High level languages for low level users

D. G. Evershed and G. E. Rippon

Department of Production Engineering and Production Management, University of Nottingham, Nottingham

Despite the presence of 'high level' languages, a communication barrier still exists between the majority of people and computers. This paper suggests how some present computer languages may be improved, and attempts to justify the application of increased effort to this subject.

(Received August 1969)

The growing number of computer systems installed during recent years in industry, universities and Government departments, is bringing an ever-increasing sector of the community into contact with these systems. The level of contact for many people is simply a rate demand, or electricity bill produced by a computerised accounting system, but for a large number of design draughtsmen, university research workers, undergraduates, accountants, systems analysts and many others in industrial life, the level of contact involves computer programming, rather than simple appreciation. The suggestions in this paper are not directed to those full-time programmers for whom computing is virtually an end in itself but rather to these 'low level' users for whom computing is a small but essential part of their job. It is accepted that systems programs, and large, frequently-run programs require the additional effort of being written in machine code to reduce computing time, but those writing such programs are so familiar with the system that flow charts are converted into programs with scarcely a thought for the syntax or semantics of the language.

Because of general ignorance of computing science, computers are, in most people's minds, surrounded by an aura of mystery—machines of infinite complexity, limitless capacity, and lightning speed. This image is not dispelled when the outsider looks into the computer room to see the professional programmers and machine operators communicating with the machine and with each other in a totally unintelligible jargon. Faced with this situation, it is easy to understand how all but the most determined 'low level' user is quite prepared to allow a few selected slide-rule calculations to substitute for a more complete computer analysis. Thus, one often hears the complaint that, for a small programming job, the calculations could be done by hand in the time that it takes to write, compile and run a program. This is a valid point for the novice programmer, who never becomes sufficiently practised to remember, for example, the intricate input/output procedures of some high level languages.

When this situation arises, as indeed it must, particularly in industry, this is evidence that the computer installation is not being used to maximum effect. Consider, for example, the design draughtsmen who, because he does not feel himself competent to perform a computer-aided evaluation of his design, uses excessively high safety factors. The cost of using the computer for a correct design evaluation may be less than the cost of overdesigning.

The current problem is to determine why occasional users are reluctant to program, even though it may considerably assist their work, and to suggest methods of overcoming this problem.

The most important factor is psychological—the certain knowledge that it will require many runs on the computer to perfect the program through the illogicality of the programmers' mind. However, once this barrier has been overcome, and the 'low level' user starts to write his program, he finds that his inability to successfully carry out routine tasks, which was one of his reasons for turning to the computer, is exploited by the syntax of the programming language itself.

This particular problem is not confined to the novice programmer, but also the experienced user seems to be continually making slight syntactic errors, which cause failure at compile time. However correct the rest of the program may be, a ; missing in ALGOL 60, or a character punched in the wrong card column in FORTRAN IV may cause failure on compilation, requiring the program to be re-run. One tends to accept such failures as occupational hazards, but even the most philosophical of users must have wondered, after his program had just failed to compile at the third attempt, whether such requirements could be avoided.

Although one must consider the fields of language and implementation separately, there is evidence that the basic needs of the programmer have not been sufficiently considered by both the 'language designers' and the 'system programmers'. The development of ALGOL, for example, was undertaken with the most regard for the power and consistency of the language. The logical result of this is ALGOL 68 which is totally incomprehensible not only to 'low level' users but to the 'experts' as well. On the other hand efficiency of implementation was the prime concern of the FORTRAN system programmers, leading to such stumbling blocks as the assigned GOTO statement or the need to specify dimensions of array parameters.

Some suggested improvements

Since the computer is a machine it can only process input which has been ordered in a predetermined manner. Therefore, if a program is to succeed, it must be written with complete accuracy. Whilst this need for accuracy is accepted, the following four sections give suggestions as to how errors induced by some present high level languages may be reduced.

Although examples are not given for every high level language, the principles outlined should be generally applicable.

(*a*) *Input/Output routines*

All computer programs are intended to produce output of some kind, and the majority require data input. Therefore,

it is especially important that these routines should be easy to use and remember.

The generalised, formatted input/output of FORTRAN IV is certainly not easy to use or particularly easy to remember. The basic statements are

READ (m, n) list

and WRITE (s, t) list

where m and s are the input and output device numbers respectively and n and t the locations of the format specifications to be used. List is the list of variables to be input or output.

The majority of programmers will only use the card reader (or paper tape reader if the installation is so orientated) for input and the lineprinter for output, so why not assume these devices? The more advanced programmer would still have special procedures available for using the other input/output devices such as magnetic tapes and graphplotters.

An assumed format could be introduced for WRITE, such as six digits before the decimal point and three after. It would assist if ANSI FORTRAN were to include this concept in its language definition and were also to define that when a number is too large for the specified field then the field should be extended. Then the programmer would be rewarded with a number rather than the galaxy of stars which are common in so many implementations today.

Input formats are completely unnecessary if numbers in the data are separated by two or more 'spaces' as in ALGOL. This method is obviously more error proof than having to punch data into the exact card columns specified by a fixed format where a mistake can easily be made in the format specification, the data layout or in card punching.

With these revised techniques, to input and printout the value of the variable Z would only require the programmer to punch

READ Z

and WRITE Z

It is obviously convenient to be able to read in or print out several variables (A, B, C, D) with a statement such as:

READ A, B, C, D

Also if formats other than the assumed one were required for output, a simple layout such as:

WRITE B. A, X, Y, Z

could be used where B is the number of digits before the decimal point, A the number after and X, Y, Z the variables.

A facility often required is that of printing out text, to aid the legibility of the lineprinter output. The instruction to print 'job heading' on a new line in Atlas Autocode could be

caption ♮ job $ heading

Compare this with the equivalent KDF9 ALGOL instruction

writetext (30, [[c] job⤫heading]);

and it is immediately obvious which form is the more concise, the more simple and, therefore, the easier to remember.

(*b*) *General computing instructions*

Throughout the body of a computer program, a small number of instructions will be used many times, thus it is desirable that the 'low level' user does not have to pause unnecessarily to think whether he has written a particular instruction correctly, or to refer to the manual to determine whether, for example, he should be using (or [or '(', or even [in his instructions.

When examples from current languages are considered it becomes obvious on the one hand that certain languages are reasonably easy to write correctly and, on the other hand, other languages seem to have been designed from a completely opposite standpoint.

A common ALGOL error is that of trying to use a variable that has not been declared. FORTRAN is superior in that it automatically declares variables as it finds them. Unless the programmer overrules the computer, variables starting with the letters I-N are integer and the remainder real. It might be easier to remember A-I as integers, but the method should reduce the number of declaration errors. Mispunched variables can go undetected using FORTRAN, but if a list of the declared variables were produced at compile time the programmer could check for this.

A new line on paper tape in Atlas Autocode or a newcard in FORTRAN are sufficient in themselves to separate statements. In ALGOL, the newline is ignored and an avoidable semi-colon has to be punched in addition. Admittedly a statement occupying two cards (or lines) of program requires a continuation sign to be punched in Atlas Autocode or FORTRAN, but this arises in the minority of cases.

A particularly good example of excessive reliance being placed on the memory of the programmer is in FORTRAN where a continuation sign has to be typed in the sixth card column. Similarly, only labels may be punched in columns, 2, 3, 4 and 5, and in Egdon FORTRAN the first column is reserved for an asterisk to specify Job Control Language. Such conventions complicate programming and are extremely error prone.

It seems reasonable to assume that the fewer the formalities required in a program the fewer the errors made in trying to carry them out. This is illustrated by the method of specifying 'basic words' in ALGOL or Atlas Autocode. Such words have to be 'primed' or 'underlined' e.g. 'REAL' or (real) whereas in FORTRAN no adornments need to be added to the simple word.

In certain implementations, Egdon FORTRAN, for example, this method does mean that words such as DO, REAL and END cannot be used as variables.

This is hardly very restrictive when one considers the wide scope that the programmer has for mispunching a reserved word in ALGOL in comparison to the times he would wish to write in Egdon FORTRAN:

DO 25 REAL = INTEGER, GOTO, END

where REAL, INTEGER, GOTO and END are variables!

Some ALGOL conventions differ between systems. In KDF9 ALGOL for example, the 'equals' of an assignment statement and a conditional statement are := and = respectively, whereas for Egdon ALGOL the = applies to assignment statements and 'EQ' to conditional statements. Atlas Autocode sensibly uses only = for both types of statement. Similarly, the relational symbol for 'less than' is 'LT' in Egdon ALGOL, but Elliott 4100 ALGOL uses <. The Elliott ALGOL is also inconsistent within itself because it uses 'LE' for 'less than or equal to' instead of ≤. Further confusion can arise if the lineprinter does not have the same symbols as the input punches. It is particularly annoying to have upper and lower case characters in a program but a computer printout of it completely in upper case.

This problem could be eliminated by the adoption of a limited standard character set in the hardware representation.

A further common requirement is to loop through part of a program, increasing the values of variables by a preset amount. The ALGOL 60 instruction

for k:=i step m until n do

and ALGOL 68 instruction

for k from i by m to n do

are self explanatory, but rather lengthy.

The equivalent Atlas Autocode instruction

cycle k = i, m, n

and the FORTRAN IV instruction

DO 25 K = I, N, M

are indeed shorter and, to the beginner, equally acceptable even though the m and n are reversed. The two latter forms may lead to errors because it is not clear whether i goes to n by steps of m, or to m by steps of n. Although there may be very few users of both AA and FORTRAN (except in Universities!) potential errors could be avoided by instructions such as:

LOOP K, I TO N BY M which is a compromise between clarity and length.

ALGOL programs characteristically have a large number of 'blocks' each contained between a begin and end, but difficulties are encountered in matching the begins and ends, particularly when the blocks are nested. PL/1 goes part way to solving the problem by having the computer count the ends, but the ALGOL system is probably more satisfactory than the system of cycle and repeat and labels.

The additional 'goto' instructions do make it harder to work through the program sequence, and flow charts often become inconveniently large.

Short of expecting the computer to work out the logic of the program, the most satisfactory system appears to be found in FORTRAN IV. When for example the instruction

DO 70 I = 1, 5, 1

is encountered, the computer executes the program section under label 70, and then returns to the line below the original 'DO' instruction. Using this system advantages are gained by having each block separately identifiable, thus enabling the programmer to check independently both the instructions in the blocks and those executing the blocks without having to sort through a maze of begins and ends or labels and jump instructions.

(*c*) *Error reporting*

Clear error reporting of failures during compiling or at run time is an important factor in the overall efficiency of any computer system.

Some compilation errors at the start of a program, e.g. undeclared variables, can have repercussions throughout the remainder of the text, causing spurious failure messages to daunt the novice programmer. However, this does not cause as much frustration as when the computer fails a program and the error message is incomprehensible. How many readers could confidently explain the cause of the Egdon 3 system failure message 'Dyadic operator incompatible with arguments'? Another failure message from Egdon is 'Goto expression not designational'. So much simpler is the equivalent Atlas Autocode error message 'Label not set', giving the name of the label as well. Obscure words used in error messages tend to emphasise the esoteric nature of computing science, and this in turn further adds to the reticence of the 'low level' user to take full advantage of the computer.

From the point of view of saving overall computing time it is desirable that compilers detect all errors in a program at one run.

This is so with the Edinburgh University Atlas Autocode compiler, but the WALGOL compiler, for example, fails to report the undeclared variables in a block or inner blocks until all other compiling errors are corrected. Although there may be no reasonable alternative to this system in this particular implementation, the 'low level' user sees this quite rightly as potentially wasteful on computing time and as yet another frustration to be overcome when he turns his hand to programming.

In high level languages, the nature of run time failures is not as useful as the output information about its position in the program text. For example, it is exceedingly tedious to have to work through a program to discover the cause of an excessively large number failing the program 'overflow set'. Computer software could easily be designed to give the line at which the program failed. In this respect, the run time failure messages of Atlas Autocode are ideal, giving simply the nature of the failure and, more important its position in the text. Egdon ALGOL error messages are particularly poor for locating faults at run time. The position has to be traced by working through a considerable quantity of dumped information about LINK, SJNS, CELLS and QSTORES. If the programmer can logically trace through this information to find the point of failure, one is inclined to ask why the software programmers did not build such steps into the computer.

In FORTRAN IV, if an attempt is made to print, for example, an integer of seven digits using a format specifying only six digits by a statement such as

```
      K = 1036295
      WRITE (10,2)K
    2 FORMAT (I6)
```

the programmer is rewarded with six stars! This is extraordinarily unhelpful.

Improvements to error reporting would be of great assistance to the user, and might prevent further diagnostic runs being necessary to trace locations of failures.

(*d*) *Conversational mode*

In recent years a major development has been the use of computers in conversational mode. Although systems for visual display, syntax analysis and interactive compilation of programs are well established, much work is at present being done on genuinely interactive compilers and incremental compilers.

As the level of interaction between the computer and the programmer increases so inevitably more and more of the machine's power is required for the internal 'housekeeping' for the system. Thus the stage is reached when the marginal advantage of the programmer being able to compile his program incrementally rather than in toto is offset by the cost of software development and the reductions in available computing power.

A further factor is that a computer should be kept busy continually if a rapid turnround of jobs is expected. Thus, any interactive system must allow multi access to users together with the facility of processing background jobs. The situation where a programmer completely occupies a machine for visual display or compilation of a program cannot be economically viable. Even so, such systems are found in practice.

Thus, the use of computers in the conversational mode has great potential, particularly for the low level user, but the cost of software and the computing power absorbed by other than the most basic systems at present makes them uneconomic and unsuitable for all but the largest machines.

Economic justification

One could not expect the human factors aspect of computer programming to be improved unless there were strong economic arguments for doing so. Some arguments are presented here, exposing inefficiencies which at present

go unrecognised, and focusing attention onto lost opportunities.

With the esoteric content of programming eliminated, a much broader section of the population will become potential computer users. Not only would more people be capable of using a computer, but those who had previously considered it frustrating and time-consuming might now find it worthwhile. For example, with the advent of scientific management, managers could well use the computer for small simulations or analyses of local problems requiring immediate solution. In general, professional people can expect the routine tasks to be performed by computer, leaving them more time to concentrate on the fundamental aspects of their work. Apart from the sociological desirability of this trend, it should bring tremendous indirect advantages in the form of improved industrial efficiency, better designs, etc.

An area where direct cost savings would occur is in training. Accepting that less involved high level languages, such as Atlas Autocode, are more easily assimilated by new users, one would expect that training periods could either be shortened or a better training given in the same time. This would attract those potential users, such as senior managers, who have only limited amounts of spare time in which to learn to program. Unfortunately, at present, when a new computer system is installed or when programmers change jobs, the method of communicating with the computer has to be re-learnt. Even with a widely used language such as ALGOL 60, different installations have different input/output procedures.

Here again, the simpler the language and operating system, the shorter the disruption of normal working after a changeover. This situation would be avoided by the use of standard languages and systems. One reason for not standardising is that some languages are more efficient than others for particular applications, but a penalty is paid in the enormous duplication of effort carried out. If a single computer language were introduced, it would still need to be translated into several other languages for use by other countries, but problems of interchangeability of programs and personnel would be reduced to a minimum.

Considerable thought and effort has been aimed at improving compiler efficiency. However, if one compilation error has been made by the programmer a corrected program will have to be run, doubling the computing time. Thus, any fractional advantages gained by a fast compiler are swamped by the wasted computing time caused by human error. The authors maintain that a considerable quantity of compiler efficiency could be sacrificed in order to accommodate language changes orientated towards reducing human errors.

Similarly, advantages of reducing the running speed of compiled programs are recognised, but less apparent is the time wasted through incorrect data or logical errors. It is clear to see how data can be mispunched, particularly in FORTRAN where each character has generally to be in a specific card column, but not so obvious that some logical errors can also be induced by poor language design. Admittedly logical mistakes are inherent in any human task, but some of the language improvements suggested above show how these might be reduced.

From the authors' experience at several establishments programs often fail because of computer operator mistakes. Such mistakes include feeding paper tapes in backwards; omitting to replace punched cards rejected by the card reader; failing to call up special procedures for graph-plotting or visual display; and failing to switch on required output devices. Some of these pitfalls could have been eliminated by better design of the system, giving direct cost savings. Introducing new improved systems is not necessarily advantageous however because adapting from an old system to a new one appears to require as much re-training and induce as many programming errors as changing from one language to another.

Conclusion

When an overall view is taken of the effectiveness of present computing systems, serious deficiencies are seen in the functioning of 'high level' languages. With the expected increase in the number of occasional computer users, there will be a corresponding increase in the costs expended on training people to program and in the computer time wasted on program development. This emphasises the need for a computer language which is easy to understand, easy to remember and free from error prone conventions. Such a language is unlikely to be designed by computer personnel who have had no training in human factor techniques, and who are unconsciously ingrained with computer jargon that is incomprehensible to the majority.

Let us hope that those commissioned to write new languages, or even subsets of the old, will be sufficiently aware of the needs of the programmer to place him in his important position as the ultimate consumer, rather than to exploit his weaknesses with the tedious and, in many cases, unnecessary conventions inherent in existing languages and implementations.

Book review

Annual Review in *Automatic Programming*. Vol. 6, part 4. 'Joss-II: Design Philosophy', by J. W. Smith, 1970; pp. 183-256. (*Pergamon Press Ltd.*, 30*s* = £1.50)

Papers in which system designers make an honest attempt to review, explain and, where necessary, criticise, the decisions that they made in designing a software system are all too rare (though similar papers about hardware systems are virtually non-existent). This paper on the design philosophy of JOSS-II—and in particular the sections on list structures and list processing, conditional expressions and storage management, and the 'reprise'—is very worthwhile on this account. However, other sections of the paper describing the details of the JOSS-II language are less successful. Much space is taken up by a rather discursive account of the rules of the language. A briefer, more formal description of the language, with more discussion of the alternatives which had been considered and discarded would have been preferable.

Even more regrettable is the lack of any attempt to assess the merits and demerits of the language, relative to other general purpose languages both conversational and conventional.

A paper such as this should not have had a 22-item bibliography in which nothing other than papers and reports relating to JOSS appears.

B. RANDELL (Newcastle upon Tyne)