design, or about the practical implications of his decisions. Thus, if and when the language is implemented, it rarely finds a sufficiently wide circle of users to reveal even its most blatant defects. Thus we have a long series of languages whose names pass rapidly into oblivion or are rapidly superseded by their inventor's second, third, and even subsequent thoughts. Who now remembers IPL 1 to 4, SNOBOLS 1 and 11, CPL, MAD, COGENT, LISP 2, and many others that I cannot recall? However, there are a few honourable exceptions: languages implemented locally at a university that by their sheer quality, without the support of the US Department of Defence and without the commercial backing of any large computer manufacturer, have managed by their merit alone to propogate themselves beyond the circle of their immediate invention. It is well worthwhile investigating the reasons behind the success of each such language. It is my belief that in each case the explanation may be found in the fact that the designer knows both the theoretical basis of what he is doing and the practical engineering aspects of its implementation. The first example of such a language I shall discuss is ALGOL 60. This language still enjoys a certain currency among those who prefer better programming languages. The main reason for its excellence is that it introduced the idea of compound statements, local declarations, conditionals, and the while form of the for statement. These program structuring methods extend to the complete program the benefits of the bracketing of arithmetic expressions, which was the major advance made by FORTRAN over machine code. It is interesting to note that these are exactly the features now attracting users of FORTRAN and COBOL to PL/1.

One important symptom of the excellence of a programming language is its stability. After a very few minor changes had been made to ALGOL 60 in 1962 and had culminated in the revised ALGOL report, and a few minor restrictions had been made by implementors and standardizing bodies, there has been no need for further change to this language. In fact, a few years ago, Donald Knuth wrote an article on *The remaining trouble spots of ALGOL 60*, CACM Oct 1967 vol 10 no 10 pp 611 to 623. These trouble spots were so trivial that nobody had the slightest worry about them any longer. A paper on the remaining trouble spots in PL/1 might run to rather more than the seven volumes of Knuth's *Art of programming*.

The trouble with ALGOL 60 is that although its basic structure was a fantastic advance on its predecessors, it has one or two small features that were inappropriately designed and that implementors tend to regard as an excuse for low quality implementation. I refer to the excessively general definition of the for statement, the absence of compulsory specification for parameters, the adoption of the name parameter as default, and the unfortunate comment convention after the basic word end. Most of these faults are cured in later languages but the lesson we should draw from them is that a language design in future must not only be correct in its general structure but also every detail must be carefully thought out in light of its effect on the user, its effect on the implementation, and its interaction effects on other details. A good programming language of the future should be one in which a programmer may safely be encouraged to use *every* feature and does not, after learning the language, have to be taught to avoid certain unexpectedly expensive constructions. Furthermore, the details of the syntax of the language should be such that there is no need for the list of common errors that appears in most of the existing programming manuals.

My next example of a language that has preserved and propogated itself by its merit alone is LISP 1.5 (39). This language was soundly based on previously existing mathematical theory, the theory of generalized arithmetic and recursive functions; it is a prime example of a sound theoretical foundation. Also, it is based on a brilliant implementation technique: the use of pointers together with a scan, mark, and collect system of garbage collection. It was McCarthy's genius that recognized that the theory and the practical technique formed a perfect marriage. This language immediately outstripped all its competitors and is still the only language of its kind generally recognized and known throughout the world; it is still widely used in its own application area. Compare this with the fate of LISP's intended successor, LISP 2, which was based on no particular insight, either of a theoretical or of a practical nature.

ALGOL 60 and LISP were so far ahead of their time that for a period of about five years they were in themselves significantly better than nearly all proposals for their purported extension or improvement.

The next language I wish to consider was developed in 1965, published in 1966, implemented shortly after that, and has been given the honorific title ALGOL W. This language was designed by the IFIP ALGOL committee, written up by myself and Professor Wirth, and implemented by him and his colleagues at Stanford University. (18) The language is now in use in universities in many parts of the world. The reason for its success is simple. The language consolidated the best features of ALGOL 60 while eliminating its trouble spots. Each new feature was designed with careful consideration both of theoretical aspects and of practical implementation. Of course, this language is now five years old and it did not give final solutions in all areas; it has been found rather weak in input and output. However, it has achieved the distinction of propagation by its merit alone, without the support of the international committee that sponsored it, and even without the active promulgation of its authors. The reason why its users are so enthusiastic are:

1 It has a speed of compilation comparable with that of the WATFOR in-core FORTRAN compiler.
2 It has excellence of diagnostics comparable to WATFOR.
3 It has speed of execution comparable to that of FORTRAN G on the IBM System/360.
4 Its reliability of implementation is outstanding in comparison with commercial products.
5 It has a range of applications significantly wider than FORTRAN.
6 It is basically a simple, small, and regular language, easy to teach and learn, and easy to use effectively.

The fact that these properties have not led to even wider use of the language is explicable only by the fact that, as I have already said, there is very little demand in the world for better programming languages.

A recent thorough investigation of ALGOL W carried out by Brian Wichmann has revealed that it is, according to objective measurement, the best ALGOL ever produced, in spite of the wide range of extensions. This contradicts the widespread belief that more powerful and generalized languages must be less efficient. See page 291.

A language that is almost in the category of languages that propagate themselves by merit alone is Ken Iverson's APL. The theoretical foundation of this language was laid in 1960 and formed the material of an interesting book. (21). The language was first implemented a few years ago in a research laboratory and the quality of sound engineering thought that went into the design of the implementation is very impressive indeed: it extended even to the design of the character set in which the language was encoded. The language then propagated by its merit within IBM and I believe that it was only with some reluctance that it was released to customers. It now has a growing band of enthusiastic users. While I do not myself regard APL as a general purpose programming language of universal application, I can well understand the enthusiasm of those whose problems are of a nature and size to be tackled by APL.

Of course, this survey of self-propagating languages cannot be complete. However, there are two other languages I would like to mention, because they seem to point clearly the direction for future advance. These are SIMULA, developed in Oslo by Dahl and Nygaard, and PL/360, developed at Stanford, again by Professor Wirth. These languages represent two ends of a spectrum, representing the placing of different emphasis on different objectives. SIMULA was designed as a high level language for complex simulation problems, with quite elaborate implementation mechanisms. PL/360 emphasized efficiency of translation and execution, even at the expense of machine dependence. Both languages have gained currency outside their immediate circle of inventors; both have been designed by scientists of inherent brilliance and long practical experience of both implementation and use of programming languages. (48,52,53,54)

PROPERTIES OF A BETTER PROGRAMMING LANGUAGE

This concludes my brief survey of the current scene. In order to explain the events of the past and to make likely predictions about the future, it is now necessary to enquire more deeply into what constitutes a good programming language. You might expect that there are as many points of view on this matter as there are programming languages, if not as many as there are programmers. In fact, among those who are experienced in using a wide variety of programming languages there is growing a reasonable consensus about what are the desiderata of programming language design and what benefits a programmer can reasonably expect to achieve, or *not to achieve*, by the use of a good programming language.

It is essential to have a clear recognition of what a programming language *cannot* be expected to do for a programmer. In common with a few others, I take an extremely modest view of what a programming language can accomplish; I believe it cannot help the programmer in any of his really difficult tasks, such as the investigations and decisions about what needs to be done, what can be done, and how to do it. A programming language cannot tell a programmer how to split up the work among his team or how long the project will take. It cannot tell him the most effective algorithms or even how efficient his chosen algorithm will be. It cannot prevent him from making mistakes in the understanding of his problem, mistakes in designing his program, and mistakes in coding it. It cannot tell him how to organize his program testing to give him the best chance of achieving a reliable product. It cannot force him to construct the documentation needed by those who subsequently want to adapt or improve the program. In summary, a programming language cannot solve any of the really deep and difficult problems that face a programmer in the practice of his art.

My conclusion from this is not that programming languages are useless but that the programming language designer must start with an attitude of appropriate modesty. The results of his work are going to play only a relatively small part in the overall process of designing and implementing programs; therefore the best service that he can perform is to avoid actual hindrance to the programmer in the pursuit of his objectives. The programmer has enough trouble without having to struggle with his programming language as well. A good programming language should be unobtrusive, so that the programmer hardly notices that he has a language standing between him and his machine. When the language is taught to beginners, the impression should be that they are learning programming rather than merely the idiosyncrasies of some programming language.

A number of interesting conclusions can be drawn from this unusually modest approach to the design of a programming language:

1 The translator should not interpose any significant delay or complexity between the submission of a program and its execution. I would put the target speed of translation as the same as that of a conventional linkage editor. This would have the additional advanatage that all programs could be held in compact source form and there would be less need for storing the linkage and load modules, which often more than double the space required in a filing system.

I find that it is very difficult to persuade people of this, so let me repeat my point. If the language translator is as fast as the linkage editor, there is no need for a linkage editor and so there is even less of a barrier standing between the programmer and his machine.

Now there are many who do not approve the idea of fast compilation. They like to feel that the computer is working very hard on digesting the programs they have written; after all it is only fair that the computer should expend as much, if not more, effort on the task of producing a program as the programmer has. If, in fact, the extra time taken by the translator really simplified the task of the programmer, no one would grudge this, but we have already agreed that this is not the case. Thus the best a translator can do is grant immediate access to the computer. The increasing number of programmers who have enjoyed the privilege of fast turnround must surely recognize the boon of fast translation that makes this possible.

2 The next requirement in a programming language is efficient execution, in terms of both speed and space. Of course, this is not an accurate statement of the requirement; as we agreed before, no programming language can force the programmer to write efficient programs. The real point is that a good language gives the programmer firm control over the efficiency of execution, in terms of both speed and space. It does not mislead him by oversimple notations into thinking that an operation is efficient when it is not. It does not use the same notation for fast and slow operations when the distinction in speeds is made by some obscure optimization process of which the programmer has no understanding and therefore over which he can exercise little control. Above all, the machine code produced by the translator must not be significantly less efficient and compact than code that would be natural on the machine in question; in other words, the use of the language should not significantly reduce to the programmer the apparent power and capacity of his machine.

Again there are many who would dispute this point, feeling that the use of a programming language is such a vast privilege that it is worth sacrificing a substantial proportion of the power and capacity of the hardware; hardware, they claim, is getting faster and larger anyway. To deal with the second point first, fast and large computers still cost more than slow and small ones and it is not acceptable to throw away needlessly the hard money that the extra power has cost. Secondly, the problems we want to put onto a computer are in general growing at least as fast as the capacity of the computers available to solve them. Finally, if there does happen to be spare capacity available on a computer, it is the duty of the language designer to place the whole of the benefit in the hands of the progammer, who can far more profitably use it to simplify the really difficult parts of his task, that is, to shorten the search for refined algorithms, to use clear but redundant coding, and above all to be able to afford a higher degree of modularity and structuring in his program design than he would otherwise be able to tolerate. Again we conclude that it is the highest duty of a programming language not to stand between the programmer and the power of his machine.

3 The third requirement of a programming language is what I call security. One aspect of security is that the language should be a self-sufficient means of communication in both directions between man and machine, not only in the case of working programs but also for the vast majority of actual programs that contain errors. This means that the response of a computer to an erroneous program must be entirely predictable by the programmer in terms of the concepts of the language itself, requiring no recourse to any other conceptual apparatus or low level technique such as storage maps, octal dumps, or even hexadecimal dumps. The language designer must by all possible strategies attempt to increase the range of errors that cannot be made or, if made, that can be detected by the implementation itself, preferably at compile time rather than during program execution, which may be too late. This task of forestalling and detecting programming errors I regard as the major benefit of the use of a high level programming language. The real reason why people adopt a high level language is not because it gives some magical power to use facilities that are not available to the low level language programmer, that is absurd. It is simply that these facilities can be invoked more certainly and more reliably than before and with less danger of programming error.

Again, I regret that there are some experts who would dispute that the forestalling and detection of programming error is a worthy objective of high level language design. They resent the amount of extra declaratory information and the redundancy of coding that is the price one must pay for security and they believe that a programming language should try to make some sort of twisted sense out of almost any program text with which it is presented. In this, of course, the high level language

> is reproducing a characteristic feature of binary machine code. The control unit of a digital computer is quite happy to make sense out of any stream of binary digits presented to it, however curious, unexpected, and unintelligible that sense will be to the programmer who happened to make a small error in his program. However, in my view, it is exactly this feature of machine code programming that persuades us to turn to high level languages.

This has been a rather abstract summary of the qualities I believe a better programming language should have: fast and efficient translators, good run-time efficiency, and high security against error. It is now possible to recall my remark that I do not consider PL/1 to be in any significant respect the sort of improvement we are seeking in programming language design. Its design objectives run directly counter to the third and most important of my requirements, security, and thus I would be inclined to deny to the language the title of *high level* in any meaningful sense. In spite of this, it is such a large and irregular language that it is not possible to implement in a manner that permits simultaneously high speed of translation and high speed of execution. Nevertheless, I predict a bright future for PL/1. Its main competition in the next ten years will come not from any new and better language but from the old faithfuls, COBOL and FORTRAN. This is, of course, a consequence of the fact that there is no real demand for better programming languages today or in the immediately forseeable future.

But perhaps my analysis is wrong. The objectives of fast compilation, efficient execution, and even high security, may be widely recognized as the correct goals for a good programming language but it is widely thought that these are incompatible and cannot be simultaneously achieved. Many people believe, for example, that there is a basic conflict between high speed of translation and high efficiency of execution. An even wider belief is that security at run time is not achievable without resorting to semi-interpretive techniqes, with disastrous loss of efficiency. Thus it is believed that you need several implementation options: for example, fast compilation, slow execution, and good security during program testing, or slow translation, fast execution, and no security for production runs. But this is ridiculous. It is on production runs that the security is most required, since it is the results of production runs that will actually be trusted as the basis of actions such as expenditure of money and perhaps even lives. The strategy now recommended to many programmers is equivalent to that of a sailor who wears a lifejacket during his training on dry land but takes it off when he is sailing his boat on the sea. It is small wonder that computers acquire a bad reputation when programmed in accordance with this common policy.

But to return to the main point, it is widely believed that it is impossible to reconcile the requirements of high speed translation, high speed execution, and high security against error. There can be no doubt that, with languages like PL/1, COBOL, and FORTRAN, there is indeed a more or less unreconcilable technical contradiction between these objectives. However, there are now strong grounds for optimism that certain directions of language design in the last five years are showing a way that, with a very small amount of assistance from the hardware, these objectives may be reconciled to a remarkable degree. These directions are based firmly on a deeper understanding of the theory of computer programming and on the practicalities of programming language implementation. I cannot in the present talk explain all the discoveries that underlie recent progress but I can list a few of them.

1 A language syntax that permits top-down analysis with no backtrack is not only easy for human comprehension, it allows also very fast translation and good error diagnosis.

2 The most efficient structure for a translator on a machine with reasonable core store is the single pass translator plus loader. Consequently, declarations should precede the use of invented names.

3 The type of the result of every operation should be known at compile time from the types of the operands and the identity of the operator.

4 Checking the match and consistency of parameters and arguments of a subroutine can and should be accomplished at compile time.

5 The proper use of pointers, references or addresses, can always and should always be rigorously checked at compile time.

6 Loops should be programmed explicitly as such and not constructed by means of jumps. This will be of simultaneous benefit both to the user of the language and to its translator.

7 The use of procedures or subroutines, their entry and exit, and the parameter passing, should be as efficient, or very nearly as efficient, as in machine code. My blood runs cold when I read manuals that recommend the avoidance of the use of procedures in a language such as PL/1. Procedures are the only tool

that the poor programmer has for understanding the complexity of his program. I think it is terrible if the language is designed in such a way that procedures cannot be used because of efficiency considerations.

These technical discoveries are not just my own personal view. They are coming to be quite widely recognized among those who have implemented and used more than two programming languages and therefore have some justifiable claim to expertise.

POLITICAL REQUIREMENTS

Now we are faced again with our long-standing paradox. We have defined the characteristics that we demand of a better programming language. We have hinted that the technical means of achieving these characteristics are becoming established. There are even languages in existence that incorporate many of these technical advances. Why is there so little evidence of the widespread use of better languages today? The answer is the same as given before: there is no general demand for a better programming language. The time has come to give an explanation of the paradox. The explanation is simple and can be derived directly from a list of requirements a programming language must satisfy in order to be adopted by programmers and managers of the present day:

1 it must be supported by IBM
2 it must link in with existing methods of data storage and computer operation
3 it must enable existing programmers to maintain their familiar programming habits

Note there is no mention of the quality of the language in these three requirements. Unless these political requirements are met, there is no point in talking about quality at all. I have no quarrel with those who adopt this point of view. In the world of practical programming, it represents a rational policy. But of one thing I am certain: at the present time there is no known method of reconciling these three political requirements with any of the three previous criteria for programming language quality. And that, in one short paragraph, is a full and complete justification of my oft-repeated view that, although there is some ground for hope in the development of a better programming language, there is at present, and in the foreseeable future, very little demand for such a language among its potential users.