PROGRAMMING LANGUAGES

PREDICTIONS AND PROSPECTS.

published 1968 at the time of IFIP congress in Edinburgh.

The safest predictions are those which assume that the future will be closely similar to the past. In the area of computing, such predictions are not quite so safe but there must come a time when they will be. For programming languages, perhaps the time has already come; so here are the predictions:

1. The languages in most widespread use in the 1970's will be ALGOL 60, and FORTRAN for technical calculations, and COBOL and PL/I for business-oriented data processing.

2. The amount of effort expended in the design and implementation of hopeful new languages will be no less in the 1970's than in the 1960's.

Why is it that new programming languages find it so difficult to make headway against the established languages? Even the sales effort of the largest manufacturer, quite successful in persuading people to adopt a new hardware design, has failed to achieve any widespread abandonment of an older programming language. The basic reason for this is that there is a strict limit on the practical benefit which can be achieved by use of a high-level programming language, and within their respective areas of application, the existing languages are already quite close to that limit. The margin for possible improvement is therefore not great enough to persuade programmers to sacrifice their existing programs and to abandon their familiar practices.

This sad fact, confirmed now by ample experience, will not be accepted by scientists designing new languages, who are searching for techniques which will actually make programming an easy task. But this is a pursuit as vain as that of the philosopher's stone. The design of computer programs will always be a significant intellectual endeavour, requiring the same inspiration, insight, experience, and meticulous care as the design of cathedrals, aeroplanes, bridges, and computers. No programming language will reduce the importance of the human intellect in this task; nor will it enable a programmer to get away with slipshod reasoning, inadequate planning, imprecise decisions, or inattention to detail.

what do we want.

There is another ultimate limitation on the design of programming languages, and this concerns the pursuit of machine-independence. There are many classes of problem for which it is possible to construct programs in a relatively machine-independent fashion; and the acceptability of the results of the program is not affected by differences in the machines on which the program is run. For example, many

scientific and engineering calculations, provided that they are numerically well-conditioned, will run on machines with widely differing floating point representations; and most of them use an integer range well within the capabilities of virtually all computers. It is this property of the problem itself which makes it possible to design and effectively use a machine-independent programming language such as ALGOL or FORTRAN. These languages have also been used successfully for programs outside the range of purely technical calculations; but it is notorious that such programs cannot be freely passed between one machine and another, because they have needed to take advantage of particular features of the implementation, such as word-length, storage layout, etc. Thus it seems that the possibility of machine-independence is much more a property of a particular application than of the language in which the program is expressed; and no language will ever cover the whole range of potential computer application in a machine-independent fashion.

This fact is well illustrated by the difficulty of transferring programs written in COBOL between machines of differing design. The successful solution of many commercial data processing problems is critically dependent on particular features of the machine on which the program is run, for example, the speed and number of magnetic tapes, the internal collating sequence of characters, the file labelling conventions, the standard file structure, etc. It is just not practical to design data processing programs which ignore these details; and the use of a high-level language can never be, in itself, a guarantee of program interchange.

It was originally hoped that the underlying character-orientation of COBOL would be the basis for interchange of programs and data; and if all machines were character-oriented, this would be an excellent solution. However, many modern machines are fundamentally word-oriented, and few actually perform arithmetic on digits expressed as characters. Thus COBOL implementations virtually have to simulate a character machine on a word machine, and this leads to heavy losses in computing power and storage capacity. For this reason, COBOL permits the programmer to specify that his data is to be stored in a manner more suited to the internal processing capability of the machine (SYNCHRONISED and COMPUTATIONAL). However, as soon as the programmer takes advantage of this, he loses machine-independence. Thus it proves possible to achieve machine-independence only for problems for which the resulting loss of efficiency is not intolerable.

The solution to this dilemma may be found in the more skilful design of a programming language, such as to permit full advantage to be taken of the characteristics of individual hardware designs, without losing machine-independence. In this prspect, the new BCL language, developed by David Hendry at the Institute of Computer Science, offers great promise. But if my theory is correct, a language which guarantees machine-independence can never be applicable to those ranges of problem which require essentially machine-dependent

solutions.

The area of application which enforces on programmers the greatest degree of machine-dependence is that of the construction of the software itself - executives, operating systems, compilers, etc. The machine-dependence here arises largely from the nature of the problem itself, and only partly from the need to achieve very high efficiencies in both running space and running time. However, the impossibility of machine-independence does not mean that the software programmer will always be barred from the other benefits of high-level language utilisation, and be forever condemned to use the inelegant and inexpressive notations of assembly code. Among the chief advantages of high-level languages is the use of nested, bracketed, and indented notations to reveal the basic structure of the algorithm to be executed. There is no reason why these notational advantages should not be grafted onto assembly language, or at least a language which is in one-to-one correspondence with assembly language. An early example of such a language is Niklaus Wirth's PL/360 (for 360-like computers) developed and used as a software-writing tool at Stanford University.

In this language the programmer retains complete control over the use of store, of registers, and has every machine code function and facility at his disposal. Furthermore, there is absolutely no overhead in the shape of run-time routines; in fact, there must not be, since the language is intended to be used for constructing run-time routines for other programming systems.

One current problem which is likely to progress towards solution in the 1970's is that of the standardisation of programming languages based on a rigorous formalisation of their syntax and semantics. At the present time, interchange of programs in supposedly machine-independent languages is inhibited by numerous minor and trivial incompatibilities, which have no explanation other than oversights on the part of the implementor, or incompleteness in the definition of the language. The Report on ALGOL 60 (ed. P. Naur and M. Woodger) was a significant advance in clarity of language definition, particularly in the syntactic area; and the 1966 drafts for a standard for FORTRAN are certainly an improvement on the previous situation. In extending techniques of language definition into the area of semantics, it is essential not to be too inflexible. The implementor must be given some freedom to adapt the language implementation to the characteristics of his machine, in areas such as integer range and floating point representation. But in other more organisational aspects such as parameter transmission, the description should allow of no ambiguity whatsoever.

Apart from its contribution to easing standardisation and program interchange, the formal definition of programming languages will be a task of mainly academic interest. It will sharpen the understanding of language designers, implementors and tutors; and it will obviate certain kinds of philosophical or metaphysical problem; but as far as practical

application is concerned, it will be as relevant as the study of the foundations of mathematics and analysis is to the use of mathematics in engineering and scientific calculations; as relevant as the geometry of Euclid is to the practice of surveying and mapmaking.

Among the most important contributors to rigorous definition of semantics are, Professor John McCarthy of Stanford University, Peter Landin of Queen Mary College, and Peter Lucas and his team at the I.B.M. Research Laboratory at Vienna.

In the area of language design itself, the 1970's will see the appearance of languages in which the possibility of coding errors has been eliminated; for example, the most frequent coding errors are:

1. Array subscript out of range
2. Use of a variable before assignment to it.
3. Mismatch of parameters on suboroutine call.

These errors can now be dealt with only by inserting highly inefficient checks at run time. The theory is that these checks will be used only during program testing, and omitted in production runs. This theory seems basically unsound. In the first place, in many of the most important application areas, program testing runs outnumber production runs by a significant factor. Secondly, it is absurd to insert checks on occasions when we are not going to rely on the results, anyway; and to omit them only when the results are going to be of importance to us. This is like wearing safety belts when the car is standing still, but throwing them off when travelling at speed.

The solution to this problem can only be in the design of languages in which coding errors are detected mainly at compile time, with only a minimum of run-time checking. This direction of progress is illustrated to some extent by ALGOL 60, and by contributions towards the development of its successor; for example, the so-called ALGOL W, implemented by Niklaus Wirth for the I.B.M. 360, and SIMULA, designed and developed by Ole-Johann Dahl and Kristen Nygaard at the Norwegian Computing Centre.

The other most promising line of development is that of the "self-extending" language, which contains a simple basic nucleus together with a means of defining language extensions in terms of the nucleus. This technique will to some extent reduce the demand for massive and complex languages, with a host of built-in features and facilities, intended to forestall every possible requirement. The ability to "make your own language" is likely to prove as important and fruitful as the facility of FORTRAN and ALGOL to "make your own subroutine". It may give rise to comprehensive families of special-purpose problem-oriented language extensions, each of them exactly aimed towards the needs of a given application area, and all of them defined in terms of a common nucleus of

procedural facilities. This will assist in solution of many of the problems at present associated with the design and use of special purpose languages, namely:

1. Restricted availability on differing machines
2. Absence of standardisation
3. Poor documentation
4. Difficulty of adaptation and extension
5. Difficulty of integration with other existing languages
6. Continuous alteration of specification on the part of the designer.

The solution of these problems is likely to lead to a major advance in the development and use of high-level problem-oriented languages in many application areas.

One of the earliest proponents of self-extending languages was John McCarthy; and their practical implementation is illustrated in Alan Perlis' FORMULA ALGOL. Recent language proposals which incorporate this feature are J. Garwick's GPL, and A. van Wijngaarden's draft design for a successor to ALGOL 60.

So these are directions which offer the brightest hope for progress in the next few years:

1. The emergence of machine-dependent high-level languages.
2. The achievement of greater machine-independence in other languages.
3. Rigorous language definition to assist in program interchange
4. The elimination of coding errors.
5. The introduction of self-extending facilities for the development of problem-oriented languages.

These developments are likely to feature in many proposals for new programming languages, but they are unlikely to find widespread application unless they can be incorporated within the framework of existing established languages. This leads to my final tip, that we shall hear a great deal more of SIMULA 67. This language has gone a long way towards machine-independence, the abolition of coding error, and towards the incorporation of self-extension features. At the same time it is based on ALGOL 60, which forms a proper subset of SIMULA. It is likely to have a great impact on the ALGOL-speaking world.

published in Computer Weekly, 1968 under the title "Limitations on languages"