Insight, not numbers

P. A. Samet

Computer Centre, University College London

This paper is based on an inaugural lecture given at University College London on 19 October 1971. The text discusses the growth of Computer Science as an academic subject and the benefits a university derives from operating its own computing service, in relation to the training of competent practitioners of the computing profession.

(Received October 1971)

Charles Babbage, who first had the idea of building an automatic computer, died exactly 100 years ago, almost to the day.

The first stored program electronic computers began operating in the late 1940s. At that time the late Professor Hartree predicted that a single machine of the same modest size as the EDSAC machine then under construction at Cambridge University, placed in London, would be adequate for all of Britain's computing needs. In the intervening 20-odd years machines have been speeded up by a factor of 1,000 or more, they have increased enormously in capacity, sophistication and complexity (rather less so in cost), there are several thousand machines operating in this country, and we still have a shortage of computing power. This is not just Parkinson's Law in action. Computers are now widely used in fields that previously had little apparent need for computational ability, in commerce, in social science, in arts subjects, as well as in scientific subjects. In a sense, Hartree was right—a single machine could have done the computing that was being done. What he did not foresee was the explosive growth of our ability to use machines.

Initially, programming was a skill to be acquired only by persons with high technical, not to say mathematical, ability. Early machines were too limited for anything like algebraic programming languages to be viable, although R. A. Brooker in Manchester was an early pioneer with Autocode. The spread and acceptance of programming languages was undoubtedly a major factor in making computing more widely accessible, a topic I return to later.

Naturally, it was not long before the new techniques began to be taught. At first to postgraduate students, then rather tentatively to selected undergraduates, during the 1960s we saw the beginning of specialist courses in Computer Science, and recently, in September 1970, came the publication of a UGC-Computer Board report urging that all university undergraduates should be taught about computing. We have also seen the growth of the computing profession and our own British Computer Society has recently started its system of professional qualifications in computing, the first of its kind in the world. I want to spend some time discussing the growth of Computer Science, a subject which in a few years has developed from nothing to an important position in our universities, being available as a specialist field of study in more than 20 institutions in Britain and many more abroad.

The early courses were almost entirely concerned with Numerical Analysis, a branch of mathematics that had long been despised by many conventional mathematicians but had suddenly assumed importance when it was realised that the computers offered the ability to find numerical solutions to major problems. Small and limited as the early computers were, they were so fast that problems which could not even be contemplated a little earlier could now be solved in a matter of days, even hours. My astronomer friends tell me that the calculations that were done by John Adams in the 1840s in predicting the position of Neptune, taking about two years of exceedingly intricate work, could be done in their entirety and considerably more accurately on the IBM 360/65 here in College in well under 2 minutes. Adams won world fame and lasting renown for his calculation, a 2 minute calculation on the 360 is a 'short job' and we handle hundreds every day.

The next topics to be included in courses were programming techniques and logical design, explaining features that make the machines work. They were still aiming at the mathematically inclined student, mostly because few computer people knew anything else.

It was only in the early 1960s that enough had been learned about the underlying principles of things like language design, compiler construction—a compiler is a special translation program for programming languages—and the handling of non-numerical information that a coherent body of knowledge began to emerge, to knit together the isolated facts and techniques that had been found. It is worth recording the names of those whose research and exposition of their work has had such a great effect: Dijkstra, Naur and his collaborators in the design of ALGOL, McCarthy, and Strachey. Any list like this is incomplete, I have given the names of those that have had the greatest influence on my own understanding of Computer Science.

And so I come to my first question, 'What is Computer Science?' It is a question I am often asked and find difficult to answer in a few words. However, I can shirk the task no longer, a professor giving his inaugural lecture ought to know what his subject is about. The trouble, of course, is that I know what it is about, at least I know what I class as being inside Computer Science and what is outside. The best definition I can think of is 'The systematic study of constructive methods—algorithms—and of systems for the automatic performance of such algorithms'. The relationship between Computer Science and information is like that between physical science and energy. Algorithms transform information in the way physical processes handle energy.

The mathematician will claim that he, too, is concerned with the study of algorithms. There is some justice in this claim but I do not think that constructive methods occupy the same central position for the mathematician as they do for the computer scientist. The mathematician is concerned with structure, with existence. I speak as a mathematics graduate, these were the things that excited me when I was a student. They still do excite me, but mostly in relation to the constructive method approach I have learned from working with computers.

There is a difference between an algorithm and a formula. A formula, if it exists at all, gives the nature of the solution whereas an algorithm, which exists for any soluble problem, tells us how to achieve the solution. An analogy may be helpful: the formula for sulphuric acid is H_2SO_4, the 'lead chamber process' is an algorithm for making it.

Well, then, if Computer Science is the study of algorithms, is

it an academic discipline, a subject worthy of study in a university? Or is it, 'merely', a technique fit for learning in a technical college or its equivalent? I cannot deny that there is a certain amount of skill that has to be learned. That, however, applies to many other subjects, too. One could say that Engineering is also, 'merely', a collection of techniques, so is Medicine—with Surgery as a craft skill that has to be learned—and I could go through all university subjects like this, subjects where we do not question their fitness to be in the academic catalogue. Of course, we continually wish to apply what we learn in Computer Science and test it with programs on a computer, but the medical student and the engineer also have to put their knowledge to the test in practical situations. I believe the confusion arises in many people's minds because they confuse *Computer Science* and *programming* or, rather, *coding*. Coding is a skill, it can be very rewarding (aesthetically, as well as financially!) and very difficult, but really has little intellectual content. What most people have experienced so far is the coding of problems in a particular programming language. I do not deny the importance and value of this skill. However, its relationship to Computer Science is like the ability to speak a language, say French, and what is studied in a language department in any university.

Notice the way I have used the word 'coding', rather than the more usual 'programming'. To my mind, 'programming' is the formulation of an algorithm to do a particular job, whereas I reserve 'coding' for the task of implementing the algorithm in a form suitable for a specific machine, the strategy and the tactics of problem solving. One of the causes of confusion is that programming languages, like FORTRAN and ALGOL, serve both as a way of describing the algorithm and as an acceptable input to a computer for the implementation.

I feel that the distinctive contribution that these languages have made is the provision of unambiguous notation for algorithms. Previously, a procedure for carrying out a process might have been expressed in words, or as a series of mathematical formulae, or in a variety of other ways. Particularly tiresome and difficult to describe were repetitive process, where one needed words like 'and so on', and recursive processes, where one of the subsidiary stages of the whole process is a 'smaller version' of the whole process. It is not easy to give meaningful non-technical examples of a recursive process. The parsing of sentences is such a process, because subclauses have the same general structure as the main sentence, in particular they can also have subclauses. The process stops because sooner or later we reach words rather than clauses or phrases, so there is no further subdivision. In London University one might suspect that the whole process of running the university is recursive, with the University dealing with the UGC, the Colleges with the University, departments with the Colleges, etc., and a cynic might feel that there are times when the recursion has no end.

Programming languages allow us clear, concise descriptions of what has to be done. We are able to describe the state of our process at any stage during its execution, or at least how one gets there and what is to happen next. We can now be sure that the description of the algorithm means the same to two different persons and also that it means the same to their, possibly quite different, computers. (Well, it should, and does so except for possible errors in their translation programs and differences caused by holding numbers to different accuracies.) An interesting by-product has been the international acceptance of some programming languages.

The key to this precision of description is that we have learned how to describe the languages themselves. The early languages were described, in effect, by their implementation on specific machines. The big step forward was the publication in 1960 of the ALGOL report, (ALGOL stands for 'Algorithmic Language'), which gave a machine independent description of a complete language. It was all so new and revolutionary that it took a long time for people to understand what it was all about. Apart from some basic symbols, the definitions of all terms were themselves recursive, which did not help initial understanding. It need not concern us at the moment that some obscurities and ambiguities were found in the definitions, and that revised reports have had to be issued. There is also a flourishing school of theologians who spend much time disputing with great vigour about what exactly is meant by various obscurities.

ALGOL itself had fairly wide acceptance in Europe but not in the USA, and never really among some of the large machine users (mostly because they used American equipment without ALGOL compilers). The influence of ALGOL, however, has been immense, in setting a standard, in language-definition, in the communication of algorithms and in being the starting point of concepts and techniques that have proved invaluable ever since.

Having an adequate notation for describing the work one wishes to do is necessary. There are many examples in science where progress was slowed down because no appropriate notation was at hand. A famous example, for mathematicians, is the development of calculus by Newton and Leibniz. Newton's dot notation was not up to the task in the same way as Leibniz's '*d*/*dx*'. I remember one of my own teachers explaining that Newton would have been able to discover Laplace's equation, which is central to much of mathematics and physics, more than 100 years before Laplace if he could have written down partial derivatives, he had all the necessary physical intuition and mathematical insight. It is my contention that one of the reasons why computers have become indispensable tools to so many activities is that it has become relatively easy to describe complicated algorithms for carrying out these activities. This has been the distinctive contribution of programming languages. Not for nothing do we also call them '*problem oriented languages*'.

Through the study of programming and programming languages we are thus led to the study of the whole field of problem solving, what is to be done, how we set about it and what tools we have available. This is an important, basic and worthwhile area of academic study. The fact that it is young as an organised discipline is irrelevant.

I have heard cogent arguments, from people that I respect, that it is impossible to teach Computer Science properly at present because it is so young. The argument goes roughly that all we can teach is the state of the art, there is so much we do not know and there are no theorems to prove. Actually, of course, much the same criticism could be applied to all subjects and on that basis nothing could be taught in a university. 'State of the art' will not stand up as valid criticism. Few things could be taught if we took much notice of 'There is so much we don't know'. What about the 'no theorems' part? This, too, is shaky ground. It is not theorems that are important, but the ability to design a system and predict its behaviour. If the objectors had confined themselves to pointing out the shortage of suitable student texts they would have had a stronger case!

Having said something of what we study, I must say something about the general attitude that we ought to take. It is my view that in attempts to make Computer Science academically respectable too many people have forgotten what Computer Science is all about. The result has been a series of highly theoretical courses, devoid of practical content and application, lacking in all inspiration, producing graduates who could answer theoretical examination questions but with no idea of how to solve even the simplest problem. I was surprised, not long ago, when visiting another university to find that its Computer Science undergraduates complained—rightly, I feel—that in their whole student career they had never once actually seen the computer system operated by the university.

I do not think any medical school would feel it right to produce doctors who have never seen a patient.

It seems to me that we should adopt the engineering approach, rather than that of pure science. Let me explain. Pure science observes the real world, and constructs a theoretical model of this for analysis. Predictions derived from the model are compared with occurrences in the real world, leading to refinement of the model. On the other hand, whatever the engineer derives from any model he may have made, in the end he had to work in the real world. He may calculate the stresses in an arch dam, constructed of perfect materials in a symmetrical U-shaped valley, but when the dam is actually built it is not possible to have perfect materials or the shape of the valley one would like. The engineer has to work within the confines of the real world. So it is with the Computer Scientist. One may plan all sorts of things, but when it comes to implementation one has to take note of the hardware that is available, of the operating systems and compilers that are available. For myself, I have found considerable satisfaction in having to work within the limitations imposed by the world and its tools. I do not think I am alone in this.

There are other advantages in the 'engineering' or 'technology' attitude, that I shall return to a little later.

Now I want to turn to the problem of doing a university's computing. The operation of a computer centre absorbs large sums of money—in London the annual running costs of all of the university's facilities come to about £1½ million quite apart from the capital sums needed—there are requirements for expert and experienced staff, there are requirements for space.

In the early days there were few computers and universities had little choice but to install their own. Now, however, there are several service bureaux operating powerful machines and access to facilities is not too difficult from almost anywhere in the country. One may reasonably enquire whether it makes sense for a university to operate its own computing system. Is a properly equipped computer centre an essential part of a university, like a library, or is it only an expensive status symbol for the Vice-Chancellor? There are actually two parts to the question, does a university get a better service by doing it in-house and does a university lose something useful and important through not having its own facilities? In London the problem comes in yet another form, whether it is better to centralise all facilities or should there be some distribution between the colleges? As Director of a college Computer Centre I can hardly fail to have strong views!

The first point to note is the curious nature of the workload of a typical university. Experience in several universities, in all parts of the world, shows that there is a predominance of small and short programs. Typical figures are 50% by number of jobs taking in all only about 10% of the available time. This type of load distribution occurs very rarely in commerce and industry, but is also found in some research establishments. Few of these programs have a long lifetime, mostly they are written *ad hoc* for particular calculations. A consequence is that the percentage of compilation runs is quite high—economically this is equivalent to saying that development costs are not spread over a long production life. There is a constant demand for the results of these short jobs to be returned to the user within a very short time and many systems now try to provide this fast 'turn round'. Some years ago we used to think that turn-round times of about 3 hours were very good, now we try to do it in a matter of a few minutes. A technical development of the past few years has been the introduction and increasing use of interactive systems, offering several users simultaneous access, often through consoles in their own offices. In this case, turn-round—now called 'response time'—can be cut to a matter of seconds. While such systems are very useful not all jobs are suitable for the interactive mode of work, just as the telephone and the postal service are complementary aspects of the communication system we all use. The ability to get one's computing done quickly and to have the results in one's hands in such a short time is a great help to many people. The short interval between job submission and receipt of results means that one is still 'in touch' with the whole problem. Equally important, although often overlooked, is the requirement that this type of service be available whenever it is needed, all day and every day. In other words, computing power must be as conveniently available as electricity or water supplies.

I think the question I have asked, about doing the computing in-house or not, is now transformed into the equivalent problem of how best to provide the kind of service I have described. It will certainly be necessary to provide equipment in various parts of the university to give convenient access to the computing machinery, wherever the processing is done.

After this, we come to questions of equipment limitations, such as available data transmission speeds. It is possible to get reasonably fast transmission lines, at a price, but the general data transmission speed is quite low. Of course all these things will improve in time. At the moment the generally used speed in London University's own network offering 'batch processing' on the CDC 6600, a mile away at Guilford Street, is 2,400 baud which corresponds to 300 characters per second. The faster lines operate at about 5,000 characters per second. Compare this, however, with the input/output capacity we have in *our own computer room*, with only two card readers, two printers, a papertape reader and a card punch, of about 8,500 characters per second (and we have several links as well). For large periods of the day we find that all of this gear is in continuous operation, there is often a backlog of card reading or printing, and we really could do with some more equipment of this kind. If we had to rely on transmission links to a machine elsewhere, our present input/output loading would suggest that, to give a service comparable to what we can offer now, we would need 28 of the slow links or two of the faster type. Without provision on this scale, turn-round would deteriorate—i.e. people would have to wait longer for their jobs even if the processing speed were infinite. Well, the CDC 6600 has provision for a total of four fast links and about 20 slow links to cover all parts of the university, so if the equipment for UCL were on the same scale as for other colleges the local users would suffer considerably. And it does not matter, really, that the links would be to a university-operated machine. The point is that one of the constraints on our decision making must be whether the information can be transmitted sufficiently quickly to a remote site to give the service our colleagues require. Not only require, the service *their* colleagues, and rivals, elsewhere *get*. As an aside, it has often struck me that the real attraction that sucks scientists into the Brain Drain is the availability of superior equipment and facilities, rather than higher pay. If I am right in this, the most effective counter-attraction is the provision of well-equipped laboratories with adequate supporting facilities, including technical and secretarial staff.

To resume my argument. We have just seen that if the load is heavy we may need several wide-band transmission links. If these have to go over any appreciable distance (more than say 10 miles), the cost is very high indeed. Then there is the service that is provided. Here, of course, the answers depend on just how much one does. An organisation that can keep a large machine occupied for 24 hours per day, every week, is not going to save by hiring the same amount of time on a machine of the same capacity. In London we are in a very special position: the machines operated by the university—and fully occupied—are larger than most service bureau machines in this country, so we would gain nothing by going outside even if we could. But this need not be the case elsewhere, and it may well be that some of the work that now fully occupies a small machine at the University of X might be done better (speed, turn-round, cost) on a large machine at Y. It may make good sense for a

proportion of large jobs, requiring more resources than can comfortably be made available locally, to be sent elsewhere for processing. In some cases the post or the railway may prove to be a satisfactory link. Many universities have operated good services like this.

All of this has been concerned with the technical problem of getting work to and from a computer. At the present time we are constantly being urged to pay much attention to cost. I do not dispute the necessity of this but would like to see some emphasis being placed on *value for money*, not just money. Following the theme I have been developing, I now ask whether a university actually gains anything useful by running its own computing service. In other words, even if it is cheaper to send it all to another location, what other factors do we have to consider? A similar question could arise regarding libraries—if there are good facilities for high speed facsimile transmission, do we still want our own library or could we rely on a centralised national library?

Although it is difficult to quantify, what we gain is *expertise*. This is important. The presence of a large computer and its associated specialist staff within the university community brings a greater awareness to other disciplines of how automatic computing equipment can help in their fields. Having experts readily available, people one meets in the refectory and the common room, is one of the principal facilities a university service can offer its users. It is very hard to attract such people to a location where they themselves are not in continuous close contact with the problem of the computing system, as much as anything because their own knowledge does not stay up to date. Let me add immediately that the computing service also gains very greatly, by being kept in constant touch with the problems of its users.

There is another side to this, involving the scale of the operation. One of the difficulties that faces all university staff in science and engineering departments is this one of scale. What you do in the laboratory is only a small model of what goes on in industry. As a result, our students are not as well trained as we would like them to be. For example, we can make a model of a skyscraper block, but there is a world of difference between making the model and building the real thing. Again, producing a drug in quantities suitable for laboratory testing is a very long way from having the ability to manufacture the same drug for national use on a large scale. It is not just that the industrial process is a larger version of the laboratory process. Very often, a completely different technique is required.

Just as it is impossible to give students experience of problems of a size comparable to what they will meet later, so it is equally hard for the staff to maintain what experience they have. Unless one has constant contact one's knowledge is soon out of date, so great is the rate of change.

One of the few technical areas where our students are taught in an environment of the right scale by staff with constant exposure to it is medicine. I think that Computer Science can be another. Installations in universities compare in size, complexity and general facilities with the most advanced equipment available anywhere, if anything they are more sophisticated than many installations in industry or commerce. The chance for students to do work at the right level—of difficulty and size—under people who are constantly faced by genuine problems is one that should not be missed. To take advantage of the opportunity to extend and improve our activities like this we must have the facilities readily available.

There is one point I want to mention here, in passing, which is of great concern to many of us involved in the teaching of Computer Science and that is the provision of computing equipment specifically dedicated to the teaching of specialists. I have already commented on the value of having large-scale equipment in our universities. However, there are occasions when the Computer Science specialist requires to do something out of the normal run of things and it is extremely difficult to accommodate such legitimate requirements within the normal operating schedule of a computing service. What is needed is some equipment without service commitments. Generally speaking, the amount needed is really very modest, with a cost of only a few per cent of the university's main equipment. It is a surprising fact that whereas subjects like Physics, Chemistry, Engineering and the like are able to get small machines of the kind I have in mind, because these are essential to the proper teaching of their students or for their research with the computer being used in connection with other equipment and are able to treat these just like their other items of laboratory apparatus, Computer Science is assumed to have no such needs and is expected to use only the main computing machinery available. Indeed, although the Science Research Council quite rightly classes Computer Science as a technological subject the University Grants Committee does not yet accept it even as an experimental subject!

I now have to answer the implied question in all this, why I think it is of use for students (and staff) to know about problems of 'the right size'. To do so we must look at the history of large enterprises, and I shall concentrate on those projects that have involved massive amounts of computing. Some have been successes, some failures. My list is, of necessity, very selective. Among the successes we have to count the space science programme and systems like airline bookings. On the other side, I am afraid that some systems with much contact with the public, like local authority work, the banks, publishers' book lists and the like, the Stock Exchange, can hardly be classed as being outstanding examples of what one would like to do. Neither are the integrated management information schemes, which were to use the new technology to do things that were impossible earlier. And if you think, that I believe all successes are in technology while the failures are in commerce, let me say I do not think very much of the software industry either, with its monstrously large and generally inefficient operating systems, which are almost invariably late as well. I cannot get much closer to the computer than that!

What makes some enterprises work while others fail? To say 'management' is only partially correct. Certainly, in the very successful projects, the skill has been in organisation as much as in possessing the necessary techniques. The staggering thing with the Apollo projects to my mind is not that the rockets and men have gone to the moon and come back safely, nor that one could calculate precisely where they would be at certain times, but the sheer human triumph of organising a project involving such a vast number of people. I believe that over 1,100 programmers were employed on the flight mission project alone: most projects find difficulty in getting 10 people to co-operate successfully.

Good organisation and discipline are obviously needed. But don't these qualities occur in commerce and industry? Of course they do, so we must look for something else. I like to think it is the approach to problems, and this brings me back to my earlier remarks about the engineer's attitude. A very large number of organisations just have not realised that programming is not only a clerical activity. Programs constitute a large technical and financial investment, and are really in the nature of capital equipment. Yet it is very common to find a firm that buys a large computer, costing perhaps £½ million, and then relies on a few inexperienced, incompetent, untrained but highly expensive programmers to make it all work. The result is predictable disaster. Symptomatic of this approach is the attitude to documentation, which is very hard to come by in most installations. Yet the self-same firms will insist on highly qualified accountants to look after their money and highly skilled engineers to design their products. It seems curious that they do not take similar care over their computing when it is to take a central role in the company's operations. I notice that

what I have classed as 'successful' has been in organisations that have a great deal of technological 'know-how' and have realised that a similar approach might be useful here also. The engineer's attitude to quality control and product testing also has much from which the computing profession could learn with profit.

We must learn from both the successes and the failures. It is for these reasons that I think scale and approach are so very important, and that the requisite knowledge and experience must be available within the university.

Before I close, I want to mount one last hobby-horse. So much of one's efforts in preparing a problem for a machine go into the organisation of the necessary arithmetic that many people appear to lose sight of their final objective. The intermediate entities are all numbers and it is all too easy to produce vast quantities of printed sheets, filled with numbers but very difficult to interpret. Humans have developed some skill in calculating but not many are good at interpreting complex relationships when these are expressed only in numerical form. If the related parts are several pages apart the task is practically impossible. Yet the presentation of output in an intelligible format is rare.

Graphical output is most valuable for presenting results in an informative way and microfilm plotters, producing films, are excellent for showing the behaviour of time dependent processes. There is no other way of doing this, pages and pages of printed figures are useless in this respect. It is generally believed that such plotters are very expensive, yet their cost is comparable with that of the very high speed printers that are now available and which are regarded as indispensable by most installations. I find there is very little realisation of the potentialities of such film-making devices.

Although my own experience has been confined almost entirely to scientific use of computers I believe that much of what I have said about the approach to and organisation of computing projects applies equally to commerce.

To sum up my remarks I can do no better than quote Richard Hamming's famous and elegant epigram, that 'The purpose of computing is insight, not numbers'.

Correspondence

To the Editor
The Computer Journal

Sir,

In spite of the *ad hoc* nature of the FORTRAN language, it is widely used: many proposals for 'improving' the language have unfortunately been adopted unilaterally by compiler writers. Chambers (1971) is to be congratulated for exposing his proposals to public scrutiny, and I should like to add some comments to the inevitable clamour which will follow. Paragraphs of Chambers' paper are referenced in square brackets [ ].

[2.1] *Character data:*
It is proposed that quote signs be accommodated in a character constant by the use of a processor-dependent mechanism. This defies the requirement of upward compatability: the escape mechanism should be fixed by the standard. Actually, it would appear that the old Hollerith constant remains unambiguous in the new language and could be kept as an alternative representation of a character constant.

[2.1(d)] Chambers proposes operators for comparing character expressions of arbitrary length: his footnote concedes that the results will be implementation-dependent. We ask (A) can the dependence be avoided and (B) can a similar result be achieved in a different way. Certainly (A) can be achieved, either by demanding that ASCII be the internal representation or by having a complicated comparison algorithm which effectively translates into ASCII before making the comparison. Let us, however, ask what is the purpose of X ω Y, where ω is a relational operator and X and Y are character expressions of possibly-different length. One major application would be sorting character strings into 'alphameric' order. But a comparison technique which encourages the user to compare the whole string at once is likely to be less efficient than one where characters are compared one at a time from the left. The only essential facility is the comparison of single characters in ASCII order. I propose that this be done, not by hiding a complicated algorithm in a simple-looking facility, but by providing a built-in function IFIXCH, of type INTEGER, which takes a single argument of type CHARACTER and returns the ASCII code value. Note that the internal representation remains arbitrary but the function always returns the ASCII code. Then operations of the type .GT. on the resulting integers will be performed efficiently and will be implementation-independent.

On this arguement alone, the proposal for IFIXCH does not look very exciting. But it permits characters to be used for indexing—a very powerful technique which would be impossible under Chambers' proposals. Hash table management would also be possible. (An inverse of IFIXCH would be provided.)

On the other hand, relational expressions can be formed between character expressions using .EQ. or .NE. without regard to the internal representation, and this construction could therefore be implemented in the new language.

[2.2] *Internal formatted conversion*, and [2.6] *Data transmission:*
To eliminate unnecessary restrictions, I propose that, instead of the format statement number f, the specification should permit a label expression $\mathscr{L}$ (specifying a labelled format statement in the current program body) or a character expression $\mathscr{C}$ (specifying the characters of a format specification explicitly). A FORMAT statement could then be selected on the basis of an indexed label array, or a FORMAT specification generated dynamically in a character array by use of ENCODE.

[2.7] *Array assignment:*
The footnote to section [2.7] mentions an objection made by the referee. Unfortunately the cure is worse than the disease! Chambers' proposal would require the results of the right hand side to be built up in temporary storage and only afterwards transferred to the array specified on the left hand side. Surely the only practical cures are:

(*a*) to complete the evaluation of one element of the array at a time, and to determine the result of pathological cases by fixing the order of evaluation in the specification, or
(*b*) to state in the specification that the effects of pathological cases such as A = A / A(1, 1) are undefined.

Yours faithfully,

A. J. FLAVELL

Max-Planck Institut für Physik und Astrophysik
8 München 23
Föhringer Ring 6
31 August 1971

Reference

CHAMBERS, J. M. (1971). Another round of FORTRAN, *The Computer Journal*, Vol. 14, pp. 312-314.