pencilled note at top of p1:
"Tony, let me know if there are any correction.
Peter [Wegner]"

EVERYTHING YOU'VE WANTED TO KNOW ABOUT PROGRAMMING LANGUAGES BUT HAVE BEEN AFRAID TO ASK

C.A.R. Hoare
November 1978

Introduction

Programming languages and their design have always excited intense controversy and intense activity. The controversy is usually conducted at a level of minute detail; and the activity has led to designs of impenetrable complexity.* Practising programmers observe that neither the detail nor the complexity seem to be relevant to the actual problems that face them daily; and each one of them believes that he could improve the language he uses, or (more rashly) could design a better one, if only his colleagues would agree.

The consequences of this are well known: a large and expensive collection of languages and incompatible dialects of each of them. It is essential to take action to bring a greater degree of order into the chaos. But how can we be sure that this action will ~~activity is~~ not merely contribute ~~contributing~~ to the chaos? Even more important, how can we convince the makers of policy and allocators of funds, who have none of the experience and technical background of those actually engaged ~~in the activity~~? How can they distinguish a policy that is wise from one that is merely clever (or worse)? How can they distinguish between those experts they can rely on and those equally distinguished experts that they can't?

This brief report attempts to answer these questions. It takes the view that the design of a programming language is not very different from the design of other engineering products - automobiles, bridges, electronic computers. Where the analogy breaks down, this is clearly highlighted, and appropriate conclusions (usually pessimistic) are drawn. By surveying previous history, the report attempts to elucidate the signs and symptoms which should have been recognized as indicative of a rapidly successful project, or of a long-drawn-out failure to meet the original hopes. It attempts to convey sufficient insight into the field to enable the non-technical policy maker to exercise genuine and most necessary control of the technical direction pursued by experts engaged on the project.

The report is structured as a series of questions and answers. Its tone is as

*See ALGOL Bulletin; or attend any language design or standardization committee meeting.

far as possible non-technical, perhaps to the point of naiveté. It appeals to anecdote and to analogy, to experience and to common sense. Every attempt has been made to achieve brevity and readability. Inevitably this means that whole areas of scientific research and development are dis~~cussed~~ missed in a brief or pungent phrase. Many readers will be alienated by the oversimplification of the issues. For this I offer apologies, but no excuse.

The report falls naturally into three parts. The first part outlines the lessons which can be learnt from study of past history; the second part tries to examine how these apply to the current DOD HOL project; and the third part looks to the future, and summarizes the research directions most relevant and promising for future language design and use.

Contents

3. What research directions are most relevant and promising for future languages?
   ~~3.1 What short-term research is most urgently required for the progress of DOD HOL?~~ stet.
   ~~3.2 What research would you recommend in the longer term?~~
   3.3 What is the relevance of problem specification languages?
   3.4 How will they influence the development of programming languages?
   ~~3.5 Will there be a place for application-oriented languages?~~
   3.5 What can we learn from controlled experiments in the use of different languages?
   3.6 What can we learn from a comparative survey of programming language features?
   3.7 What can we learn from formal methods of language definition?
   3.8 What can we learn from program verification studies?
   3.9 What are the prospects of automatic verification or even synthesis of computer programs?
   3.11 What is the best method of determining the relative merits of rival languages?
   3.~~12~~10 How will hardware development affect the relevance of existing languages and the development of new ones?

1. What can we learn from past experience of language design?

1.1 Why so many general-purpose programming language designs?

Because it always seems that no existing language is adequate. When CCITT wished to standardize a language for programming electronic switching systems (i.e. computer-controlled telephone exchanges) they made an exhaustive survey of existing languages, and came to the conclusion that none of them was adequate. The Purdue Workshop Long-Term Programming Langauge Committee in Europe (LTPL/E) made an even more exhaustive survey before coming to the conclusion that none of them was adequate. The DOD HOL project was even more thorough in its survey of criteria and languages, and came to the conclusion that none of them was adequate. I have made an exhaustive survey of committees which have made exhaustive surveys of programming languages; and I am regretfully tempted to come to the same conclusion.

1.2 Why have there been so few special-purpose languages?

Computers are used for a wider range of applications than any other mechanism; and yet they are not particularly well suited for any particular one of them. That is one of the reasons why programming them is so difficult. But a programming language is not subject to the same constraints of hardware nor the same requirement for generality. A great deal of the problem of programming could be mitigated if a language were specially designed for the needs of a particular class of user, say structural engineers, or astrophysicists, or accountants, or secretaries, or perhaps even managers!

However, this has not happened to any great extent, except perhaps in discrete event simulation. The reason is perhaps twofold:

(1) The design of a special-purpose language requires at least as much engineering skill and judgment as a general-purpose language, in design, description, and implementation.

(2) The user of a special-purpose language soon begins to miss the general algorithmic capability of a general-purpose language, and facilities for data storage and input/output. Thus even special-purpose languages (e.g. SIMULA I or SIMSCRIPT) turn out to be at least as complicated as general-purpose ones.

For these reasons, it has been more effective to superimpose special-purpose features on a simple general-purpose programming language like FORTRAN, by implementing a package of useful functions and procedures. But these packages can get quite bulky, and inefficient, and their interfaces can be complex and prone to error. SIMULA 67 was the first language to tackle this problem, but leaves some aspects of efficiency to be desired.

1.3 Were the designers of previous programming languages incompetent?

Undoubtedly, the designers of some existing languages have been technically less than in control, in that their designs have had to be quite radically modified before they could be implemented or used. The earliest drafts of the design of PL/I give evidence of this*. But the task of language design has attracted a number of the most brilliant software engineers in the world - John Backus, Peter Naur, John Reynolds, Dahl and Nygaard, John MacCarthy, Niklaus Wirth, Doug McIlroy, Edsger Dijkstra. If these great talents cannot design an adequate programming language (or quail before the magnitude of the task), this cannot possibly be attributed to technical incompetence.

1.4 Why are their language designs so inadequate? Or do they only seem so?

If we accept that the designers are competent, then perhaps the reason is that their designs are being judged by a standard that is technically unachievable. Many people cherish a hope that the adoption of a better programming language will actually make programming into an easy task. But this, I fear, is a hope as vain as the pursuit of the philosopher's stone. The design of computer programs, right

*and numerous less famous languages seem to owe more to historical accident than to conscious design!

down to the level of detail, will always be a significant intellectual endeavor, requiring the same inventiveness, insight, experience and meticulous care as the design of cathedrals, aeroplanes, bridges, and computers. It demands the balancing of many incompatible objectives: sophistication of specification, correctness, readability, adaptability, machine-independence, economy of storage, speed of execution, early delivery. The design of a program to meet the interests of its users will always depend on the relative importance of these objectives; and a good programmer almost unconsciously adapts his approach to the perceived needs of his client. No programming language will reduce the importance of the human intellect for this task; nor will it enable the programmer to get away with slipshod reasoning, inadequate planning, imprecise definitions, or inattention to detail.

This argument in no way states that all languages are equally helpful; it merely states that we should not expect too much, and maybe we should be more contented with the best designs of the past.

1.5 Why do programmers get so attached to the programming language they currently use?

The attachment of programmers to existing languages is in stark contrast to the enthusiasm of committees in condemning all such languages as inadequate; and it reflects a well-founded suspicion that no alternative language is likely to be very much better. Furthermore, a programming language conditions the whole culture and working framework of its user, to such an extent that use of a rival language becomes almost unthinkable. We are willing to tolerate major deficiencies in a familiar language, or friend; but the smallest flaw in a rival justifies our bitter hatred.

But there is another, more disturbing reason for the intense partisanship of a programmer - that his existing programming language has presented so many problems, complexities, and unexpected traps that he spent a long time learning how to master it. The prospect of a new language presents a severe threat - it might take even longer to learn to use it properly than the old one. But perhaps an even worse threat is that it might actually be <u>easier</u> to learn and use effectively, since that would wholly devalue the programmer's hard-won professional expertise in his earlier language and its implementation. In this way, a complex programming language can drive out a simple one.

1.6 Why does design of a new programming language seem so desirable and so simple?

Having determined the inadequacy of all previously designed programming languages, it is always decided that the solution is to design a new one; and this conclusion was reached independently by CCITT, LTPL/E, and the DOD HOL project. In view of the acknowledge technical competence of earlier designers, and the apathy or even the opposition of most experienced programmers, this seems an absolutely extraordinary conclusion. If it were motor cars or mixing machines, it would be quite laughable. Committees which evaluate computer hardware designs for standardization seem to realize this well enough.

But a programming language seems somehow different. It is not subject to the same sort of obvious and well-understood engineering constraints as the products of traditional engineering; and the penalties for technical inexperience are not so obvious or so dangerous. There is no danger that incorrect calculation of stresses or strengths of materials will lead to collapse. It seems so simple: all we need is to specify a range of useful features and facilities; and a team of competent compiler writers will be able to implement them. In fact no great feats of invention are called for; all that is necessary is to make a collection of the best features of existing languages, and merge them into a single language; and that is surely only a routine engineering development project.

1.7 How long does it take to complete a language project from initial design to final standardization?

On first starting the design of a new language, it is typical for the designers to work to a three-month timescale. The history of ALGOL 68 is fairly typical in this respect, but PL/I and other less well-known languages exhibit the same story. Usually the deadline has to be extended to (say) six months, after which a hurried design is produced. However, this is clearly unsatisfactory to the sponsoring body, and even to the authors, and another three months must be allowed for revision to produce the final and definitive design. Unfortunately, exactly the same history repeats itself, not once, but several times. After about two years the patience of everybody concerned can stand it no more, and the design is "finalized".

The next stage is that of implementation. But of course the degree of ~~definition~~ precision and consistency required for successful implementation is far greater than for a lan~~ug~~guage definition that satisfies a committee; and the implementors discover hosts of incredibly complicated special cases and ambiguities and inconsistencies.

The standardization committees must be reconvened, and set to work on putting the language to rights. Typically this takes about six years. It's a soul-destroying job, because it is obvious that there is no "~~right~~ best" solution to the oversights of the original design; and that the problems involved are wholly irrelevant to the improvement of the quality and efficiency of practical programming. And by this time, it is obvious that the language has not succeeded in its original objective of making programming easy!

In brief, a language design project takes about ten years from initiation to "successful" standardization: this is true of FORTRAN (1954/6-1966), ALGOL 68 (1966-1974), PL/I (1962/4-197?). PASCAL (designed c. 1970) is not yet standardized. ALGOL 60 might seem an exception to this rule: 1957-1962; but this is a language which ignored many problems, some of which are still unsolved. COBOL has undergone a lot of changes since 1960; I do not know whether it is yet sufficiently stable or well-defined to serve as a secure medium of interchange of programs between machines -- which is a primary objective of language standardization.

1.8. Why does language design always turn out to be so difficult?

Because of the extraordinarily complex and unexpected interaction effects between all parts of the language. Even an experienced language designer, complaining about some irregularity in the design of a relatively simple language like PASCAL, is astonished to find that it was a very carefully designed irregularity, required to ensure that some other part of the language should work smoothly. When a list of desired language features is first compiled, and the early designs are made, it all seems so easy, because the interaction effects are not immediately apparent. It is usually only during implementation (and after) that all the tricky and subtle problems come to light.

It is illuminating to draw a contrast with the engineering design of a motor car, for which the different "features" of the design are obviously independent, because they are physically disjoint, and can be assumed to be independent, unless they are deliberately connected. Thus we can safely separate the specification and design of the seat covers, the steering mechanism, the exhaust system, etc. These features can be safely redesigned and improved separately without fear of interaction effects (though occasionally our confidence is shattered by an unexpected result). The main reason why computer programming is more difficult (in this respect) than other branches of engineering is that every decision is capable of interacting in an unexpected way with every other decision, unless the most rigorous steps are taken to design and maintain a strict structuring discipline. But we do not yet generally recognize what kind of structuring discipline is appropriate for the design of programming languages -- though some thinkers believe that the discipline imposed by "axiomatic" definition methods could be fruitful. I also believe that some fundamental decision about storage and resource allocation strategy will be required. But any such discipline will of necessity place considerable constraints on the lists of "features and facilities" which can reasonably be inserted into the structure.

But perhaps I am being too subtle: it requires only one really foolish feature to wreck a complete design. Imagine an early automobile design standardization committee which suddenly realizes that if the chassis were fitted with wings perhaps it would fly! The ALGOL 60 "dynamic own array" was soon recognized to be like this; and so were several features of IRONMAN which were removed from STEELMAN. Perhaps the requirement for arbitrary scale factors is equally unwise.

1.9 Which were the technically successful language design projects of the past?

Here is a list:

FORTRAN, ALGOL 60, LISP, APL/BASIC, PASCAL.

The list cannot pretend to be complete; for example, I do not know enough about RPG II to pass judgment. Also, I do not wish to pass judgment on the present design of languages like COBOL, ALGOL 68, and PL/I. All I wish to suggest is that, as design projects, they were historically associated with a certain amount of technical blundering; and from a technical point of view, they do not clearly dominate their predecessors (except perhaps in size and complexity).

1.10 What are the technical reasons for their success?

The success of each language can be readily explained by the technological breakthroughs on which they are based. These are:

FORTRAN: register optimization for arithmetic expressions; and optimization of address calculation for loops. The introduction of subroutines and parameters to make the language in effect extensible.

ALGOL 60: the introduction of nestable program structures; the concepts of locality and scope of variables; arrays with dynamic bounds; recursion; and the efficient implementation of all these using a run-time ~~clock.~~ stack.

LISP: a simple uniform list structure; function definition as a uniform programming method; and the brilliant invention of the scan-mark garbage collector, which makes all of these efficiently implementable.

APL: the introduction of vectors and matrices as primitive data values; the introduction of powerful primitive vector operations to obviate the need for programmed loops; the introduction of a conversational mode of use, and a brilliantly engineered implementation. BASIC shares many of these properties.

PASCAL: user-defined types, with full type checking, including checks on references; introduction of set concept; inclusion of packed and unpacked representation; omission of many attractive features; and recognition that the need for highly efficient implementation should influence details of design. ALGOL W and PL/360 each had some of these properties.

1.11 What other factors contributed to their success?

Clearly, the most important is the existence of a ready market for the product, and the absence of effective competition. The easiest market to invade, as soon as efficiency problems are overcome, are programmers using machine/assembly code (FORTRAN, COBOL). LISP was clearly better than its competitors at just the time of the upsurge of artificial intelligence research, and benefited from a machine design (PDP 10) oriented towards its efficient implementation. APL and BASIC arrived at just the time when the oversophistication and inconvenience and inefficiency of standard manufacturers' software was making conventional batch processing very difficult to use.

PASCAL was designed initially for the educational market, which needs a language for teaching programming. Such a language must be free from unexplained complexity, simple enough for initial courses, powerful enough for second-year courses, and efficient enough for significant student projects. It coincided with a wider realization of the merits of structured programming and data structuring. And now it seems to be gaining an additional fillip as a replacement for machine code on larger microprocessors.

1.12 Why is machine and assembly code still so widely used?

I am sure that there is no single answer; and research into existing programs should attempt to quantify the relative importance of the various factors, e.g.

(1) a rather small and/or slow object machine
(2) high volume of object code produced by a compiler
(3) need for very fast response times
(4) control of highly elaborate peripheral interfaces
(5) interfacing to awkward interrupt and priority level hardware
(6) access to machine facilities not available in high-level language
(7) access to machine features deliberately hidden by the high-level language
(8) combination of (6) or (7) with the need to minimize frequency of crossing the procedure or module interface
(9) need to make patches to existing or even running programs in machine code
(10) inertia, due to presence of so much existing machine code
(11) a slow or inconvenient compiler running in an unsympathetic operating system
(12) avoidance of the awkwardness of cross compilation
(13) lack of sympathy with high-level language

I would suspect that item (13) is not nearly as widespread as enthusiasts for high-level languages would like to believe (since many of their languages do little to attack the remaining problems which lead to use of machine code). I think that every one of these reasons (except the last) would justify the use of several hundred instructions of machine code in a large system program. So any program with more than three thousand instructions of machine code may be symptomatic of the influence of reason (13).

2. The Present

2.1 How do the answers to the above questions relate to teh past and future progress of the DOD HOL project?

It seems reasonable, in the absence of evidence to the contrary, that a new project which, to date, has closely mirrored the progress of previous projects will continue to do so. The early stages of DOD HOL are disconcertingly reminiscent of the early days of ALGOL 68 or PL/I. The most difficult new feature of the language (parallelism) has been completely respecified. Each of the candidate languages has undergone two or more complete design reviews. There is still little evidence that the intense interactions between features have been fully explored. The project is still essentially at the stage of a "wishing spec", as PL/I was when it was first called NPL. It will be interesting to see how far the Red and Green candidate languages have changed since Phase 1!

So it seems safe to predict that the detailed design of the language will go through a fairly lengthy evolution during its practical implementation; and it will grow in the same complex and unpleasant way as PL/I, by the addition of voluminous and numerous footnotes, explaining to the programmer how he must avoid all the ticklish interaction effects.

But there is no doubt that this can and will be done; and that implementations will be delivered, perhaps not very late, to a number of customers, who will try to use them. They will suffer from the usual spate of unpleasant surprises at the unreliability of the system, and the inefficiency of some of its features. Gradually, they will learn to avoid the use of some of the most expensive features (e.g. parallel processing, overloaded operators, exception handling); and improved optimization techniques will take care of other problems; and there is always the possibility of liberal use of embedded or separately compiled machine code.

Then will come the task of reformulating the specification of the language to correspond with the realities of the various implementations, and the varying strategies which they have had to adopt under pressure of immediate users. And if this prediction is correct, the design will stabilize not much earlier than ten years from now.

But it seems incredible that we should be condemned to repeat all the mistakes of the past. Surely the state of the art in language design has progressed sufficiently that we can hope for slightly swifter progress. For example, DOD HOL have chosen a much more worthwhile list of design criteria than PL/I, e.g. simplicity in place of modularity, reliability in place of defaults, readability of code in place of machinability, etc. When the pressure of competitive language design is removed, perhaps the successful candidate will be free to make a drastic "cleanup" of his design, and thereby ensure early delivery of a satisfactory product which can go into immediate and widespread use. Thus perhaps we can still hope to meet Don Knuth's deadline for UTOPIA 84.

2.2 How does the DOD HOL language project compare technically with earlier language design projects?

First, it is very obviously and significantly superior to the CCITT project and the LTPL/E project, which have seriously failed to take previous experience into account. In spite of my reservations about DOD HOL, I believe that the other two projects should learn a lot from DOD HOL, or even everything!

Comparisons with ALGOL 68 and PL/I are much closer - rather too close for comfort. But DOD HOL has the advantage of building on a genuine technological breakthrough in programming language design -- the PASCAL experience. It also avoids certain shibboleths like "defaults", "orthogonality", etc.

But it shares with ALGOL 68 and PL/I an ambition to move into areas, like concurrency and exception handling, in which there is no reason to expect that the design can be got correct, and every reason to fear that the consequences of incorrect design may be serious. There is so much of immediate potential value in the project, that it is doubly unfortunate to place the whole project at risk of delay or failure through over-ambition. It may turn out as laughable as the standard automobile design, specified as sprouting wings in the hope that it might fly.

2.3 Are the novel aspects of the management of the project likely to contribute to its success?

Some of the political/managerial aspects of the DOD HOL project are very promising. The long progression of strawmen and woodenmen displayed very impressive improvement in the identification of significant objectives and issues. Not only that; the time and care expended on soliciting opinion both within DOD and from outside was a valuable and necessary educational exercise, and helped to develop a political climate in which the design could both be successful and be widely accepted. It was time well spent.

And then, perhaps somewhere between tinman and ironman, something began to go wrong. In an attempt to clarify the document, the description of the overall objectives and criteria began to get outweighed by the description of actual features required; and no attention at all was given to investigating the conflicts between the objectives and criteria, or to the relative weights and priorities which would be relevant to the resolution of these conflicts.

Even worse, no attention was paid to the really fundamental design decisions, about the amount of overhead allowable for procedure call and process change, or the underlying strategy of static, dynamic, or garbage collected storage allocation. Of course, these are difficult decisions, but they are far too important to be left to the experts. It is almost as ridiculous as asking an engineer to design a vehicle without telling him whether it is a motor bicycle, a car, or a lorry -- especially if the list of required features contains items peculiar to each one of them.

The next step was both novel and extremely sensible -- to delegate the design to expert teams with experience of the implementation and even design of programming languages. Even the method of competitive design by four separate teams has its analogue in engineering and architectural practice.

But there was again one fatal flaw in the terms of the competition -- that there was no known way of judging the true merits of the winning designs, apart from their "aesthetic" appeal. For a public building, this is very reasonable, and even for a programming language it is very important. But there are other even more important criteria like size, efficiency, speed of translation, -- even completeness and consistency of design. In architecture, such properties as ~~area covered~~ width and height are instantly obvious; and other properties such as light, thermal efficiency, and quantity of materials can be accurately assessed <u>before</u> the construction starts. Unfortunately, for programming language design (as for the design of programs), there is no sure way of doing this.

Thus the attempt to select between language designs at the end of phase 1 was premature; and unfortunately, very little extra evidence of value will be available at the end of phase 2. Meanwhile, the attempt to compete on the basis of aesthetic and political appeal, together with close conformity with the detail of IRONMAN/STEELMAN, may have had an unfortunate effect on the realism and practicality of the design.

But the most dangerous aspect of the project lies on the highly contentio$\overset{u}{\cancel{n}}$s borderline between politics, management and technology, namely

(1) Failure to distinguish those technical features which are clearly within the state of the art from those which are more speculative;

(2) Failure to give clear and realistic timescales for implementation and design and standardization.

2.4 What are the major currently unresolved language design issues?

In general, each individual issue raised by STEELMAN can be resolved (more or less) if that were the only feature to be added to a simple existing language such as PASCAL. However, clearly parallelism is a serious unresolved issue, as can be deduced from the radical changes made in that section of STEELMAN, IRONMAN, etc. I think exception handling is also a serious issue; there are some grounds for believing that a programmer who relies on exception handling to deal with exceptions will write programs less reliable and robust than those of his colleague who always tests explicitly for exceptions.

The really serious issues are those of interaction effects, of which a rather large number seem to be inherent in STEELMAN, for example.

- parallelism, recursion, and efficient real-time store management.
- generic, overloading, encapsulations, subtypes, and translation time facilities.
- parallel processes and exception handling.
- aliases, encapsulation, and independent compilation.
- checks on aliases, sharing, non-assignable record components, alteration of tag fields, together with encpasulation, independent compilation, etc. optimization
- asynchronous termination, in-process processing time clocks, explicit priority control, and minimization of execution time and space.
- approximate, exact, and integer arithmetic.
- low-level input/output, type-checking, and machine and operating system independence.

This list is not complete; those who have worked more intensively upon it could easily extend it.

I am sure that the language designers will be able to find some sort of resolution to most of these well-known conflicts. But the questions which remain are:

(1) Will the resolution meet the design criteria of generality, reliability, simplicity, efficiency, etc.? Or will it be ad hoc, infecting the rest of the language with its irregularity?

(2) Will there not be a ~~most~~ host of even more obscure and unfamiliar interaction effects, remaining to be discovered in the implementation and use of the language? /#

2.5 Which of these issues are of real significance to the practicing programmer?

Really, none of the issues is of any real significance. Each issue must, of course, be resolved; and usually it can be resolved in three or four different ways, about which a design or standardization committee can argue interminably. But the arguments never touch on the essential needs of the practicing programmer. All the solutions are equally irrelevant or unsatisfactory, giving rise to yet more problems in other apparently disconnected parts of the language. It is instructive to recall the intensity of discussion of the "issues" arising from ALGOL 68, PL/I, or even ALGOL 60. It is clear that none of them saved any money or time on behalf of the users of the language.

2.6 How can a design avoid the more irrelevant issues?

An engineer, scientist, or even mathematician often has to tackle intractable problems. But if he is wise, he spends at least as long in convincing himself that the problems are relevant and unavoidable -- and often they are. But the wise man often develops a feeling that his problems are of his own making, or arise from a mistaken apprehension of the reality of the situation. In this case he will rapidly seek methods of ensuring, not that the problem is solved, but that it does not even arise -- much to the annoyance of his clever colleagues, who are still working on a most ingenious and elaborate solution!

In the design of a computer program or programming language, the easiest and sometimes the best way of resolving a conflict between incompatible features is simply to abandon one of them, or to reduce its specification to remove the conflict. An engineer has the duty to detect when such simplifications are possible in the ultimate interests of his client. In architecture, medicine, and other branches of engineering, a client respects such professional advice and usually follows it. In programming, perhaps this is rarer: political and commercial interests will tend to overrule the advisor who claims that the problem is not worth solving, in favor of one who claims to have found a solution:- the unsatisfactory nature of the solution is never apparent until after the project is completed -- and then everybody will agree that it has been a great success! That "it should have succeeded earlier and better" is a remark that the wise engineer must discipline himself not to make.

When first making a specification it is an excellent practice to make a long list of everything that might be desired; but then it is wise to place some kind of priority ordering on the list, so that the designer has some flexibility in resolving design conflicts in the interests of his clients.

2.7 Which language design issues are central, and cannot be evaded?

The essential purpose of a programming language is to help in building a bridge between a high-level abstract (formal or informal) specification of a program and its execution in the code of some machine. Some languages only build out a short span from the machine (e.g. assembly code and PL/360). Other languages like ALGOL 60 take a more abstract approach, building a span nearer the middle of the gap. But ALGOL 60 gained much of its simplicity by ignoring a lot of machine-oriented problems like input/output, which had to be achieved by machine code inserts. Thus the span never quite reached the edge on either side!

The successful synthesis between the simple abstraction of ALGOL 60, and the relatively simple conceptual world of the machine code programmer, is the central issue of programming language design; and the attempt to mix high-level and low-level features in the same language leads to a conflict which underlies all the other conflicts of feature and of detail. Most of the complexities and irregularities of PASCAL arise from its mixture of high-level and low-level approaches; and on the whole, the great number of detailed compromises have been fairly successful.

But how can one make progress on the basis of a compromise? It is very easy to point out that the language fails to meet all of its conflicting objectives: PASCAL is too high-level to deal comfortably with (say) binary input/output; and it is too low-level to give full security checking on tagged records. (This problem can be solved, but only if an implementation makes complete alias checks of the entire program!) It seems so easy to specify that a language must have high-level <u>and</u> low-level features. But it is a very serious and central problem to weld these together into a single homogeneous framework, that will be simple enough for the majority of programmers to understand and control.

And it is very important to succeed in this; since the penalty of failure is that reliable use of the language will depend on an understanding of the high-level features as well as the low-level features, and in addition on a full mastery of the relationship between them. And this third item could be far more complicated than the other two put together, since it embraces effectively the whole compiler, the run-time system, and the underlying machine.

Of course, an unsuccessful mix of high-level and low-level features can often be mitigated by getting the most skillful programmers to write modules in machine code. The trouble is that in some applications this may comprise a large percentage of the whole program, especially if the run-time penalty for crossing module boundaries is severe.

2.8 How do the answers given above relate to the DOD HOL project?

My suggestion is that some relaxation of the STEELMAN specification as soon as possible would greatly assist the design, and permit much earlier delivery of a really useful product. All past experience shows that it is much more dangerous to clutter a language with prematurely designed features than to make a determined attempt to live without them. It is relatively easy to add a new feature to meet a well-understood need in an existing simple language; it is almost impossible to remove an over-elaborate feature from a ~~lnguge~~ language which has already gone into use! There is no need to solve all problems in one fell swoop; the attempt to do so can be injurious.

In many respects STEELMAN has already made some concessions to this attitude; at least another ten subparagraphs should be removed or relaxed -- the choice to be made by the designer/implementor.

2.9 Which aspects of STEELMAN would you recommend omitting?

This is a question that should be answered by the designer of the language. Since it is the interaction effects that are crippling, one would need to study the entire language in order to find out which features should be excised.

At this stage, it is possible only to recommend omission of those requirements which probably do not give rise to serious interaction effects, for example the requirement for exact arithmetic on fixed-point numbers of any scale. This has never been required in any previous language; and it will become even less relevant in future, as floating point hardware becomes more prevalent.

2.10 Why do academics sometimes take such a negative and even cynical attitude to the project?

Academics suffer from the same prejudices and prior commitments as other programmers; it is just that they are more articulate in expressing a justification for their views. They are also resentful when their own favorite language ideas (e.g. assignable procedure variables) have been omitted from the project. In addition, an academic is inclined to be jealous of the success of a large-scale project, in which the intellectual and scientific content must of necessity be compromised in favor of considerations of politics, finance, commerce, timescales, and management. It is extremely difficult for anyone to judge whether the compromises have gone so far as to put at risk the ultimate value and success of the project.

However, with the DOD HOL project, the serious criticisms have been the reverse of usual. Academics criticise the project for containing too many features, and even criticise it for containing features proposed by themselves and their close colleagu~~r~~es. They also criticise it for paying insufficient attention to the aspects of timescale and economics of both development and use. These criticisms should carry more weight, because they are based on a long personal experience of similar language design projects of the past.

2.11 What steps can be taken to mitigate the delays involved in the design of a new language?

If the use of a modern high-level language can be of value to existing programming projects in DOD, then any delay in its widespread introduction will be costly. Part of the cost could be avoided by encouraging immediate use of some existing language which approximates most closely in objective, spirit, and detail (to) the selected design. And by "existing" I mean a language which already has a number of implementations, good textbooks, and a wide circle of appreciative users.

The encouragement of an "intermediate" language as an interim measure will have additional long-term benefits, since early experience of ~~its~~ implementation and use of a similar language could beneficially influence the design and implementation of the new one; and its successful use could pave the way for easier acceptance of its superior successor.

Of course, this recommendation will be resisted by those who suspect that the successor language will not in fact turn out to be superior. But I regard that fear as yet another strong argument in favor of my recommendation, -- it provides a safe fall-back position, just as ALGOL W ~~could have~~ provided a safe fall-back against unexpected delays in the ALGOL 68 project.

I am _not_ suggesting universal adoption of the interim language. It should be used only on an experimental basis; but a fairly widespread experiment!

2.12 What strategy can be adopted to smooth the transition to a new language?

If my predictions are correct, the earliest implementations of a new and complex language are necessarily inefficient, unreliable, unstable, and late. A premature attempt to enforce the use of the language for a practical project is likely to ensure that the project itself will suffer all these problems as well. Many project managers will justifiably refuse to use the language again. For this reason, a transition to a new language must be planned with some care.

The suggestion that I shall make is based on the observation that the main merit of a good high-level programming language is the aid which it gives to the intellect in the design and documentation of a computer program; and the aid which it gives in actually coding the program may be secondary. (If a new language does not verify this observation, then its use should be avoided rather than encouraged!)

So my suggestion is that the language should be used on some suitable projects as a design and documentation language, which will then be encoded _by hand_ into some existing language, already implemented and familiar to the project team. Thus the language can be put to good use long before the first implementation is ready:- indeed, this would be a _very_ good way of organizing the language implementation project itself. The whole secret of what IBM calls "structured programming" is that the program is first written in a simple, clear, ALGOL-like language, before being translated (by hand) into some more complicated and obscure code like FORTRAN.

A secondary advantage of the scheme is that it is possible to evaluate the language almost immediately, and also that its implementation can be evaluated by comparing the automatically-translated version with a hand-coded version of the same program.

The disadvantage is that the project must be planned on a longer timescale: though it may then (if my prediction is correct) be completed early.

3. The Future

~~3.1 What kind of research is most urgently required for future progress in language design and use?~~

~~The most urgent question to answer by investigation is: why is there still so much machine code in use? Even programs written in a relatively low-level language like CMS2 contain large amounts of machine code. Before claiming that a new language is a solution to this problem, we ought to find out in greater detail the exact nature and extent of the problem.~~ (see 1.12)

~~3.2 What research would be relevant in the longer term for future progress in language design and use?~~

~~The most fruitful and necessary area of research is in the rigorous or even formal specification of computer programs prior to their implementation.~~

3.3 What is the relevance of problem specification languages?

A formal specification, like the blueprint of an engineering product, is the first reliable indication of feasibility, cost and quality of the eventual product (and delivery date). It also lays down a plan which makes it possible for large teams to work on separate parts of a product, with some confidence that when the parts are put together, they will work together.

An attempt could be made to design, develop, and test a language for program specification. This should be very firmly based on logic and mathematics. It should start simply as an "assertional" language, for expressing preconditions and postconditions of programs and their parts. It should contain rather a large number of high-level concepts, but no low-level concepts whatsoever. The particular proposed selection of concepts and notations must be tested by application on many kinds of program.

It is most important that no attempt be made to make the language directly implementable: this would cripple the language and with it the mental processes of its user, who must be wholly oblivious of machine constraints and machine efficiency, and consider only clarity of structure and content. However, the language should be subject to the usual syntax and type consistency checks.

When we have successfully learned how to use this language, perhaps a subset could be selected for rather inefficient implementation: this would be used to check that the specification meets the needs of the client. Methods for then obtaining an acceptably efficient program can then be applied in a systematic or even semi-automated fashion.

3.4 How will problem specification languages affect the development of programming languages?

The most important influence of a problem specification language will be to place a programming language in its proper context and perspective, as being only one of the conceptual tools available for the analysis and solution of problems. It is important that the language in which a program is actually conveyed to a computer should be designed in conformity with the other tools.

Apart from the direct benefit of the use of a very high-level specification language, it will inculcate in its user an appreciation of its logical simplicity and coherence which will lead him to demand the same qualities in his programming language.

3.5 What can we learn from controlled experiments in the use of different languages?

Controlled experiments are an excellent scientific method in dealing with phenomena which display an adequate degree of uniformity and repeatability, so that samples drawn at different times from the same population will have the same distribution. However, the technique has rarely been applied successfully to human intellectual activity, or to questions of professional expertise.

The trouble is that no~~t~~ two problems and no two programmers are the same; and the interaction effect between problem and programmer is likely to be significant (e.g., a programmer may be more or less familiar with the type of problem, or the language, or its implementation, as well as being more or less competent than his colleague). And of course even the same programmer may in time gain expertise or lose interest. The normal way of neutralizing variation of this kind is by statistical sampling. But we are interested only in rather substantial programming projects, and any realistic experiment would be prohibitively expensive:- e.g. fifty teams of programmers, each engaged in the same five-year programming project, followed by fifteen years of maintenance and enhancement. By the time the experiment is complete, its results would not be of the least relevance! For similar reasons, realistic controlled experiments are rarely used in other engineering professions, though they are used on patients in medicine and psychology.

Statistical experiments have been performed on student programmers, to determine whether some aspects of the syntax of their language could have an impact on their rate of errors -- for example, semicolon used as terminator led to fewer errors than semicolon used as separator. But student syntax error rates have no relevance whatsoever to serious professional programming; and anyway, the experiment would probably have come to exactly the opposite conclusion if the students had learnt ALGOL 60 instead of BASIC as their first programming language. And semicolons are almost wholly irrelevant to the real issues of programming.

3.6 What can we learn from a comparative survey of programming language features?

Not very much. The real merit in a programming language is its success in combining a useful (but not redundant) set of features into a single programming tool, which can be widely used and efficiently implemented. It is the right balance between features, and the avoidance of interaction effects, that is most important. Thus I fear that the exhaustive survey of languages and features conducted by the LTPL/E and CCITT were largely irrelevant, except as a learning exercise.

But there is some danger that feature comparisons can be dangerously misleading, especially if there is a feeling that "the more features the better". Features are like dials, knobs, and controls in a motor car: provided that performance and safety are not jeopardized, the <u>less</u> of these the better. A car that has a gear lever, ignition advance control, rev counter, choke and thermometer calibrated in fahrenheit and centigrade is probably less useful for the vast majority of motorists than a car that does not need these features. And a cocktail cabinet would be positively dangerous.

However, the argument cannot be taken to extremes. In the present state of the art, a car without speedometer and steering wheel would be even <u>less</u> useful. But it is a very worthwhile objective of research to design personal transport systems that do not need these features. Reduction in features would also seem to be a valid direction of research in programming languages.

3.7 What is the use of formal methods of language definition?

They are the same as those of the rigorous specification of any product prior to implementation. Without such a specification no product can be properly engineered, and will inevitably give rise to unpleasant problems of implementation and use.

As with other software and hardware products, a formal specification gives an early warning of unexpected complexities and interaction effects between various features. A wise engineer will heed this warning, and attempt rigorous simplifications, before the project gets out of his control. However, I regret that most program designers are not allowed to do that, so when the formal specifications get too complicated, they abandon the formality that should have safeguarded them, and proceed with informal specifications, which seem (at the time) much simpler. But the price that is paid later can be severe. So a formal specification method is like a fat man's diet: it is useful only if the resulting discomfort is endured.

A formal specification technique bears the same relation to language design as a language does to an application. I have a strong hope that formal specification will show the way to modular language design, in a way that permits minimization and control of the interaction effects which have plagued more informal design methods. An old example of this: the use of context-free grammar separates this important aspect of language design from all the many others. That is why the ALGOL 68 grammar is a backward step - it amalgamates concerns which could be separated.

3.8 What can we learn from program verification studies?

The most important lesson is the simple one that programs can be verified. In other words, [redacted] programming is an exact science like mathematics, and not an experimental science like chemistry, or a statistical science like sociology. This means that in principle, computer software can achieve absolute standards of reliability, far greater than those of computer hardware engineering, or any other engineering product.

But this can be achieved only if the basic tool of the programmer (his language) satisfies the same standards of rigor and exactness as logic and mathematics. The design of languages sympathetic to program verification has started, but there is still some way to go.

3.9 What are the prospects for automatic verification of computer programs?

Rather poor: they don't work well on programs that are incorrect; so the user should make sure his program is correct to start with. But then the automatic verification system is redundant. Furthermore, the effort required to get the system's assent to a program is considerable, and requires familiarity with the inner working of the system. Finally, the system's assent is obtained by feeding in additional "facts" about the problem domain; if the program has an error, there is a risk that the additional "fact" will contain the same error.

The development of automatic verification has led to important new insights in programming methodology which far transcend its value as a tool for practical programming.

3.10 How are current hardware developments likely to affect the relevance of existing languages?

The main impact of the microprocessor on the civilian market is its startlingly low cost; and its secondary impact is its limited size. This will lead to greater interest in user-oriented special-purpose languages, which are simple enough to be controlled by a non-programmer, and to be compiled and interpreted by a very low-cost computer. It will also give rise to a requirement for a language capable of programming a multi-computer system.

Low cost is not likely to make such an impact on the military market as on the civilian; partly because mil spec products will remain more expensive, and partly because computer hardware is already cheap enough compared with other components of weapons systems. However, reduction in size and power consumption will be much more ~~just as~~ significant.

The development of low-cost user-oriented languages will probably not be so significant in military application, for the same reason -- that low cost is not as important as high performance.

The requirement for programming multi-computer systems has long been recognized in military applications, at least for logistic systems. The advent of the microprocessor will reproduce the same requirement in embedded systems.

The availability of low-volume, low-power floating-point units will make the use of fixed-point arithmetic irrelevant. However, it will still be important to retain fixed-point as a "packed representation" of a floating-point number; it will be unpacked for the arithmetic operations, and repacked again after. The need to perpetuate "facilities" of existing languages may seriously delay this desirable development.

It is too early to say whether the STEELMAN specification of parallelism will help or hinder the programming of multicomputer systems. A lot will depend on the good judgment of an implementor in imposing realistic restrictions on access to non-local variables, access to processing time clock, exception handling, etc.

In the longer term, I look forward to a sufficient increase in computing power and storage capacity to permit the economic use of rather abstract high-level languages, which gain in simplicity by omitting all lower-level features altogether. But I fear that the enormous inertia inherent in the use of existing languages will ensure that any available power and space will be used to implement their irrelevant complexity. The world eagerly awaits PL/I on the 8086!

3.11 What is the best scientific method for determining the relative merits of rival programming languages?

The best method is undoubtedly an informed and expert opinion, based on a thorough study of the languages, their implementation, and their application. But who should be selected as expert? and what happens when the experts disagree? This is a problem faced by all those whose task it is to make important decisions, and the answer is: make a thorough study of your expert, his past advice and predictions, his qualifications, his personal attitudes, the clarity of his advice, etc. etc. None of these studies is guaranteed to discriminate an expert from a charlatan; but they give the best results available.

Just occasionally, a language appears that is so far ahead of its competitors that it is widely adopted in education and research, even though it is supported by no political or commercial pressure whatsoever. LISP was one of these languages; PASCAL was another. One plausible explanation for the spread of these languages is their merit. This is a historical and practical argument, not a scientific one.