Improving Problem-Oriented Language by Stratifying It

By Philip R. Bagley

In order to lighten the programming task, further development of problem-oriented computed languages must take the direction of relieving the programmer of specifying those details which in theory can be supplied mechanically. The first step toward accomplishing this is to develop programming languages which permit the mechanically-supplyable expressions to be stated separately from the remainder of the program expressions. An immediate benefit of this first step is that the programmer is free to formulate, express, and modify procedures and data for the solution of a problem without concern for the manner in which the procedures and data will be implemented on a computer. By having the mechanically-supplyable detail separately stated, the way is paved for the development of translating programs which can supply this detail. These ideas are not radically new, but rather serve to emphasize a trend which is apparent in current computer languages but which has not been carried far enough.

Introduction

This paper is a result of a search for principles to guide the further development of programming language. It is the thesis of this paper that in order to lighten the burden of programming, further development of problem-oriented computer languages must take place in the direction of relieving the programmer of specifying those details which in theory can be supplied mechanically (that is, by programs). This thesis is not advanced as a revolutionary principle, but rather as a restatement and emphasis of a trend which is already apparent in current programming-language development. The definition in this paper of several levels of language within a programming language may serve to intensify the trend.

A General Characteristic of Current Programming Languages

If we hope to supply mechanically those program expressions which in theory can be so supplied, we must first organize our program language in a way which allows those expressions to be stated separately from the expressions which cannot be so supplied. It appears, however, that in most current computer languages these two types of expressions are not wholly separable. In the succeeding paragraphs I try to substantiate this statement about non-separability, both to illustrate what I mean by separability and to demonstrate that current languages fall short of this goal.

In most if not all of the current computer program languages—called "problem-oriented languages"—the programmer must be concerned in some degree with how his program will be handled either on a specific computer or on a class of computers with specific characteristics. The programmer must in general be aware of the following types of information at the same time that he is working out the program procedure: the amount and type of secondary storage (such as magnetic tapes) that will be used, the arrangement (coding and format) of data in internal and secondary storage (this includes the computer's method of representing numbers), the likelihood of various types of errors and a selection of methods for dealing with them, and the relative frequencies of various operations to be performed.

We offer some illustrations of specific cases of non-separability. These illustrations have been drawn from COBOL (Conference on Data Systems Languages, 1960) and MAD (Michigan Algorithm Decoder, based on ALGOL) (Arden, Galler, and Graham, 1960) because these languages are moderately well known and represent in large measure the state-of-the-art in programming-language development.

(1) In COBOL, a datum cannot be described as an abstract quantity (that is, in terms of the kind of quantity and its precision or number of characters), but it must be described in terms of how it will be represented in a machine's internal storage. One cannot say, for example, "quantity A is a dollar value precise to 1/100 of a unit," but rather that "quantity A is a decimal value expressed by a dollar sign followed by six digits, a comma and a decimal point to follow the first and fourth digits, respectively, when printed, but not when stored, initial zeroes to be suppressed," etc.

(2) In COBOL, the references to program error-handling routines must be made in the same sentences as those operations which may give rise to an error, instead of being expressed in separate statements.

(3) In COBOL, the choice of operation names for the movement of information in or out of the computer depends on what type of input-output device is to be used. Example: READ, WRITE, OPEN, and CLOSE refer to tape files; ACCEPT refers to low-volume input devices such as card readers; DISPLAY refers to low-volume output devices.

(4) In COBOL, notations indicating where the procedure may be segmented for the purpose of fitting it in machine storage are written as part of the procedure statements.

(5) In COBOL, the number of characters comprising a quantity must be known and kept in mind when that quantity is moved from one "field" (storage position) to another, for if the receiving position is smaller than the source position, one or more characters will be lost.
(6) In MAD, there is no explicit provision for expressing the precision with which quantities will be stored and processed, hence these precisions are determined implicitly by the characteristics of the machine on which the procedure will be run (or alternatively by its translator); the programmer must know these precision factors in advance of writing his program in order to know what the precision of his result will be.
(7) In MAD, for recursive use of a subroutine (that is, in the case that a subroutine calls itself) one must express "save" and "restore" operations to preserve the subroutine temporary storage on a push-down list. These operations are not part of the logic of the subroutine, but rather a consequence of the way it is implemented.

By far the strongest dependence of the purely logical part (the procedure and data description part) of current languages on computer characteristics cannot be illustrated by simple examples like the above. This dependence can be recognized only through a first-hand knowledge of the extent to which the choice of procedures and internal coding of data is determined by the physical characteristics of the computer that is expected to be used.

These examples are not meant to imply that in all current programming languages one cannot separate to some extent those expressions concerned solely with the logic of a program from those concerned with how the program is to be implemented for a specific computer. But on the other hand, it is not possible yet wholly to separate the two kinds of expression. I believe that it is correct to say that there does not exist today an operating programming system* in which it is possible to do the following:

To write down an arbitrarily-chosen procedure to be carried out, and an explicit description of the data upon which it is to be carried out, which meet the conditions that:

(1) The procedure and data description need not specify how the data is to be coded or physically organized, nor specify any guidelines to efficiency or reliability of operation.
(2) Any number of humans who were given the procedure and data description, along with the information required to understand the symbols, would, when they applied the procedure to the data, arrive at identical results except for the physical spacing of the output.

* This phrase is meant to rule out ALGOL (Naur *et al.*, 1960) which by itself is not a complete programming language. Until it is extended, it lacks ability to express input-output formats and processes, for example.

The above paragraphs indirectly illustrate what is meant by separability of procedure and implementation expressions. This lack of complete separability of procedure and implementation expressions in current languages stands in the way not only of developing translating programs to mechanize the implementation task as fully as possible; it also imposes a positive twofold burden on the programmer. First, it prevents him from concentrating exclusively on procedure logic during the time he is working out a method of solution to a problem. Second, implementation details (such as the way data is coded and stored) become interwoven with the procedure logic, making it less easy to revise tentatively expressed procedures.

Outline of Stratification

As I stated earlier, I believe that a necessary step toward simplifying the programmer's job is to separate completely the programming expressions concerned with implementation from those concerned with procedures. For reasons which will become clear, it appears useful to stratify the language into four levels rather than the more obvious two, so that a programmer may concentrate on and state separately the following types of information:

(1) The procedures and the data description (in the abstract) devoid of expressions concerned with anything other than the intellectual method of problem-solution.
(2) The essential constraints on the intellectual or abstract solution of the problem. These constraints are largely concerned with physical formats of inputs and outputs.
(3) The implementation details, such as the internal format and coding of data, storage allocation, conversion of procedure into machine instructions.
(4) Information about the program, which cannot be mechanically deduced, and which can be given at the programmer's option for the sake of efficiency, such as the relative probabilities of specific choices at a decision point.

The succeeding paragraphs will discuss these strata in more detail, and the potential benefits and drawbacks of this stratification scheme.

If we actually use the method of stratification outlined here, I feel strongly that we will find that a vast percentage of a programmer's effort is expended on specifying implementation (levels 3 and 4 above). If we can show that these levels can be mechanized, it will then be obvious what a price we are paying in order to have the unnecessary luxury of being able to specify how a program is to be implemented.

Procedure and Data Description

The first level of language should enable the programmer to specify procedures and the nature of the data involved with those procedures, this language to be completely devoid of expressions concerned with any-

thing other than what a human being would need in order to carry out the procedures and achieve the desired result. The procedures would, nonetheless, have to be completely unambiguous, for we are not assuming any problem-solving ability on the part of the executor, only the ability to follow the procedure steps.

That the array of procedures available to the programmer in this language is limited by the choices of the language designer does not concern us here. What does concern us is that the portion of the language used to describe procedures and data should not refer to or imply how these procedures and data will be implemented on a computer.

At this language level we must be free of concern with such things as: the word length of the computer or storage medium involved, that the storage media (internal or external) have restrictive characteristics (tapes have one-dimensional serial access), that storage registers are physically adjacent address-wise in one dimension (with the consequence that we cannot insert a register between any two consecutive registers), the specific representations of data in storage (such as the manner in which "logical" or "status" variables are numerically coded), the organization of program and data in storage with respect to time, procedures for dealing with machine errors, details of input-output format (editing and spacing).

The Essential Constraints

The second stratum includes constraints consisting primarily of details about input-output formats. These constraints have the common characteristics:

(1) they are extraneous to the planning of the procedures,
(2) if they are not specified by the programmer, they can be supplied more or less arbitrarily by the translator.

Examples of this type of constraint are: the physical arrangement of data on some external medium (e.g. punched cards, tapes, printed page), the coding of data or of characters or both as the data is found on some external medium. I do not mean that specific input and output formats cannot be requirements of a problem, but I am claiming that the formats can be described separately from their content.

Expressing data formats and coding is often intricate. There is perhaps a lesson here for designers and users of future programming systems: in cases where exact format is not really important, formats should be laid out as far as possible so that their description will be simple.

The Implementation Details

Implementation details consist of that information which must be supplied in order that a program can be run on some specific computer. This includes the internal (to the machine) format and coding of the procedures and data, the allocation of storage (with respect to time), the selection of specific input-output units and operations to control them, additions to the program for performing checks, and a certain amount of compensation for errors.

Current translators supply some amount of this implementation detail for us, but they rarely supply all of it. Programmers supply some because of their conviction that the efficiency of the resulting program will thereby be greater. The author's contention is that these details are *in theory* wholly mechanizable. At the present time, however, we are weak in our ability to write translators which can both supply all these details and turn out an efficient program.

Information Relating to Efficiency

The fourth stratum of information is information concerning the programmer's intent and expectations regarding the program. Examples would be: (1) the relative frequencies with which various choices will be made at a single decision point, and (2) precedence matrices to show interdependence of various segments of programs.

This class of information can contribute to the construction of a program which is more efficient than one constructed without this information. Furthermore, this class of information is not deducible by any known techniques (short of operating a program).

It is the author's private and unsubstantiated conviction that this class of information is rarely vital to the implementation of a program and can therefore be largely neglected. The major influence on the efficiency of a program will be the programmer's basic choices of procedure steps—this information will be incorporated in the top stratum or procedural level.

Benefits of Stratification

Let me enumerate what appear to be the potential benefits of a language stratification such as I have outlined.

First, a most obvious benefit is the freedom the programmer would have to work out methods of problem solution. He would be free of concern about matters of efficiency, reliability, choice of terminal equipment and secondary storage, coding and format of data in storage, etc., until his program concept has been completely worked out. He would not necessarily achieve an error-free procedure at this stage, but he is hopefully less prone to error if he is not forced to deal with matters irrelevant to the logic of problem solution. Furthermore, he is free to revise the logic without having first to disentangle from the logic any concepts concerned with implementation.

A second important benefit is that, by relieving the programmer of some of his present burden, he will be able to cope with problems whose size or complexity are at present beyond his capability to deal with effectively.

Third, the ability to express the essential algorithm and the description of the data involved, without any of the

details concerned with implementing them on a computer, opens the way to evolving checking methods for discovering and correcting logical errors. It is conceivable that a program could be written, and the logic completely checked out, without any consideration being given for putting that program on any computer.

Fourth, if a program can be expressed in terms of the strata that have been described, then the way is paved for mechanizing the translation of that program into machine code. This translation process (which might be done either by compiler or interpreter) is in theory mechanizable. There are some practical difficulties, however, concerned with achieving efficiency of the resulting translated program.

Fifth, the expression of a program on various levels will make easier the understanding of the program by persons other than the original programmer. In other words, the documentation of a program, which is essential for most programs, is thereby clarified. Hence if a second programmer needs to modify the program, as is occasionally the case, he will be substantially aided by clear documentation.

Sixth, a program expressed at the top level—without regard for implementation—has a moderate degree of universality. That is, it can be implemented for a wide range of computers. It would be truly universal if it were not for the fact that not every computer can execute every program. (This inability may be due to lack of appropriate input-output characters, or lack of ability to execute a program at a tolerable rate of speed, or lack of suitable terminal equipment.)

Seventh, when the task of implementation has been to a significant degree taken over by a translating program, the associated programming language will be easier to use. This ease will be the result of the elimination of a vast number of programming rules which are concerned with implementation. I, for one, look forward with great eagerness to *thin* programming manuals.

The Bugaboo of Efficiency

It may be argued that the significant weakness in the scheme I propose is that programs constructed by these techniques will be likely to have poor operating efficiency. That is, they may take much longer to run than programs which do the same thing but are coded by other means. This criticism is valid. A loss of program operating efficiency is the price of making easier the writing of programs. It is the inevitable result of taking some of the burden of specifying detail from the programmer's shoulders and placing it on mechanical program translators. I feel strongly that we should be willing to pay this price—a poorer utilization of computer time and storage—in return for making faster the preparation and revision of programs for computers. Presumably we will get a substantial discount on the price in the form of a reduction of computer time spent in correcting program errors.

Some of this inefficiency is the consequence of our not knowing how to write capable translators. As our ability to write translators improves, the efficiency of translated programs will improve. However, let's not wait for better translators. Let's accept the inefficiency imposed by present translating techniques and immediately work toward relegating to translating programs as much as possible of the programming job.

How Far from the Goal are We?

Of all the better-known current programming languages, LISP (McCarthy *et al.*, 1960; Woodward and Jenkins, 1961) is perhaps the nearest to a true illustration of the top level of stratification—the language which deals with procedures and conceptual data descriptions. LISP is totally unable to talk about any aspect of implementation—neither coding, nor efficiency, nor reliability. It is thus a living example that, for at least certain classes of programs, the process of implementing a program can be entirely mechanized.

Conclusion

This paper has argued that in order to relieve the programmer's burden, the task of implementing a program for a specific computer must be relegated to a mechanical process. I firmly believe that without such mechanical aid we will be severely limited in the complexity of programs that we can deal with.

It has been pointed out by some of my friendly critics that I underestimate the importance, in many applications, of achieving efficient coding and program organization. I agree that in present situations where a program must be run on a specific machine, the ideas I am advocating may not apply (because we do not yet know how to write translators which can organize and code a program with the same ingenuity shown by the average programmer).

Given a program which can be put on a given machine only if efficiently coded, there are two obvious approaches:

(1) Code it efficiently (by the programmer's exercising close control over how the program is translated to machine code by a translating program) and run it on the given machine.
(2) Code it somewhat less efficiently (by taking away from the programmer most if not all of his control over how the program is translated) and run the program on some more capable machine.

The programmer effort and cost is greater in the first approach than in the second. The machine cost is greater in the second approach than in the first. An advantage of the first approach is that one can utilize his available machine. An advantage of the second approach is that the elapsed time to the completion of the program run is very likely to be shorter than in the first approach.

The approach that one will favour is based on an intuitive evaluation of the relative importance of these

factors. My personal attitude is that the second approach is going to appear progressively more attractive as:

(1) the capability of machines increases with respect to cost (that is, the capability per dollar increases);
(2) the desirability of minimizing overall time from problem statement to executed solution becomes increasingly important;
(3) improved techniques of translating programs into machine code are developed.

The development of larger and cheaper memories seems virtually assured. We can expect as a consequence that the storage capacities of machines will be progressively less limiting for a given cost.

In order to decrease preparation time of a program one can put more programmers on the job, but a law of diminishing returns prevents program preparation time from being made arbitrarily short. If we wish to decrease the preparation time still further, some of the programmer's work must be shifted to a facility which can carry it out more speedily than the programmer, namely, a "translating program" (this is a poor name for it) on a machine.

The reason we currently rely on human beings to produce efficiently coded programs is that our techniques used in translating programs produce results having noticeably poorer efficiency. There is no reason to believe, however, that the capability of translating programs will not be continuously improved to the point that translating programs rival in operating efficiency the present products of programmers.

I do not pretend to have analysed here all of the elements of programming nor treated all of the things which make programming hard. I have presented one major facet of programming complication and suggested an approach to alleviate it.

Grateful acknowledgement is made to Thomas L. Connors of The MITRE Corporation whose searching questions and comments led to the formulation of the ideas expressed here.

References

ARDEN, B., GALLER, B., and GRAHAM, R. (1960). *The Michigan Algorithm Decoder.* Ann Arbor, Michigan: Univ. of Michigan.
CONFERENCE ON DATA SYSTEMS LANGUAGES (1960). *Initial Specifications for a Common Business-Oriented Language (COBOL).* Washington, D.C.: U.S. Dept. of Defense.
MCCARTHY, J., *et al.* (1960). *LISP I Programmer's Manual.* Cambridge, Mass.: M.I.T. Computation Center and Research Lab. of Electronics.
NAUR, P., *et al.* (1960). "Report on the Algorithmic Language ALGOL 60," *Communications of the A.C.M.*, Vol. 3, p. 299.
WOODGER, M. (1960). "An Introduction to ALGOL 60," *The Computer Journal*, Vol. 3, p. 67.
WOODWARD, P. M., and JENKINS, D. P. (1961). "Atoms and Lists," *The Computer Journal*, Vol. 4, p. 47.

Book Review

Digital Computer and Control Engineering, by R. S. LEDLEY, 1960; 835 pages. (London: *McGraw-Hill Publishing Company Ltd.*, 112s. 6d.)

This book forms a comprehensive introduction to digital system engineering, and many of its 23 chapters are authoritative and well written. The book is divided into five main sections under the following titles:

1. Introduction to Digital Programmed Systems.
2. Functional Approach to Systems Design.
3. Foundations for the Logical Design of Digital Circuitry.
4. Logical Design of Digital Circuitry.
5. Electronic Design of Digital Circuits.

The general tone of the book is well suited to the needs of the advanced student of engineering. The first section introduces the main topics of computer engineering, and prepares the reader for subsequent sections of the book. It contains in addition two chapters devoted to programming, the first presenting elementary concepts, the second covering more advanced topics, ending with a brief description of automatic-programming techniques including ALGOL. Most of this introductory material is presented remarkably clearly. However, floating-point representation is dismissed in one and a half pages leaving, I fancy, the student unaware of its importance, and complementary representations of negative numbers are not mentioned at all!

Section 2 is a brief introduction to systems design which concludes with the introduction of PEDAGAC, a simple general-purpose computer whose design is used throughout the remainder of the book to provide the "thread of continuity" between the various topics. Sections 3, 4, and 5 deal with the central theme of the book, logical and electronic design.

I thought the author seemed more at home with logical design techniques than with electronic design. In particular, I enjoyed his treatment of logical design under constraints, in Chapter 12. A good deal of this is original work and is presented as a complete treatise for the first time. Earlier chapters lead naturally from first principles of Boolean algebra to the manipulation of Boolean matrices and the design of arithmetic and control circuits.

The section on electronic design, particularly Chapter 20 which deals with semiconductor circuits, is less successful. I found three incorrect and several confusing descriptions of circuit behaviour. Nevertheless, a great deal of ground is covered, including comparatively recent developments such as tunnel diodes, microwave logic, and cryotrons.

On the whole, the book must rank as one of the most successful introductions to computer and control engineering.

N. E. WISEMAN.