please take one
Xerox.

Notes on the standardisation of Programming Languages

C.A.R. Hoare

Preliminary Draft - February 1975.

Summary. This paper surveys the conditions for successful standardisation of products and designs, and enquires how far computer programming languages satisfy these conditions. It concludes with suggestions for future action in research, development, and standardisation.

1. Purpose and Motivation.

The first reason for the establishment of a standard is the possibility of thereby saving time, effort and money. A standard can save money by assisting in the interchange of products, people, and ideas. It serves as a form of contract between groups of suppliers and groups of customers; it enables products of different workshops and manufacturers to operate harmoniously together, and thereby fosters economy, competition, and progress. It permits people trained on one manufacturer's equipment to transfer without too much retraining to another's. And finally, a standard may serve as guarantee of a certain degree of quality and safety in a product which claims to conform to the standard.

These three aspects of standardisation may be well illustrated by standards for punched cards and card handling equipment. The purpose of the standard is to permit a card supplied by one manufacturer to be punched on a card punch made by another and read by a card reader made by yet a third. The layout of the keyboard on the punch conforms to a standard which permits ready transfer of trained punch operators; finally the quality of the paper of which the card is made must be adequately robust to stand repeated mechanical handling. And these standards are remarkably effective; the trouble-free use of alternative suppliers of cards and card equipment is very widespread.

The primary purpose of a programming language standard is to permit free interchangeof programs between computers made at different times and places and even by different manufacturers. And certainly, compared with assembly code programming, the early languages were quite successful in facilitating program interchange between machines with different instruction codes. But compared with the free interchange of punched cards and other physical products, the situation is by no means ideal. It is a rash man who proposes to transfer a program of any size between one machine and another, and does not expect any trouble.

There are several reasons for this. Firstly, operating systems and other software often get in the way. Secondly, the art of programming language design was still quite backward, and did not yet permit the construction of a watertight standard for a machine-independent language. Thirdly, implementors have not been prevented from encouraging their users to take advantage of enticing "extensions" which are not part of the standard.

The second motive for standardisation, namely, the transfer of trained personnel, has also been a qualified success. When machine codes were simpler than they are today, a good programmer could learn a new assembly language in a few weeks. Today, he may take nearly as long to learn all the peculiarities of a different implementation of the same language on a different machine or in a different operating system.

The third motivation for standardisation, namely control of the quality of the implementation, has been almost completely neglected; and many implementations of standard languages have been unreliable and inefficient, almost to the point of unusability.

Perhaps it is therefore fortunate that the costs of transferring programs from one machine to another are being reduced by other methods, such as the introduction and perpetuation of wide ranges of compatible machines such as the IBM/360 and ICL/1900. Furthermore, an occasional complete rewrite of a large program is often advantageous in improving its usefulness and structure and future adaptability - as long as you are not faced with a simultaneous rewrite of all your programs. And in some applications, like real time command and control, the requirement to move a program from one machine to another is fairly uncommon.

There has often been expressed the hope that the use of a common standard programming language will permit the ready transfer of programmers from one application area to another. This was the hope underlying the design of PL/I as a common language for scientific and commercial programmers, and which leads to the suggestion that COBOL should be adopted as a standard programming language for real time applications. Unfortunately, the major problem in moving a programmer from one application area to another is his lack of familiarity with the new application area, a lack which may take several years to overcome. A simultaneous change to a new language is a comparatively minor problem - and it may even be an advantage, since a well-designed application-oriented language can actually hasten and sharpen his understanding, and suggest useful programming methods. A programmer who is unwilling or incapable of changing his language is probably well advised not to change his application area either.

This discussion leads to the conclusion that the standardisation of programming languages has not so far been altogether successful. The main part of this paper examines some of the reasons for this, by surveying some of the conditions for successful standardisation namely, the technological simplicity or even irrelevance of the major decisions involved, the wide acceptance of a suitable vocabulary and method for defining the standard, the specification of an appropriate range of tolerance to permit various implementation methods, the existence of a cheap and widely applicable yardstick for testing whether an interchangeable product meets the standard, and the prevalence of a favourable political climate. In each case, some special difficulties for programming language standardisation are revealed.

However these somewhat negative conclusions in no way detract from the benefits that can be obtained in programming practice by the adoption of a suitable high-level language in place of assembly code, even if that language does not have the qualities or even the political support necessary for standardisation in the strict sense.

2. Technological Irrelevance.

The design of a standard necessarily involves taking a number of decisions. For many standards, these decisions are purely conventional, and have little technological significance; they could as well be made one way or the other, without expense or loss of quality. There is no particular reason why punched cards could not have been half an inch longer or wider. In some cases later discoveries reveal that some earlier standard is somewhat inappropriate: I am told that a seven-foot track width for the railway would have been technically superior to the present four and a half feet; and that a superior layout of keys of a typewriter could improve productivity of typists by a measurable percentage. However, these technological improvements would be quite marginal compared with the cost of making a change.

It is widely believed that the choice or design of a programming language has this same property of technological irrelevance; and in certain aspects this belief is correct. It does not matter whether a callable piece of program is called a PROCEDURE or a SUBROUTINE; any more than in natural languages does it matter whether a feline animal is called "a cat" or "un chat".

It is true also that many of the baroque features of existing or proposed programming languages are completely irrelevant to the needs of programmers and detailed discussion of their design is even more so, ~~irrelevant.~~

However I believe this reasoning is dangerous. The important decisions in programming language design are not merely a choice of notation to denote a well-understood concept; they are concerned with the design of the concepts themselves, and they require fundamental decisions about the nature of machines, of computations, and most importantly, of the human activity of programming. Furthermore, the logical interdependence of the design decisions is so great that the achievement of

mere consistency requires a high degree of technical skill and vigilance. And finally the solution for baroque irrelevancies is surely to remove them from a language before standardisation.

I would like to argue that the design decisions involved in a programming language are of the utmost technical and economic relevance, and that if the taking of these decisions is regarded as an arbitrary matter for rapid standardisation, the resulting language could add very significantly to the costs of programming and using computers. I would like to support this view by a short thought-experiment. Imagine first a language, described somewhat inadequately by a manual of several hundred pages. The manual is sufficiently incomplete that successful use of the language depends on a study of implementation manuals of comparable size. The problems of programmer training are immense. The complexity of the language is such that not even a trained programmer can understand it wholly, and therefore constantly runs the risk of unexpected errors and inefficiency in his programs. The diagnostics are so numerous and abstruse that a specialist advisory service is required to decode them; and even so, the language is so full of pitfalls that the vast majority of errors remain undetected by the compiler. In tracing the effect of an error in a running program,the only diagnostic aid sometimes has to be a hexadecimal dump. Compilers for the language are so large and complex and unreliable that there is a constant risk that they have introduced yet further errors into a program. The testing or updating of a program of significant size involves heavy compilation costs, and the programmer is effectively deprived of conversational debugging and even fast turn-round by the large bulk and slow speed of the compiler and/or object program. The inefficiency of the object program requires purchase of larger main stores and backing stores and of more and faster processors in order to run the programs. Programs expressed in the language are so cryptically unreadable that the task of maintaining them let alone adapting them to meet changing circumstances is a daunting one. And finally, the language is full of machine and implementation dependent features, which are central to the use of the language and can hardly be avoided; and the chance of writing a non-trivial program that could be transferred to another machine, or even another operating system, is negligible.

There can be little doubt that the total direct costs of using such a language, in terms of programmer effort, advisory services and hardware, must be rather great, not to mention the indirect costs of the unreliability and inflexibility of the resulting programs. And the cost of implementation and promotion would also not be negligible. These are my grounds for supposing that the decisions involved in programming language design are of great economic importance to the user, and are not purely matters of arbitrary choice.

3. Descriptive precision.

A standard serves as a form of contract between groups of suppliers and customers for a product; and for this reason it is usually formulated in a precise, almost legalistic, terminology, designed to forestall misunderstanding. It usually appeals to a widespread agreement among specialists and laymen on the terminology and descriptive method; for example, the use of measurements in centimetres leaves little scope for ambiguity or doubt.

For the clear definition of the context-free aspects of the syntax of a programming language, the Backus-Naur-Form, pioneered in the definition of ALGOL 60, was technically very successful, and has gained wide acceptance. However, for programming languages semantics there is no generally agreed method for their precise description, and each language has adopted a different style. The use of ordinary English prose has been proved ambiguous and ineffective, but no more formal notation has been agreed, although this has been the subject of much research. One proposed method is the Vienna Definition Language, which has been applied to PL/I; but alternative methods are the mathematical semantics proposed by Scott, and the axiomatic method proposed by Floyd.

Unfortunately, when these methods are applied to currently fashionable programming languages, they turn out to be extremely cumbersome and unilluminating; one feels like a keen geometer, who is trying to describe the Venus de Milo (or even some less attractive artefact) in Cartesian coordinates. Although it can in principle be done, it seems not worth the pain.

Another problem in the definition of a programming language standard, whether expressed formally or in English, is accuracy. The descriptions of many current programming languages are riddled with inaccuracy and ambiguity, and standardisation committees have spent many years trying to remove them, making hundreds of amendments per year. When they finally present the language for standardisation, it is not necessarily because all errors are removed, but merely because they are sick and tired of the whole project.

This is doubly unfortunate when we are trying to persuade programmers of the merits of absolute precision and accuracy, and they find that their most vital tool is so lacking in these qualities.

My own view is that the fault lies not so much with the description methods, but with the complexity of the product they are attempting to describe. Most successful standards apply to products of relative structural simplicity, like punched cards, paper tape, nuts and bolts, etc.; and even so, they stretch our descriptive powers to the limit. But if an attempt were made to standardise a more complex object such as a Boeing 707, the same difficulties would arise as in programming languages. Indeed, I suspect that a necessary condition for any progress in language design or standardisation is the achievement of greater simplicity.

4. Tolerance.

The most difficult technical decisions involved in designing a standard are not in determining the absolute dimensions of a product, but rather in determining the tolerance or range of permitted deviation around the absolute norm. For example, the tolerance on the dimensions of punched cards must be sufficiently wide to permit relatively cheap mass production; but not too wide to place an intolerable burden on the user's card handling equipment. The determination of a range of tolerance has immense economic and commercial implications. Fortunately, the decision once made can be quite accurately expressed by the conventional $\pm$ techniques of engineering measurement.

The need for tolerance is just as important for programming languages, to permit efficient and economic transfer of programs from one machine to another, with (for example), a change of word length. Unfortunately, there has been very little research into this problem, even at the level of number representation, let alone at the level of a complete language; and the details of arithmetic are either too rigidly defined, or not defined at all. If different implementations are permitted an unrestricted freedom to implement ordinary arithmetic in different ways, there can be little hope for program interchange.

The main difficulty is that there is no generally accepted method for restricting the range of permitted variation; and a programming language standard either has to specify something precisely, or states that it is implementation-defined or even undefined. Quite apart from reducing the prospects of program interchange, this has a bad effect on the quality of language design, since an implementor is permitted to give any result at all to a program which

invokes an undefined operation; he can even allow the program to run wild, giving subtly or totally incorrect answers or none at all.

It seems to me that the axiomatic method is probably the best tool for specifying a norm and permitted deviation in the design of computer arithmetic and other features; and it has two possible additional advantages, (1) that an implementor can specify his particular implementation decisions clearly and succinctly by additional axioms; and (2) the programmer can use the axioms to prove that his program meets the standard, as well as meeting its desired specification.

5. Yardstick.

One of the important considerations for the success of a standard is that there should be a relatively cheap and simple test to determine whether and how far a product meets the standard. The early standard for (British) proof alcohol was based on a test using widely available materials like guncotton and matches; and similar calibration tests are now available for standard lengths, times, etc. Even for punched cards it is possible to use a simple template to check the dimensions and positioning of the holes, so that in case of difficulty, it is possible to tell quickly whether the card or the reader is at fault.

In the case of computer programs, it is just as important to be able to tell whether a particular program meets the standard, and can safely be transferred from one implementation to another, so that if any difficulties arise, they can be correctly attributed to the programmer, or to the implementation. And one would think that the computer itself would be an ideal tool for making such a test, if only a program were available to do it. Even better, the test should be made by every compiler as part of its syntax and context checking. The PFORT project was an attempt to apply this method to FORTRAN, but unfortunately it cannot deal with any of the really difficult points, which are equally prevalent in other languages, and I fear that the reason is that none of the languages have been designed with sufficient care to permit such a program to be written. The necessary condition for it is the strict observance of security in all stages of the language design; and even the concept of security, let alone its importance, has hardly yet been recognised by language designers. I would suggest in future that the existence of a watertight program checking program be made the first precondition of standardisation.

A suitable yardstick for implementations of a programming language is even more difficult to construct, and is perhaps in principle impossible. This problem is not unique to programming languages; it is equally difficult to make an acid test whether a card punch or card reader is meeting the standard. The

usual maintenance technique is to tune a card punch to produce cards to within a much narrower tolerance than the standard, and to tune the card reader to accept cards with a much wider tolerance, and hope that the machinery will remain within specification until the next maintenance. For a card reader, the engineer may use for marginal testing a standard deck of badly punched (even non-standard) cards.

Unfortunately, programming language implementations are not at all like mechanical equipment; their errors are of a discrete kind, and once an implementation has been tuned to accept a particular standard set of test programs, it will always accept them, even though it fails on every other program presented to it! I fear that such a series of standard tests can only show the presence of bugs, never their absence; and as with all software (and many other products too), the only way of policing the standard is to inspect and control the methods of manufacture, i.e. the programming discipline of the implementation team.

Validation tests are certainly useful for compiler checkout; they are certainly useful for testing the quality and efficiency of a compiler; but unfortunately they can never validate its design.

6. Political climate.

A standard serves as a form of treaty or contract between groups of suppliers and customers, and imposes certain constraints on each individual supplier, which he may or may not be willing to accept; and it is only safety standards that are legally enforceable. Standards are most likely to be successful if they are agreed and adopted by groups of suppliers and customers of roughly comparable economic power, where no single supplier or customer can derive commercial benefit from deviation from the standard, or from setting up a rival standard of his own.

Unfortunately, in the field of computing standards, the political climate of agreement among equal negotiators does not apply, since one manufacturer is so much more powerful than all the others put together, and, quite justifiably, bases its standardisation policy on its own commercial interests. Thus, almost wherever we see a computer standard, either national or international (which usually agree with each other), we also see another so-called de facto standard, which is designed by one manufacturer in its own interests; and this is the standard which is most widely used; and other manufacturers often have to follow the de facto standard instead of, or even as well as, the standardised standard. But only the largest suppliers can afford this, and the largest can afford it best.

7. Conclusion.

The conclusion of this survey suggests that there are at present many factors which inhibit successful standardisation of programming languages, and which reduce the effectiveness of existing language standards. This pessimistic conclusion is, I suppose, typical of an ivory tower academic, who is suspected of wishing to promote his own abstruse theories, or even his own programming language, and is unwilling to compromise with the realities of the market place. I can only declare that these suspicions are quite groundless. I have supported all my arguments, not by any academic theory, or advanced technicality, but by means of common-sense judgement and long experience; and I hope they will be convincing to people with no experience of standardisation, and even with no knowledge or experience of programming or programming languages. I fear that the experts will be more difficult to convince.

Apart from an unfavourable political climate, I suspect that most of the problems of programming language standardisation arise from the deficiency in present-day technology in programming language design, and that in the next ten years, the study of programming methodology will provide a far more secure theoretical and practical foundation for the design and standardisation of the most important tool of the programmer's trade. And the discussions of this paper suggest some possibly fruitful avenues for research and development.

(1) There is plenty of experience of the program interchange problems with existing languages. Perhaps a systematic attempt should be made to list and classify the problems and to tackle each one methodically by reformulating the definition or modifying the language. This will be excellent practice preliminary to the design of a standard in which these faults do not occur.

(2) The attempt should be made to construct a program that will decide whether some other program conforms to one of the existing standards, or at least to point out all the places where there is risk of machine or implementation dependencies. Such a check would, of course, have to be made on the complete program, and could not be made on the independently compilable modules. And again, the difficulty of constructing the check might suggest useful reformulations to the standard **and simplifications to the language.**

(3) Further work needs to be done on the simplification and clarification of definition methods, so that they can be an actual aid to the standardisation effort, instead of an additional burden.

(4) Particular effort needs to be expended on the problem of partial specification, which permits a rigidly circumscribed freedom to an implementor, which he will need in order to secure adequate efficiency. Here it might be a good idea to start with computer arithmetic.

(5) Since a major purpose of standardisation is to transfer programmer expertise; and since a significant part of programmer expertise is to understand and act on diagnostic messages, a firm attempt should be made to standardise diagnostics. **The production of standard diagnostics should be a major function of the yardstick program.**

(6) More attention should be paid to establishing standards for the quality of an implementation - for example, its size, speed, object code efficiency, resistance and responsiveness to programming error. Standard benchmarking may play a role in this. B. Wichman has done some useful work in this area ("ALGOL 60 compilation and assessment").

(7) But the major barrier to all these preliminary enterprises, and to the standardisation itself, is the logical complexity of the languages. I have a theory that the most complex and difficult programming tasks are best carried out with very simple programming languages; and I would most strongly urge future language designers to try to test and validate this theory by practical coding experiments of large projects in **very small and simple languages.**

I believe that a thorough study of these relatively mundane topics would be far more beneficial to the progress of standardisation (and even the cost and quality of programming and compilers), than the proliferation of yet further programming language designs, many of which are incredibly complicated, and reflect far more the brilliance of their designers than any real contribution to the efficiency or reliability of programming, or even to standardisation.

So my first, last, and most important piece of advice to a proposed standardisation committee is not even to consider a language for standardisation until it satisfies the following criteria:

(1) It has been designed and has been reasonably stable for at least five years.

(2) There are at least two high quality implementations on machines with widely differing structure.

(3) These implementations have been in successful use in the intended application area for at least two years.

(4) There is adequate evidence of interchange of programs between the implementations.

(5) A timescale of at least four years should be allowed for the standardisation process. In any other field but programming languages it would be considered insane to standardise straight from the drawing board. And bitter experience of standardisation committees for ALGOL 68 (8 years) and PL/I (10 years) proves that it is somewhat unwise in this field too.

Unfortunately, there are very few languages which satisfy even these non-technical criteria, and it may be felt that their known technical deficiencies are sufficiently great as to make them a suitable basis for standardisation. But this melancholy observation, so far from being an excuse for the design of a new language, should be recognised as the strongest possible warning against it.

If the arguments against designing one new language are strong, the arguments against designing three of them are overwhelming. And the same is true of standardisation. If the only alternative to one standard is three standards, the arguments in favour of one are very strong, even if that language suffers from many known defects, both in standardisation and in use. But please let the language be a simple one which has already proved its viability and usefulness in practice, and not yet another ALGOL 68 or PL/I.