Hints on programming-language design

This paper originated from Hoare's keynote address at the ACM SIGPLAN conference in Boston, October 1973 (although it did not appear in the proceedings, it was distributed at the conference). It was subsequently printed as a Stanford Artificial Intelligence Memo (AIM-224, STAN-CS-73-403) in December of that year. The version printed here was published as [43].

Hoare was active in committees working on the design and control of programming languages over many years: he made many contributions to the *ALGOL Bulletin* (e.g. [13]), was a member of IFIP's WG 2.1 (1962–74) and even chaired the ECMA TC10 group working on the standardization of PL/I. This paper is perhaps the most rounded of his published comments on language-design philosophy. The earlier [25] makes more pointed references (including a very positive one to APL); [40] and [57] make further points.

This paper must clearly be read in the context of its date of composition (1973). The relative weight of comment on debugging and reasoning about programs clearly changed as a result of his own later research. Also, a richer notion of types would be appropriate today. But the sound advice in this paper transcends any minor aspects in which it might be considered to be out of date. (Many versions exist of the story about the Mariner I Venus probe. All of them blame software; they differ as to the precise details.)

Subsequent to this publication, Hoare and Wirth consulted for SRI on their 'Yellow' language response to the 'Tinman' requirements. Their consistent advice to simplify even this language was unheeded – but the final Ada language (the 'Green' proposal) was even more baroque.

Abstract

This paper presents the view that a programming language is a tool that should assist the programmer in the most difficult aspects of his art, namely program design, documentation, and debugging. It discusses the objective criteria for evaluating a language design,

C. A. R. Hoare, Hints on programming language design. In C. Bunyan (ed.) *Computer Systems Reliability,* State of the Art Report Vol. 20, pp. 505–34. Reprinted with permission. Copyright ©1974, Pergamon/Infotech.

and illustrates them by application to language features of both high-level languages and machine-code programming. It concludes with an annotated reading list, recommended for all intending language designers.

13.1 Introduction

I would like in this paper to present a philosophy of the design and evaluation of programming languages that I have adopted and developed over a number of years, namely that the primary purpose of a programming language is to help the programmer in the practice of his art. I do not wish to deny that there are many other desirable properties of a programming language: for example, machine independence, stability of specification, use of familiar notations, a large and useful library, existing popularity, or sponsorship by a rich and powerful organization. These aspects are often dominant in the choice of a programming language by its users, but I wish to argue that they ought not to be. I shall therefore express myself strongly. I fear that each reader will find some of my points wildly controversial; I expect he will find other points that are obvious and even boring; I hope that he will find a few points that are new and worth pursuing.

My approach is first to isolate the most difficult aspects of the programmer's task, and state in general terms how a programming-language design can assist in meeting these difficulties. I discuss a number of goals, which have been followed in the past by language designers, and which I regard as comparatively irrelevant or even illusory. I then turn to particular aspects of familiar high-level programming languages, and explain why they are in some respects much better than machine-code programming, and in certain cases worse. Finally, I draw a distinction between language-feature design and the design of complete languages.

13.2 Principles

If a programming language is regarded as a tool to aid the programmer, it should give him the greatest assistance in the most difficult aspects of his art, namely program design, documentation, and debugging.

(1) *Program design.* The first, and very difficult, aspect of design is deciding what the program is to do, and formulating this as a clear, precise, and acceptable specification. Often just as difficult is deciding how to do it: how to divide a complex task into simpler subtasks, and to specify the

purpose of each part, and define clear, precise, and efficient interfaces between them. A good programming language should give assistance in expressing not only how the program is to run, but what it is intended to accomplish; and it should enable this to be expressed at various levels, from the overall strategy to the details of coding and data representation. It should assist in establishing and enforcing conventions and disciplines that will ensure harmonious co-operation of the parts of a large program when they are developed separately and finally assembled together.

(2) *Programming documentation.* The purpose of program documentation is to explain to a human reader the way in which a program works so that it can be successfully adapted after it goes into service, to meet the changing requirements of its users, or to improve it in the light of increased knowledge, or just to remove latent errors and oversights. The view that documentation is something that is added to a program after it has been commissioned seems to be wrong in principle and counter-productive in practice. Instead, documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear self-documenting code, and even perhaps to develop and display a pleasant style of writing. The readability of programs is immeasurably more important than their writeability.

(3) *Program debugging.* Program debugging can often be the most tiresome, expensive, and unpredictable phase of program development, particularly at the stage of assembling subprograms written by many programmers over a long period. The best way to reduce these problems is by successful initial design of the program, and by careful documentation during the construction of code. But even the best-designed and best-documented programs will contain errors and inadequacies, which the computer itself can help to eliminate. A good programming language will give maximum assistance in this. Firstly, the notations should be designed to reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler, before the program even begins to run. Certain programming errors cannot always be detected in this way, and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine- or implementation-dependent effects, which are inexplicable in terms of the language itself. This is a criterion to which I give the name *security*. Of course, the compiler itself must be utterly reliable, so that its user has complete confidence that any unexpected effect was occasioned by his own program. And the compiler must be compact and fast, so that there is no appreciable delay or cost involved in correcting a program in source code and resubmitting for another run; and the object code too should be fast and efficient, so that extra instructions can be inserted even in large and time-consuming programs in order to help detect their errors or inefficiencies.

A necessary condition for the achievement of any of these objectives is the utmost simplicity in the design of the language. Without simplicity, even the language designer himself cannot evaluate the consequences of his design decisions. Without simplicity, the compiler writer cannot achieve even reliability, and certainly cannot construct compact, fast, and efficient compilers. But the main beneficiary of simplicity is the user of the language. In all spheres of human intellectual and practical activity, from carpentry to golf, from sculpture to space travel, the true craftsman is the one who thoroughly understands his tools. And this applies to programmers too. A programmer who fully understands his language can tackle more complex tasks, and complete them more quickly and more satisfactorily, than if he did not. In fact, a programmer's need for an understanding of his language is so great that it is almost impossible to persuade him to change to a new one. No matter what the deficiencies of his current language, he has learned to live with them; he has learned how to mitigate their effects by discipline and documentation, and even to take advantage of them in ways that would be impossible in a new and cleaner language which avoids the deficiency.

It therefore seems especially necessary in the design of a new programming language, intended to attract programmers away from their current high-level language, to pursue the goal of simplicity to an extreme, so that a programmer can readily learn and remember all its features, can select the best facility for each of his purposes, can fully understand the effects and consequences of each decision, and can then concentrate the major part of his intellectual effort on understanding his problem and his programs rather than his tool.

A high standard of simplicity is set by machine or assembly code programming for a small computer. Such a machine has an extremely uniform structure, for example a main store consisting of 2^m words numbered consecutively from zero up, a few registers, and a simple synchronous standard interface for communication with and control of peripheral equipment. There is a small range of instructions, each of which has a uniform format; and the effect of each instruction is simple, affecting at most one register and one location of store or one peripheral. Even more important, this effect can be described and understood quite independently of every other instruction in the repertoire. And finally, the programmer has an immediate feedback on the compactness and efficiency of his code. Enthusiasts for high-level languages are often surprised at the complexity of the problems that have been tackled with such simple tools.

On larger modern computers, with complex instruction repertoires and even more complex operating systems, it is especially desirable that a high-level language design should aim at the simplicity and clear modular description of the best hardware designs. But the only widely used languages that approach this ideal are FORTRAN, LISP, and ALGOL 60, and a few languages developed from them. I fear that most more modern programming languages are getting even more complicated; and it is particularly irritating when their proponents claim that future hardware designs should be oriented towards the implementation of this complexity.

13.3 Discussion

The previous two sections have argued that the objective criteria for good language design may be summarized in five catch phrases: simplicity, security, fast translation, efficient object code, and readability. However desirable these may seem, many language designers have adopted alternative principles that belittle the importance of some or all of these criteria, perhaps those that their own languages have failed to achieve.

13.3.1 Simplicity

Some language designers have replaced the objective of simplicity by that of modularity, by which they mean that a programmer who cannot understand the whole of his language can get by with a limited understanding of only part of it. For programs that work as the programmer intended this may be feasible; but if his program does not work, and accidentally invokes some feature of the language that he does not know, he will get into serious trouble. If he is lucky the implementation will detect his mistake, but he will not be able to understand the diagnostic message. Otherwise, he is even more helpless. If to the complexity of his language is added the complexity of its implementation, the complexity of its operating environment, and even the complexity of institutional standards for the use of the language, it is not surprising that when faced with a complex programming task so many programmers are overwhelmed.

Another replacement of simplicity as an objective has been orthogonality of design. An example of orthogonality is the provision of complex integers, on the argument that we need reals and integers and complex reals, so why not complex integers? In the early days of hardware design, some very ingenious but arbitrary features turned up in order codes as a result of orthogonal combinations of the function bits of an instruction, on the grounds that some clever programmer would find a use for them; and some clever programmer always did. Hardware designers have now learned more sense; but language designers are clever programmers and have not.

The principles of modularity, or orthogonality, insofar as they contribute to overall simplicity, are an excellent means to an end; but as a substitute for simplicity they are very questionable. Since in practice they have proved to be a technically more difficult achievement than simplicity, it is foolish to adopt them as primary objectives.

13.3.2 Security

The objective of security has also been widely ignored; it is believed instead that coding errors should be removed by the programmer with the assistance of a so-called *checkout* compiler. But this approach has several practical disadvantages. For example, the checkout compiler and the standard compiler are often not equally reliable. Even if they are, it is impossible to guarantee that they will give the same results, especially on a subtly incorrect program; and, when they do not, there is nothing to help the programmer find the mistake. For a large and complex program the extra inefficiency of the debugging runs may be serious; and even on small programs the cost of loading a large debugging system can be high. You should always pity the fate of the programmer whose task is so difficult that his program will not fit into the computer together with your sophisticated debugging package. Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea? Fortunately, with a secure language, the security is equally tight for production and for debugging.

13.3.3 Fast translation

In the early days of high-level languages it was openly stated that speed of compilation was of minor importance, because programs would be compiled only once and then executed many times. After a while it was realized that the reverse was often true, that a program would be compiled frequently while it was being debugged; but instead of constructing a fast translator, language designers turned to independent compilation, which permits a programmer to avoid recompiling those parts of his program that he has not changed since the last time. But this is a poor substitute for fast compilation, and has many practical disadvantages. Often it encourages or even forces a programmer to split a large program into modules that are too small to express properly the structure of his problem. It entails the use of wide interfaces and cumbersome and expensive parameter lists at inappropriate places. And even worse, it prevents the compiler from adequately checking the validity of these interfaces. It requires additional file space to store bulky intermediate code, in addition to source code, which must, of course, never be thrown away. It discourages the programmer from making changes to his data structure or representation, since this would involve a heavy burden of recompilation. And finally the linkage editor is often cumbersome to invoke and expensive to execute. And it is all so unnecessary, if the compiler for a good language can work faster than the linkage editor anyway.

If you want to make a fast compiler even faster, I can suggest three techniques, which have all the benefits of independent compilation and none of the disadvantages.

(1) *Prescan.* The slowest part of a modern fast compiler is the lexical scan, which inputs individual characters, assembles them into words or numbers, identifies basic symbols, removes spaces, and separates the comments. If the source text of the program can be stored in a compact form in which this character handling does not have to be repeated, compilation time may be halved, with the added advantage that the original source program may still be listed (with suitably elegant indentation); and so the amount of file storage is reduced by a factor considerably greater than two. A similar technique was used by the PACT I assembler for the IBM 701.

(2) *Precompile.* This is a directive that can be given to the compiler after submitting *any* initial segment of a large program. It causes the compiler to make a complete dump of its workspace, including dictionary and object code, in a specified user file. When the user wishes to add to his program and run it, he directs the compiler to recover the dump and proceed. When his additions are adequately tested, a further precompile instruction can be given. If the programmer needs to modify a precompiled procedure, he can just redeclare it in the block containing his main program, and normal ALGOL-like scope rules will do the rest. An occasional complete recompilation will consolidate the changes after they have been fully tested. The technique of precompilation is effective only on single-pass compilers; it was successfully incorporated in the Elliott ALGOL programming system.

(3) *Dump.* This is an instruction that can be called by the user program during execution, and causes a complete binary dump of its code and workspace into a named user file. The dump can be restored and restarted at the instruction following the dump by an instruction to the operating system. If all necessary data input and initialization is carried out before the dump, the time spent on this as well as recompilation time can be saved. This provides a simple and effective way of achieving the FORTRAN effect of block data, and was successfully incorporated in the implementation of Elliott ALGOL.

The one remaining use of independent compilation is to link a high-level language with machine code. But even here independent compilation is the wrong technique, involving all the inefficiency of procedure call and all the complexity of parameter access at just the point where it hurts most. A far better solution is to allow machine code instructions to be inserted in-line within a high-level language program, as was done in Elliott ALGOL; or, better, provide a macro facility for machine code, as in PL/360.

Independent compilation is a solution to yesterday's problems; today it has grown into a problem in its own right. The wise designer will prefer to avoid rather than solve such problems.

13.3.4 Efficient object code

There is another argument, which is all too prevalent among enthusiastic language designers, that efficiency of object code is no longer important; that the speed and capacity of computers is increasing and their price is coming down; and that the programming-language designer might as well take advantage of this. This is an argument that would be quite acceptable if used to justify an efficiency loss of ten or twenty percent, or even thirty and forty percent. But all too frequently it is used to justify an efficiency loss of a factor of two, or ten, or even more; and worse, the overhead is not only in time taken but in space occupied by the running program. In no other engineering discipline would such avoidable overhead be tolerated, and it should not be in programming-language design, for the following reasons:

(1) The magnitude of the tasks we wish computers to perform is growing faster than the cost-effectiveness of the hardware.
(2) However cheap and fast a computer is, it will be cheaper and faster to use it more efficiently.
(3) In the future we must hope that hardware designers will pay increasing attention to reliability rather than to speed and cost.
(4) The speed, cost, and reliability of peripheral equipment are not improving at the same rate as those of processors.
(5) If anyone is to be allowed to introduce inefficiency it should be the user programmer, not the language designer. The user programmer can take advantage of this freedom to write better-structured and clearer programs, and should not have to expend extra effort to obscure the structure and write less clear programs just to regain the efficiency that has been so arrogantly pre-empted by the language designer.

There is a widespread myth that a language designer can afford to ignore machine efficiency, because it can be regained when required by the use of a sophisticated optimizing compiler. This is false: there is nothing that the good engineer can afford to ignore. The only language that has been optimized with general success is FORTRAN, which was very specifically designed for that very purpose. But even in FORTRAN, optimization has grave disadvantages:

(1) An optimizing compiler is usually large, slow, unreliable, and late.
(2) Even with a reliable compiler, there is no guarantee than an optimized program will have the same results as a normally compiled one.
(3) A small change to an optimized program may switch off optimization with an upredictable and unacceptable loss of efficiency.
(4) The most subtle danger is that optimization tends to remove from the programmer his fundamental control over and responsibility for the quality of his programs.

The solution to these problems is to produce a language for which a simple straightforward 'non-pessimizing' compiler will produce straightforward object programs of acceptable compactness and efficiency: similar to those produced by a resolutely non-clever (but also non-stupid) machine-code programmer. Make sure that the language is sufficiently expressive to enable most other optimizations to be made in the language itself; and finally, make the language so simple, clear, regular, and free from side-effects that a general machine-independent optimizer can simply translate an inefficient program into a more efficient one with guaranteed identical effects, and expressed in the same source language. The fact that the user can inspect the results of optimization in his own language mitigates many of the defects listed above.

13.3.5 Readability

The objective of readability by human beings has sometimes been denied in favour of readability by a machine; and sometimes even been denied in favour of abbreviation of writing, achieved by a wealth of default conventions and implicit assumptions. It is of course possible for a compiler or service program to expand the abbreviations, fill in the defaults, and make explicit the assumptions. But, in practice, experience shows that it is very unlikely that the output of a computer will ever be more readable than its input, except in such trivial but important aspects as improved indentation. Since in principle programs should be read by others, or re-read by their authors, *before* being submitted to the computer, it would be wise for the programming language designer to concentrate on the easier task of designing a readable language to begin with.

13.4 Comment conventions

If the purpose of a programming language is to assist in the documentation of programs, the design of a superb comment convention is obviously our most important concern. In low-level programming, the greater part of the space on each line is devoted to comment. A comment is always terminated by an end-of-line, and starts either in a fixed column or with a special symbol allocated for this purpose:

LDA X [THIS IS A COMMENT

The introduction of free format into high-level languages prevents the use of the former method; but it is surprising that few languages have adopted the latter. ALGOL 60 has two comment conventions. One is to enclose the text of a comment between the basic word **comment** and a semicolon:

comment this is a comment;

This has several disadvantages over the low-level convention:

(1) The basic word **comment** is too long. It occupies space that would be better occupied by the text of the comment, and is particularly discouraging to short comments.
(2) The comment can appear only after a **begin** or a semicolon, although it would sometimes be more relevant elsewhere.
(3) If the semicolon at the end is accidentally omitted, the compiler will without warning ignore the next following statement.
(4) One cannot put program text within a comment, since a comment must not contain a semicolon.

The second comment convention of ALGOL 60 permits a comment between an **end** and the next following semicolon, **end**, or **else**. This has proved most unfortunate, since omission of a semicolon has frequently led to the compiler ignoring the next following statement:

... **end** this is a mistake $A[i] := x$;

The FORTRAN comment convention defines as comment the whole of a line containing a C in the first column.

C THIS IS A COMMENT

Its main disadvantages are that is does not permit comments on the same line as the code to which they refer, and that it discourages the use of short comments. An unfortunate consequence is that a well-annotated FORTRAN program occupies many pages, even though the greater part of each page is blank. This in itself makes the program unnecessarily difficult to read and understand.

The comment convention of COBOL suffers from the same disadvantages as that of FORTRAN, since it insists that commentary should be in a separate paragraph.

More recently designed languages have introduced special bracketing symbols (e.g./* and */) to enclose comments, which can therefore be placed anywhere in the program text where they are relevant:

/* THIS IS A COMMENT */

But there still remains the awkward problem of omitting or mispunching one of the comment brackets. In some languages, this will cause omission of statements between two comments; in others it may cause the whole of the rest of the program to be ignored. Neither of these disasters are likely to occur in low-level programs, where the end-of-line terminates a comment.

13.5 Syntax

Another aspect of programming-language design that is often considered trivial or arbitrary is its syntax. But this is also a mistake: the designer should select and observe the best possible syntactic framework for his language, for two important practical reasons:

(1) In a modern fast compiler a significant time can be taken in the assembly of characters into meaningful symbols (identifiers, numbers, and basic words) and in checking the context-free structure of the program.

(2) When a program contains a syntactic error it is important that the compiler should be able to pinpoint the error accurately, to diagnose its cause, recover from it, and continue checking the rest of the program. Recall the first American space probe to Venus, reportedly lost because FORTRAN cannot recognize a missing comma in a *DO* statement. In FORTRAN the statement:

$$DO\ 17\ I = 1\ 10$$

looks to the compiler like an assignment to a (probably undeclared) variable *DO17I:*

$$DO17I = 110.$$

In low-level programming, the use of fixed field format neatly solves both problems. The position and length of each meaningful symbol is known, and it can be copied and compared as a whole without even examining the individual characters; and if one field contains an error it can be immediately pinpointed, and checking can be resumed at the very next field.

Fortunately, free-format techniques have been discovered that solve the problems nearly as neatly as fixed format. The use of a finite-state machine to define the assembly of characters into symbols, and one of the more restrictive forms of context-free grammars (e.g. precedence or topdown or both) to define the structure of a program: these must be recommended to every language designer. It is certainly possible for a machine to analyse more complex grammars, but there is every indication that the human programmer will find greater difficulty, particularly if an error is present or even only suspected. If a compiler cannot diagnose the syntax of an individual statement until it reaches the end of the program, what hope has a poor human?

As an example of what happens when a language departs from the best-known technology, that of context-free syntax, consider the case of the labelled END. This is a convention whereby any identifier between an END and its semicolon automatically signals the end of the procedure with that name, and of any enclosed program structure even if it has no END of its own. At first sight this is a harmless notational convenience, which Peter Landin might call 'syntactic sugar'; but in practice the consequences are disastrous. If the programmer accidentally omits an END *anywhere* in his program, it will automatically and without warning be inserted just before the next following labelled END, which is very unlikely to be where it was wanted. Landin's phrase for this would be 'syntactic rat poison'. Wise programmers have therefore learned to avoid the labelled END, which is a great pity, since if the labelled END had been used merely to *check* the correctness of the nesting of statements it would have been very useful, and permitted earlier and cleaner error recovery, as well as remaining within the disciplines of context-free languages. Here is a classic example of a language feature that combines danger to the programmer with difficulty for the implementor. It is all too easy to reconcile criteria of demerit.

13.6 Arithmetic expressions

A major feature of FORTRAN, which gives it the name FORmula TRANslator, is the introduction of the arithmetic expression. ALGOL 60 extends this idea by the introduction of a conditional expression. Why is this such an advance over assembly code? The traditional answer is that it appeals to the programmer's familiarity with mathematical notation. But this only leads to the more fundamental question, why is the notation of arithmetic expressions of such benefit to the mathematician? The reason seems to be quite subtle and fundamental. It embodies the principles of structuring, which underlie all our attempts to master a complex problem or control a complex situation by analysing it into simpler subproblems, with clean and narrow interfaces between them.

Consider an arithmetic expression of the form

$$E + F$$

where E and F may themselves be simple or complex arithmetic expressions.

(1) The meaning of this whole expression can be understood wholly in terms of an understanding of the meanings of E and F;
(2) the purpose of each part consists solely in its contribution to the purpose of the whole;
(3) the meaning of the two parts can be understood wholly indepen-dendently of each other;
(4) if E or F is itself an arithmetic expression, the same structuring principle can be applied to the analysis of the parts as is applied to the understanding of the whole;
(5) the interface between the parts is clear, narrow, and well controlled: in this case just a single number; and, finally,
(6) the separation of the parts and their relation to the whole is clearly apparent from their written form.

These seem to be six fundamental principles of structuring: transparency of meaning and purpose; independence of parts; recursive application; narrow interfaces; and manifestness of structure. In the case of arithmetic expressions these six principles are reconciled and achieved together with very high efficiency of implementation. But the applicability of the arithmetic expression is seriously limited by the extreme narrowness of the interface. Often the programmer wishes to deal with much larger data structures, for example vectors or matrices or lists; and languages such as APL and LISP have permitted the use of expressions with these structures as operands and results. This seems to be an excellent direction of advance in programming-language design, particularly for special-purpose languages. But the advance is not purchased without some penalty in efficiency and programmer control. The very reason why arithmetic expressions can be evaluated with such efficiency is that the operands and results of each subexpression are sufficiently small to be held in a high-speed register, or stored and recovered from a main-store location by a single instruction. When the operands are too large, and especially when they might be partially or wholly stored on backing store, it becomes much more efficient to use updating operations, since then the space occupied by one of the operands can be used to hold the result. It would therefore seem advisable to introduce special notations into a language to denote such operations as adding one matrix to another, appending one list to another, or making a new entry in a file; for example:

$A. + B$	instead of $A := A + B$ if and A and B are matrices
L1.append(L2)	if *L1* and *L2* are lists.
F. output(x)	if *F* is a file.

Another efficiency problem that arises from the attempt of a language to provide large data structures and built-in operations on them is that the implementation must select a particular machine representation for the data, and use it uniformly, even in cases where other representations might be considerably more efficient. For example, the APL representation is fine for small matrices, but is very inappropriate or even impossible for large and sparse ones. The LISP representation of lists is very efficient for data held wholly in main store, but becomes inefficient when the lists are so long that they must be held on backing store, particularly disks and tapes. Often the efficiency of a representation depends on the relative frequency of various forms of operation, and therefore the representation should be different in different programs, or even be changed from one phase of a program to another.

A solution to this problem is to design a general-purpose language that provides the programmer with the tools to design and implement his own representation for data and code the operations upon it. This is the main justification for the design of 'extensible' languages, which so many designers have aimed at, with rather great lack of success. In order to succeed, it will be necessary to recognize the following:

(1) The need for an exceptionally efficient base language in order to define the extensions.
(2) The avoidance of any form of syntactic extension to the language. All that is needed is to extend the meaning of the existing operators of the language, an idea that was called *overloading* by McCarthy.
(3) The complete avoidance of any form of automatic type transfer, coercion, or default convention, other than those implemented as an extension by the programmer himself.

I fear that most designers of extensible languages have spurned the technical simplifications that make them feasible.

13.7 Program structures

However far the use of expressions and functional notations may be extended, a programmer will eventually require the capability of updating his environment. Sometimes this will be because he wants to perform input and output, sometimes because it is more efficient to store the results of a computation so that the stored value can be used rather than recomputed at a later time, and sometimes because it is a natural way of representing his problem (for example, in the case of discrete-event simulation or the monitoring and control of some real-world process).

Thus it is necessary to depart from the welcome simplicity of the mathematical expression, but to attempt to preserve as far as possible the structuring principles that it embodies. Fortunately, ALGOL 60 (in its compound, conditional, **for**, and **procedure** statements) has shown the way in which this can be done. The advantages of the use of these program structures is becoming apparent even to programmers using languages that do not provide the notations to express them.

The introduction of program structures into a language not only helps the programmer but also does not injure the efficiency of an implementation. Indeed, the avoidance of wild jumping will be of positive benefit on machines with slave stores or paging hardware; and if a compiler makes any attempt at optimization, the clear indication of the control structure of a program can only simplify this task.

There is one case where ALGOL 60 does not provide an appropriate structure, and that is when a selection must be made from more than two alternatives in accordance with some integer value. In this case, the programmer must declare a switch, specifying a list of labels, and then jump to the *i*th label in this list.

```
       switch SS:= L1, L2, L3;
         ⋮
       go to SS[i];
  L1:  Q1; go to L;
  L2:  Q2; go to L;
  L3:  Q3;
   L:
```

Unfortunately the introduction of the switch as a nameable entity is not only an extra complexity in the language and implementation but also gives plenty of scope for tricky programming and even trickier errors, particularly when jumping to some common continuation point on completion of the alternative action.

The first language designers to deal with the problem of the switch proposed to generalize it by providing the concept of the label array, into which the programmer could store label values. This has some peculiarly unpleasant consequences in addition to the disadvantages of the switch. Firstly, it obscures the program, so that its control structure is not apparent from the form of the program but can only be determined by a run-time trace. And, secondly, the programmer is given the power to jump back into the middle of a block he has already exited from, with unpredictable consequences unless a run-time check is inserted. In ALGOL 60 the scope rules make this error detectable at compile time.

The way to avoid all these problems is a very simple extension to the ALGOL 60 conditional notation, a construction that I have called the **case** construction. In this notation, the example of the switch shown above would take the form:

```
case i of
{Q1,
 Q2,
 Q3};
```

This was my first programming-language invention, of which I am still most proud, since it appears to bear no trace of compensating disadvantage.

13.8 Variables

One of the most powerful and most dangerous aspects of machine-code programming is that each individual instruction of the code can change the content of any register or store location and alter the condition of any peripheral: it can even change its neighbouring instructions or itself. Worse still, the identity of the location changed is not always apparent from the written form of the instruction; it cannot be determined until run time, when the values of base registers, index registers, and indirect addresses are known. This does not matter if the program is correct, but if there is the slightest error, even only in a single bit, there is no limit to the damage that may be done, and no limit to the difficulty of tracing the cause of the damage. In summary, the interface between every two consecutive instructions in a machine-code program consists of the state of the entire machine: registers, main store, backing stores, and all peripheral equipment.

In a high-level language, the programmer is deprived of the dangerous power to update his own program while it is running. Even more valuable, he has the power to split his machine into a number of separate variables, arrays, files, etc.; when he wishes to update any of these he must quote its name explicitly on the left of the assignment, so that the identity of the part of the machine subject to change is immediately apparent; and, finally, a high-level language can guarantee that all variables are disjoint, and that updating any one of them cannot possibly have any effect on any other.

Unfortunately, many of these advantages are not maintained in the design of procedures and parameters in ALGOL 60 and other languages. But instead of mending these minor faults, many language designers have preferred to extend them throughout the whole language by introducing the concept of reference, pointer, or indirect address into the language as an assignable item of data. This immediately gives rise in a high-level language to one of the most notorious confusions of machine code, namely that between an address and its contents. Some languages attempt to solve this by even more confusing automatic coercion rules. Worst still, an indirect assignment through a pointer, just as in machine code, can update any store location whatsoever, and the damage is no longer confined to the variable explicitly named as the target of assignment. For example, in ALGOL 68, the assignment:

$$x := y$$

always changes x, but the assignment:

$$x := y + 1;$$

may, if x is a reference variable, change any other variable (of appropriate type) in the whole machine. One variable it can *never* change is x! Unlike all other values (integers, strings, arrays, files, etc.) references have no meaning independent of a particular run of a program. They cannot be input as data, and they cannot be output as results. If either data or references to data have to be stored on files or backing stores, the problems are immense. And on many machines they have a surprising overhead on performance, for example they will clog up instruction pipelines, data lookahead, slave stores, and even paging systems. References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover.

13.9 Block structure

In addition to the advantages of disjoint named variables, high-level languages provide the programmer with a powerful tool for achieving even greater security, namely the scope and locality associated with block structure. In FORTRAN or ALGOL 60, if the programmer needs a variable for the purposes of a particular part of his program, he can declare it locally to that part of the program. This enables the programmer to make manifest in the structure of his program the close association between the variable and the code that uses it; and he can be absolutely confident that no other part of the program, whether written by himself or another, can ever interfere with, or even look at, the variable without his written permission, i.e. unless he passes it as a parameter to a particular named procedure. The use of locality also greatly reduces the width of the interfaces between parts of the program; the fact that programmers no longer need to tell each other the names of their working variables is only one of the beneficial consequences.

Like all the best programming-language features, the locality and scope rules of ALGOL 60 are not only of great assistance to the programmer in the decomposition of his task and the implementation of its subtasks; they also permit economy in the use of machine resources, for example main store. The fact that a group of variables is required for purposes local only to part of a program means that their values will usually be relevant only while that part of the program is being executed. It is therefore possible to re-allocate to other purposes the storage assigned to these variables as soon as they are no longer required. Since the blocks of a program in ALGOL 60 are always completed in the exact reverse of the order in which they were entered, the dynamic re-allocation of storage can be accomplished by stack techniques, with small overhead of time and space, or none at all in the case of blocks that are not procedure bodies, for which the administration can be done at compile time. Finally, the programmer is encouraged to declare at the same time those variables that will be used together, and these will be allocated in contiguous locations, which will increase the efficiency of slave storage and paging techniques.

It is worthy of note that the economy of dynamic reallocation is achieved without any risk that the programmer will accidentally refer to a variable that has been re-allocated, and this is guaranteed by a compile-time and not a run-time check. All these advantages are achieved in ALGOL 60 by the close correspondence between the statically visible scope of a variable in a source program and the dynamic lifetime of its storage when the program is run. A language designer should therefore be extremely reluctant to break this correspondence, which can easily be done, for example, by the introduction of references, which may point to variables of an exited block. The rules of ALGOL 68, designed to detect such so-called 'dangling references' at compile time, are both complicated and ineffective; and PL/I does not bother at all.

13.10 Procedures and parameters

According to current theories of structured programming, every large-scale programming project involves the design, use, and implementation of a special-purpose programming language, with its own data concepts and primitive operations, specifically oriented to that particular project. The procedure and parameter are the major tools provided for this purpose by high-level languages since FORTRAN. In itself, this affords all the major advantages claimed for extensible languages. Furthermore, in its implementation as a closed subroutine, the procedure can achieve very great economies of storage at run time. For these reasons, the language designer should give the greatest attention to this feature of this language. Procedure calls and parameter passing should produce very compact code. Lengthy preludes and postludes must be avoided. The effect of the procedure on its parameters should be clearly manifest from its syntactic form, and should be simple to understand and resistant to error. And, finally, since the procedure interface is so often the interface between major parts of a program, the correctness of its use should be subjected to the most rigorous compile-time check.

The chief defects of the FORTRAN parameter mechanism are:

(1) It fails to give a notational distinction at the call side between parameters that convey values into a procedure, those that convey values out of a procedure, and those that do both. This negates many of the advantages that the assignment statement has over machine-code programming.
(2) The shibboleth of the independent compilation prohibits compile-time checks on parameter passing, just where interface errors are most likely, most disastrous, and most difficult to debug.
(3) The ability to define side-effects of function calls negates many of the advantages of arithmetic expressions.

At least FORTRAN permits efficient implementation, unless a misguided but all too frequent attempt is made to permit a mixture of languages across the procedure interface. A subroutine that does not know whether it is being called from ALGOL or from FORTRAN has a hard life.

ALGOL 60 perpetuates all these disadvantages, but not the advantage. The difficulty of compile-time parameter checking is due to the absence of parameter specifications. Even if an implementation insists on full specification (and most do) the programmer has no way of specifying the parameters of a formal procedure parameter. This is one of the excuses for the inefficiency of many ALGOL implementations. The one great advance of ALGOL 60 is the value parameter, which is immeasurably superior to the dummy parameter of FORTRAN and PL/I. What a shame that the name parameter is the default!

But perhaps the most subtle defect of the ALGOL 60 parameter is that the user is permitted to pass the same variable twice as an actual parameter corresponding to two distinct formal parameters. This immediately violates the principles of disjointness, and can lead to many curious, unexpected effects. For example, if a procedure:

$$\textit{matrix multiply}\ (A, B, C)$$

is intended to have the effect:

$$A := B \times C$$

it would seem reasonable to square A by:

$$\textit{matrix multiply}\ (A, A, A)$$

This error is prohibited in standard FORTRAN, but few programmers realize it, and it is rarely enforced by a compile-time or run-time check. No wonder the procedure interface is the one on which run-time debugging aids have to concentrate.

13.11 Types

Among the most trivial but tiresome errors of low-level programming are type errors, for example using a fixed-point operation to add floating-point numbers, using an address as an integer or vice versa, or forgetting the position of a field in a data structure. The effects of such errors, although fully explicable in terms of bit patterns and machine operations, are so totally unrelated to the concept in terms of which the programmer is thinking that the detection and correction of such errors can be exceptionally tedious. The trouble is that the hardware of the computer is far too tolerant and forgiving. It is willing to accept almost any sequence of instructions and make sense of them at its own level. That is the secret of the power, flexibility, and simplicity, and even reliability, of computer hardware, and should therefore be cherished.

But it is also one of the main reasons why we turn to high-level languages, which can eliminate the risk of such error by a compile-time check. The programmer declares the type of each variable, and the compiler can work out the type of each result; it therefore always knows what type of machine-code instruction to generate. In cases where there is no meaningful operation (for example, the addition of an integer and a Boolean) the compiler can inform the programmer of his mistake, which is far better than having to chase its curious consequences after the program has run.

However, not all language designers would agree. Some languages, by complex rules of automatic type transfers and coercions, prefer the dangerous tolerance of machine code, but with the following added disadvantages:

(1) The result will often be 'nearly' right, so that the programmer has less warning of his error.
(2) The inefficiency of the conversion is often a shock.
(3) The language is much complicated by the rules.
(4) The introduction of genuine language extensibility is made much more difficult.

Apart from the elimination of the risk of error, the concept of type is of vital assistance in the design and documentation phases of program development. The design of abstract and concrete data structure is one of the first tools for refining our understanding of problems, and for defining the common interfaces between the parts of a large program. The declaration of the name and structure or range of values of each variable is a most important aspect of clear programming, and the formal description of the relationship of each variable to other program variables is a most important part of its annotation; and finally an informal description of the purpose of each variable and its manner of use is a most important part of program documentation. In fact, I believe a language should enable the programmer to declare the units in which his numbers are expressed, so that a compiler can check that he is not confusing radians and degrees, adding height to weights, or comparing metres with yards.

Again not all language designers would agree. Many languages do not require the programmer to declare his variables at all. Instead they define complex default rules, which the compiler must apply to undeclared variables. But this can only encourage sloppy program design and documentation, and nullify many of the advantages of block structure and type checking; the default rules soon get so complex that they are very likely to give results not expected by the programmer, and as ludicrously or subtly inappropriate to his intentions as a machine-code program that contains a type error.

Of course, wise programmers have learned that it is worthwhile to expend the effort to avoid these dangers. They eagerly scan the compiler listings to ensure that every variable has been declared, and that all the characteristics assigned to it by default are acceptable. What a pity that the designers of these languages take such trouble to give such trouble to their users and themselves.

13.12 Language-feature design

This paper has given many practical hints on how *not* to design a programming language. It has even suggested that many recent languages have followed these hints. But there are very few positive hints on what to put into your next language design. Nearly everything I have ever published is full of positive and practical suggestions for programming language features, notations, and implementation methods; furthermore, for the last ten years, I have tried to pursue the same objectives in language design that I have expounded here; and I have tried to make my proposals as convincing as I could. And yet I have never designed a programming language, only programming language features. It is my belief that these two design activities should be more clearly separated in the future.

(1) The designer of a new feature should concentrate on one feature at a time. If necessary, he should design it in the context of some well-known programming language that he likes. He should make sure that his feature

mitigates some disadvantage or remedies some incompleteness of the language without compromising any of its existing merits. He should show how the feature can be simply and efficiently implemented. He should write a section of a user manual, explaining clearly with examples how the feature is intended to be used. He should check carefully that there are no traps lurking for the unwary user, which cannot be checked at compile time. He should write a number of example programs, evaluating all the consequences of using the feature, in comparison with its many alternatives. And finally, if a simple proof rule can be given for the feature, this would be the final accolade.

(2) The language designer should be familiar with many alternative features designed by others, and should have excellent judgement in choosing the best and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of the scope and purpose and range of application of his new language, and how far it should go in size and complexity. He should have the resources to implement the language on one or more machines, to write user manuals, introductory texts, advanced texts; he should construct auxiliary programming aids and library programs and procedures; and, finally, he should have the political will and resources to sell and distribute the language to its intended range of customers. One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.

13.13 Conclusion

A final hint: listen carefully to what language users say they want, until you have an understanding of what they *really* want. Then find some way of achieving the latter at a small fraction of the cost of the former. This is the test of success in language design, and of progress in programming methodology. Perhaps these two are the same subject anyway.

13.14 Appendix: annotated reading list

Naur, P. (1960)
The more I ponder the principles of language design, and the techniques that put them into practice, the more is my amazement at and admiration of ALGOL 60. Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors.

Of particular interest are its introduction of all the main program-structuring concepts and the simplicity and clarity of its description, rarely equalled and never surpassed. Consider especially the avoidance of abbreviation in the syntax names and equations and the inclusion of examples in every section.

Knuth, D. E. (1967)
Most of these troublespots have been eliminated in the widely used subsets of the language. When you can design a language with so few troublespots, you can be proud. The real remaining troublespot is the declining quality of implementations.

Wirth, N. and Hoare C. A. R. [9]
This language is widely known as ALGOL W. It remedies many of the defects of ALGOL 60 and includes many of the good features of FORTRAN IV and LISP. Its introduction of references avoids most of the defects described above under 'Variables'. It has been extremely well implemented on the IBM 360, and has a small and scattered band of devoted followers.

Wirth, N. (1968)
This introduces the benefits of program structures to low-level programming for the IBM/360. It was hastily designed and implemented as a tool for implementing ALGOL W; it excited more interest than ALGOL W, and has been widely imitated on other machines.

Wirth, N. (1971c)
Pascal was designed to combine the machine-independence of ALGOL W with the efficiency and control of PL/360. New features are the simple but powerful and efficient type definition capabilities, including sets, and a very clean treatment of files. When used to write its own translator, it achieves a remarkable combination of clarity of structure and detail together with high efficiency in producing good object code.

Dahl, O.-J., *et al.* (1972)
This expounds a systematic approach to the design, development, and documentation of computer programs. The last section is an excellent introduction to Simula 67 and the ideas that underlie it.

McCarthy, J. (1960)
This paper describes a beautifully simple and powerful fully functional language for symbol manipulation. It introduces the scan–mark garbage-collection technique, which makes such languages feasible. LISP has some good interactive implementations, widely used in artificial intelligence projects. It has also been extended in many ways, some good and some bad, some local and some short-lived.

ASA Standard FORTRAN (1964)
This language had the right objectives. It introduces the array, the arithmetic expression, and the procedure. The parameter mechanism is very efficient, and potentially secure. It has some very efficient implementations for numerical applications. When used outside this field it is little more helpful or machine-independent than assembly code, and can be remarkably inefficient. Its input–output is cumbersome, prone to error, and surprisingly inefficient. The standardizers have maintained the horrors of early implementations (the equivalence algorithm, second-level definition) but, in resolutely setting their face against the 'advance' of language-design technology, have saved it from many later horrors.

ASA Standard COBOL (1968)
Describes a language suitable for simple applications in business data processing. It contains good data structuring capability but poor facilities for abstraction. It aimed at readability but unfortunately achieved only prolixity; it aimed to provide a complete programming tool, in a way few languages have since. It is poor for variable format processing. The primacy of the character data item makes it rather inefficient on modern machines; and the methods provided to regain efficiency (e.g. SYNCHRONIZED) often introduce machine-dependency and insecurity.