Principles and Problems of a Universal Computer-Oriented Language

By Philip R. Bagley

Hypothesized several years ago, UNCOL (Universal Computer-Oriented Language) has as its basic premise that programs expressed in a problem-oriented language (such as ALGOL) can be translated first into UNCOL and thence into machine code. This translation would involve keeping invariant certain aspects of a program while it adapts those aspects which depend on the machine chosen to execute the program.

The Goal: Convertibility of Programs

One much-desired goal in the computing field is the ability to run a program on various computers without having to do a consequential amount of rewriting of the program. As a step toward this goal, we wish to investigate here the problems of translating a computer program from some source language, in which it is initially expressed, into one or more target languages. These target languages will usually be machine codes for specific computers.

The source language form of a program might be at any of several language "levels." It could be in any of the following:

(1) machine language for a specific machine;
(2) any one of a number of existing or future problem-oriented languages, such as FORTRAN (IBM, 1958);
(3) a language which is largely independent of a specific machine and which is not especially problem-oriented, such as ALGOL (Naur *et al.*, 1960; Woodger, 1960).

We shall be primarily concerned in this investigation with programs expressed initially in language other than machine language (machine code). We are at a loss to deal effectively with those in machine code because we do not know how to formulate rules for extracting the essence or intent from a program expressed in machine code. This difficulty exists because, when a procedure is expressed in machine code, some information is obscured (e.g. a variable is often forced to be expressed as some integral multiple of computer word-length), and some extraneous data or actions may be added (such as introducing an auxiliary variable indicating the quantity of entries in a list).

It is not sufficient to have a language which can only express enough concepts to permit the construction of a corresponding algorithm (i.e. procedure) in a machine language. To be acceptable, an algorithm for a specific machine must be tolerably efficient in its operation. That is, its operating time should be somewhere in the neighbourhood of the theoretical minimum operating time for any equivalent algorithm executed by that specific machine. Even if the translation to machine code can be accomplished, some programs will not operate with satisfactory efficiency on an arbitrarily selected computer. A specific computer may be inadequate for a given program in any of several ways:

(1) It may lack sufficient internal storage of the rapid random-access type, with the consequence that instructions or data must be brought in as segments from secondary or external storage an excessive number of times. What is excessive will depend on the choice of the specific program and of a specific computer.
(2) It may have a secondary storage which is randomly accessible at an average interval which is significantly greater than that envisioned by the original programmer. To illustrate, a program that refers to drum locations at random, probably cannot be made to run tolerably efficiently if tapes are substituted for drums.
(3) It may lack a special terminal device, such as a light gun, and have no suitable substitute therefor.
(4) It may simply take too long to perform the necessary computations.
(5) Some characters needed by a program may be lacking from its set of input-output characters. Whether any character transliteration to available characters is acceptable depends on the specific application.

It is clear that, given a program, some computers cannot execute it in a feasible manner, because of gross inefficiency of operation or lack of a suitable terminal device.

The Nature of the Conversion

What is the basic nature of the translation of a program from source to target language? *Certain aspects of the program must remain unchanged through the translation process.* The most important of these invariant aspects are the essential algorithm (its "meaning" or "content") and the form and formats of the data. Among the most important of the non-invariant aspects are the organization of the program and data in both internal and secondary storage. "Invariant" and "non-invariant" correspond at least approximately to our intuitive notions of "machine-independent" and "machine-dependent" respectively. It will be convenient to continue to use these latter terms since they are more familiar.

The process of translation involves, though not necessarily in sequential steps, the following:

(1) Selection from the program of those aspects which require modification because of the characteristics of the particular target machine (that is, the machine which will actually execute the translated program).
(2) Appropriate modifications of the program content in accordance with:
   (*a*) the logical structure of the target machine;
   (*b*) the requirements for efficiency of execution on the target machine;
   (*c*) the need for reliability checks.
(3) The expression of the modified program content in the form of machine code for the target machine.

The task of analysing the original program is made more difficult because of several factors:

(1) Existing languages for expressing programs are not wholly explicit. There is information essential to the program which is not expressed directly in the program language. As an example, neither FORTRAN nor ALGOL has provision for expressing the number of digits required for a variable. Sometimes non-explicit data is written somewhere in a manual, as, for example, a collating sequence (precedence of characters).
(2) There is sometimes arbitrary and logically unnecessary information in a program formulated in an existing program language. An example is the sequence in which constants are stored; another is the choice, in some instances, between fixed-point and floating-point arithmetic.

Translation, then, involves adapting the machine-dependent aspects of the program while preserving unmodified the machine-independent aspects of the program.

The UNCOL Concept

The use of an intermediate language form called UNCOL (SHARE *ad hoc* Committee on Universal Languages, 1958; Steel, 1960) is the basis of one proposed method of making programs convertible. The central notion of UNCOL is that there is some form in which any program can be expressed which is intermediate between any problem-oriented language (POL) and any machine language (ML). If UNCOL is to be practical, the use of the intermediate language (UNCOL) form of expressing a program must not result in a grossly inefficient machine-language program.

We must agree at the outset that the programs with which UNCOL must cope are programs which are suitable for execution on more than one machine. It would make no sense, for example, to talk of translating a program which was written to diagnose malfunctions of a specific computer.

One of the problems in applying the UNCOL concept arises from the fact that the programs expressed in most if not all existing POLs are *not* wholly machine-independent. The UNCOL concept is based on the idea that at least part of a program expressed in a POL is machine-independent; hence that aspect of the program does not have to be changed when one translates the program into machine code. If the UNCOL idea is to be achieved, the process of translating from POL to UNCOL must somehow separate the machine-*independent* aspects of the program expressed in POL from the machine-dependent aspects. The machine-dependent aspects of the program may need to be organized differently according to the computer chosen as the target computer. A stumbling block in the UNCOL-to-ML translation is that we do not in general have methods (other than human ingenuity) for organizing or adapting, for a particular computer, such machine-dependent details as storage allocation, data organization, etc.

A slightly different hypothesis for what UNCOL might be would perhaps stimulate some useful ideas. UNCOL might be a descriptive language which tells how to extract the essential algorithm from a program expressed in a POL and tells how to identify the information that will have to be juggled to fit a particular machine. UNCOL in this sense would be used for writing an *accompaniment* to a program expressed in POL.

An Alternative Concept

There are techniques other than the UNCOL intermediate language idea for performing the conversion of a program from one language to another. If we have a program expressed in some source language and if

(1) we have sufficient additional information to render the program unambiguous (in a practical sense);
(2) we know how to separate the information contained in the source language into machine-independent and target-machine-dependent components;
(3) we know how to arrange the machine-dependent information to achieve suitable efficiency of operation of the program on a given target machine;

then we can write a translation algorithm for converting the program in the particular source language into the particular target language. One such translator is needed for each source language to target language pair. These translation algorithms are our current conventional compilers.

If, however, we also had a metalanguage in which we could completely describe a source language, then a translator could be written which would convert a program written in an "describable" source language into a particular target language. One such translator would be needed for each target language.

If we had a metalanguage in which we could completely describe the target language (implied here is a complete description of the characteristics of the target computer) instead of the source language, then a translator could be written which would convert a program in a particular source language into any describable target language. One such translator would be needed for each source language.

If we had metalanguages for completely describing both the source language and the target language, then a translator could be written which would convert a program expressed in any describable source language into a program expressed in any describable target language. This would indeed be a *universal* translator from any describable source language to any describable target language.

In a sense we have metalanguages for "describing" other languages. But they are operational or algorithmic rather than descriptive. For example, the translator from FORTRAN language to IBM 704 machine-code contains in some sense a description of the FORTRAN language. Such an algorithmic metalanguage appears to be universal, in that it can contain the description of any POL.

One can consider the POL-to-UNCOL translator as the metalinguistic description of the source language, and the UNCOL-to-ML (machine language) translator as the metalinguistic description of the target language. Looked at in this way, the UNCOL concept *is* a universal translator. One cannot help feeling that in this universal translation process based on operational metalanguages, the task of preparing operational descriptions of the source and target languages is much more laborious than preparing straight descriptions. But at present we don't even know how to make straight descriptions: we don't have the descriptive metalanguages we need.

A recent development by Sibley (1961) is a noteworthy step in the direction of making a "universal" translator. His program, called the "SLANG Processor," accepts as input: (1) a program expressed in a language which is supposedly machine-independent and is similar in many respects to ALGOL-58, and (2) a description of a computer. The SLANG processor then converts the program into machine-code for the described computer. At the present stage of development his system lacks generality, his program language being specifically designed for expressing compiling programs. His computer description language will nicely accommodate computers having the general characteristics of current IBM computers, but has not enough provisions for dealing with the whole class of commercially available machines.

The Translation Process

Challenging though the alternatives to the intermediate algorithmic language might be, the intermediate language or UNCOL approach appears at present to be the most promising and is the one that we will investigate further in the remainder of this paper. First, let us consider the process of going from a POL version of a program to an ML version, and observe what kinds of information are introduced at the several steps. These steps are outlined graphically in Fig. 1.

A program expressed in some problem-oriented language is to be translated into UNCOL. One such translator is envisioned for each POL. In this translation process the translator will very likely have to supply

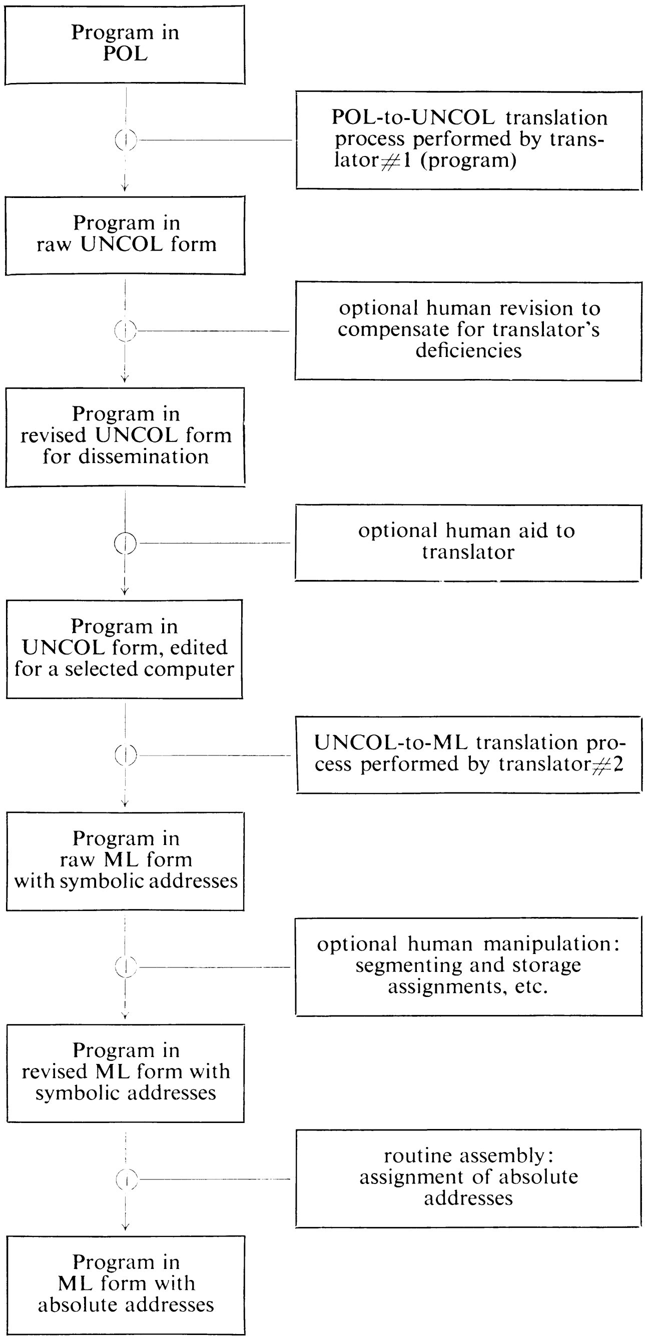
Program in POL
POL-to-UNCOL translation process performed by translator #1 (program)
Program in raw UNCOL form
optional human revision to compensate for translator's deficiencies
Program in revised UNCOL form for dissemination
optional human aid to translator
Program in UNCOL form, edited for a selected computer
UNCOL-to-ML translation process performed by translator #2
Program in raw ML form with symbolic addresses
optional human manipulation: segmenting and storage assignments, etc.
Program in revised ML form with symbolic addresses
routine assembly: assignment of absolute addresses
Program in ML form with absolute addresses

Fig. 1.—Steps in the process of transforming a program from POL to ML via UNCOL

information which is not rendered explicitly in the POL version of the program, but which is included in a user's manual associated with the POL. An example is the collating sequence (analogous to alphabetic sequence) for the assumed set of input and output characters. Furthermore, since the goal of UNCOL is to maximize the independence of a program from any particular machine, this translator has the task of discovering those elements of the program, expressed in POL, which are machine-dependent. That such machine-dependent elements exist should be apparent from the fact that most POLs are designed with a specific computer in mind. For example, data or file descriptions which are obviously constrained to fit a specific machine word-length will have to be somehow re-expressed in a form independent of machine word-length.

The UNCOL version of a program, then, is to be a representation of a program in essentially (if not totally) machine-independent form. The essential algorithm should be in a canonic form that does not impose any arbitrary sequence on steps that from the standpoint of logic could be executed simultaneously. As many as possible of the machine-dependent aspects should be expressed in the form of parameters (such as: input-output channel numbers, input-output character set), to which actual values will be assigned by the UNCOL-to-ML translator.

UNCOL should provide for the expression of information which is not logically essential to the procedure, but which can sometimes be used by the UNCOL-to-ML translator to improve the efficiency of the resulting machine-coded program. Such information, which we might term "side information," might embrace such things as: (1) approximate probabilities of following the various paths at a decision or branch point, and (2) segmenting points, to be used in case the entire program will not fit in the internal memory of the target machine.

It is to be expected that deficiencies will exist in the POL-to-UNCOL translation process. Hence provision is needed for optional manipulation, by humans, of the UNCOL output of the translator in order to compensate for the deficiencies. Examples of such deficiencies are: the resulting operating inefficiency of the translated program, inability of the translator to identify machine-dependent aspects, and lack of provision in the POL to express relative frequencies of choosing among alternate paths. If storage assignments in the ultimate (machine-language) version of the program are to be performed mechanically (later in the translation process), then by this stage the interdependencies of the various parts of the program must be known. Either they must be derivable (which is unlikely), or expressed in the POL (which is also unlikely), or inserted by a human.

At this stage, the program is "published" in UNCOL form and is ready to be adapted for some selected computer. This adaptation might best be done in two steps, of which the last is the conventional assembly. In this way a form of the program will exist which is in machine language but without actual addresses: thus a human can have the opportunity to influence the assignment of actual storage locations, and the segmenting of the program into parts in case it becomes necessary.

Bridging the gap between the UNCOL version and the machine language before assembly ("symbolic machine language") is the task of a second translation program, the UNCOL-to-ML translator. This translator must fill in the gaps, in the UNCOL version of the program, which represent machine-dependent parameters, such as: input-output unit assignments, and list of available characters. The UNCOL-to-ML translator is also to be concerned with supplying the coded values of "logical" entries in tables and with determining table format. Upon this translation process (humanly-aided if need be) falls the task of utilizing unique machine features, such as a program interrupt system, and manual controls (such as break-point switches). Before the translator is applied, however, human aid may be desirable or necessary in choosing some of the machine-dependent parameters: for example, equating those characters assumed existent by the program to those actually available on the chosen computer's terminal equipment. Finally, the translator must transform the essential algorithm into symbolic machine language, and perhaps impose some further ordering upon the parts of the algorithm in order to reduce it to the number of computations that may be carried out simultaneously on the selected machine. The output of the translator is subject to human manipulation, as mentioned above, before it is put through a routine assembly process.

General Principles for the UNCOL Language

Let us consider further what seem to be the guiding principles on which UNCOL should be based. It appears that casting UNCOL in the form of a language for a hypothetical machine would cause it to contain information which is unnecessary, and also unduly restrict the freedom to manipulate the program. For example, the mere fact that instruction steps have been written usually implies an assumption that the length of the quantities being processed will not exceed the word length of the machine. Also, the computation of intermediate variables may be indicated when, in fact, for some machines, they need never be computed explicitly (i.e. stored). This indicates that UNCOL should not be a hypothetical machine language wherein all expressions are imperative statements (commands).

To repeat what has been said before, the machine-independent part of a program which UNCOL must express are: (1) the essential algorithm, and (2) the forms and formats of the data involved. The essential algorithm is the algorithm expressed in a manner which does not impose any arbitrary sequence on steps that are independent of each other, and hence from a logical standpoint can be executed simultaneously. The *forms* of data include whatever is theoretically necessary to compute with the data: the types of quantities represented, their units, their precisions, the hierarchical relations between them, etc. The *formats* of the data are concerned with the relative positions in which input and output data appear on some physical medium. But if format description is to be kept machine-independent, it must *not* be in terms of the physical characteristics of a specific unit of input-output equipment.

The details of an algorithm are not necessarily machine-independent, hence it is important where machine independence is desired that the algorithm be expressed on a level high enough to avoid machine-dependent detail.

This level of expression to avoid machine dependence, however, is a relative thing rather than a universal constant. It depends on the collective abilities of the class of machines we seek independence of. (This is an admittedly vague notion at present.) Various parts of a program may have to be expressed in UNCOL at different levels of detail, depending on the POL-to-UNCOL translator's ability to discern the intent of the program. Perhaps only the lowest level will be truly universal, and we may be forced to resort to it to accommodate some unusual POL expression.

To give the maximum leeway for rearranging an algorithm for a selected target machine, we would like the algorithm to be expressed at the highest possible level of abstraction. The exact way in which an expression is factored for computation, for example, will be influenced by the actual instructions available on the target machine. We are without a satisfactory procedure for inspecting an algorithm and re-expressing it at a higher level of abstraction. (It could in theory be accomplished by looking up correspondences in a table of equivalent expressions, but the size of this table would be enormous.) It follows that any intermediate language such as UNCOL should not force us (except in rare cases) to re-express an algorithm in more detailed terms (i.e. at a lower level of abstraction) than the original algorithm in the source language.

Ideally UNCOL should provide us with the freedom to adjust the speed-versus-storage emphasis. Other things being equal, the most economical choice for the speed-storage ratio is that which utilizes all the storage capacity of the target machine. The choice of expressions for UNCOL should not unnecessarily restrict our freedom to rearrange the program in an attempt to achieve the optimum ratio.

An efficient result of an UNCOL-to-ML translation might require human aid. To make the human's job easier, the expressions in UNCOL—at least the ones the human has to manipulate—should be in a form readily understood and amenable to manipulation.

Efficiency

In practice there are concepts, known to the algorithm writer, which can contribute to improving the efficiency of an algorithm when it is adapted for a specific computer. Examples of such concepts are: the earliest and latest times in a program when a specific variable is used, and points at which a program can be segmented or overlaid to conserve storage. In theory, such concepts are deducible by inspection of the algorithm, but in many cases the methods for performing this mechanical deduction are either unknown or inadequate. Hence, it is a practical rather than a logical necessity that UNCOL have the ability to express information about the algorithm which will contribute to the construction of an efficient program for a specific computer.

Producing an efficient machine-language program from one expressed in UNCOL is likely to be a difficult task because of our poorly-developed understanding of algorithms. Techniques are sorely needed for finding the fastest algorithm within specific constraints.

Reliability Checks

We would like somehow to provide for reliability of program operation in the face of errors of various kinds. As far as the reliable execution of the computation or algorithm is concerned, the methods of detecting and compensating for machine errors are very machine dependent. UNCOL might accommodate expressions to indicate the timing and nature of reliability checks and compensations to be performed, while leaving the implementation of these checks to the UNCOL-to-ML translator. Specific machine reliability checks might better be provided in another way, however: by executing the machine-language program under control of a supervisory routine which contains a reliability control routine (to which control would be sent by an automatic interrupt system, or by periodic branch instructions inserted by the UNCOL-to-ML translator).

Reliability checks concerned with input data (that is, checks for correct format, sequence, quantity of data, legal symbols, etc.), on the other hand, appear to be more dependent on the nature of the input data than on the machine; hence, input data checks are more appropriately considered as part of the machine-independent procedure.

The Input-Output Problem

Perhaps the biggest stumbling block in developing UNCOL is the devising of expressions which pertain to formats of input-output data and designations of data movement. The content of such expressions is not wholly independent of machine design. The wide variety of input-output designs in machines may require a corresponding variety or flexibility in UNCOL expressions concerned with input-output.

A promising approach to the designation in UNCOL of input-output transfers (data movement) appears to be to mark or label each of the following:

(1) which data is read in;
(2) which data is put out;
(3) the earliest points in the procedure at which the input data could logically be moved into the computer: this will usually be at the very beginning of the procedure;
(4) the earliest possible points in the procedure at which the output data could logically be moved out of the computer: this will ordinarily be after one coherent unit of data, such as a page of print, or a complete display, has been made up.

Different input data may have differing "earliest input points," or times; similarly, different output data may have differing "earliest output points," or times. It will be the task of the UNCOL-to-ML translator to devise an appropriate set of input commands lying time-wise between the earliest possible points and the points at which the data is needed. It must also devise the appropriate output commands lying between the times the data is computed and the end of the procedure.

The formats of input data or output data, or both, may be a part of the program to be expressed in UNCOL. Such things as input card formats, displayed configurations, and arrangement of printed output pages may be an essential and fixed requirement on a program. Furthermore, they are not usually independent of any machine; they imply the presence of specific terminal equipment. There exists the problem of expressing these specific input-output formats in as general a manner as possible.

Severe difficulties may exist because of the differences in the sets of characters available at the various stages of translation from POL to ML. The sets of characters of a POL, an ML, and UNCOL itself may not coincide. The first difficulty is that, in translating a program from POL to UNCOL to ML, the translators must transliterate from one character set to the next. This transliteration problem may be compounded by the lack of needed characters on the computers in which the translators operate. The second difficulty is concerned with how UNCOL can represent a specific (POL) character which is not in the UNCOL character set. It might be done by substituting for the character its English name. This in effect maps any character not directly expressible in UNCOL into combinations of the 26 letters (and space). Alternatively, UNCOL might be provided with a very large character set embracing most of the characters which are likely to be met in programs. The third difficulty is that if the intent of a program to be translated is to produce an output at least partially composed of characters which are not available on the target machine, then it is questionable whether an acceptable translation is possible at all. For example, it may not be sensible to attempt to translate and run a program whose output is a table containing U.S. dollar values, on a machine which has no dollar sign.

Making UNCOL Extendible

It has become commonplace to say that a program language should be extendible: that is, it should be able to accommodate new expressions and new concepts. To what extent does it make sense to speak of UNCOL being extendible?

Extending a program language to encompass a new expression is done by defining a new expression in terms of the existing ones. Such a definition is constructed according to the rules of the language. It appears that the class of definition which may be made is limited to those specifically provided for by the rules. This limitation applies even if a new rule can be constructed (in accordance with the rules already extant). To put it another way, all expressions in a language are reducible by the language rules (syntax) to terms which cannot be further defined in the language (without using circular definitions). These undefined terms cannot be changed by expressions in the language. Hence, all provisions for constructing definitions are mere matters of convenience: they do not permit the expressions of concepts which could not be expressed without the definition.

While UNCOL can be made extendible in the sense just described: that is, of constructing definitions, what good is this? It can afford economy of expression for concepts which must be repeatedly expressed. It in effect permits a shorthand notation for use within UNCOL only. Without a corresponding modification in existing POL-to-UNCOL translators, such translators will not embody such shorthand expressions in their UNCOL output; future translators could, of course, make use of them. Similarly, without a corresponding modification in UNCOL-to-ML translators, any shorthand expressions will not be acceptable to such translators, hence the shorthand expressions will have to be replaced by their definitions (which are in undefined terms, terms that are understood by the UNCOL-to-ML translators). It would appear that the introduction of shorthand expressions (via the construction of definitions) in UNCOL will have no effect on a program translated from POL-to-UNCOL-to-ML, hence the value of such expressions is limited to a possible economy of notation in UNCOL itself.

The Nature of UNCOL

Let us consider briefly the possible nature of UNCOL as an intermediate language. Some of the possibilities are as follows:

(1) Some variation of Turing-machine language, suitably extended to include description of aspects of a program other than the internal computations.
(2) A machine language for some hypothetical machine having the characteristics of present-day general-purpose computers; again with some extension of the language to treat the non-algorithmic aspects of a program.
(3) A problem-oriented language, like ALGOL, which reflects to some extent the general nature of present-day computers.
(4) As broad as possible a problem-oriented language, perhaps like LISP (McCarthy, 1960; Woodward and Jenkins, 1961) which does not reflect to any significant degree the nature of computing machines.
(5) A language composed of the above-mentioned languages, thus having the capability of expressing program sequences at a variety of levels of abstraction.

It is this author's conviction that at least the algorithmic part of UNCOL will have to be able to represent a program segment at any one of several levels of abstraction (i.e. those mentioned as items 1, 2, and 4 above). The expression in languages above the Turing-machine level are short, conventional notations for those Turing-machine programs which we have found useful. (This is a relatively small percentage of the astronomical number of possible Turing-machine programs.) These higher-level expressions by no means include all the machine actions which could conceivably be useful. In other words, there is no way to describe some program actions other than to resort to Turing-machine language.

Why isn't extended Turing-machine language alone satisfactory as an UNCOL? While such a language might be theoretically adequate for expressing a program, there is a real and unanswered question as to whether a satisfactory translation could be made from it into a machine language. This translation process involves the discovering of a group of UNCOL expressions which corresponds to a yet smaller group of machine-language instructions. (Although it would be possible to simulate a Turing-like machine on any general-purpose digital computer, the execution of one or more computer instructions for each Turing-like machine instruction would result in gross and intolerable inefficiency.)

The Feasibility of UNCOL

Arguments as to whether or not UNCOL is feasible can be categorized into questions of *universality*, questions of *efficiency*, and questions of *analysability*.

As far as computations internal to a machine are concerned, any UNCOL which can express the basic computing steps of a Turing-machine is universal. It can then represent any computation that any deterministic computing machine (past, present, or future) can do.

It appears that UNCOL must represent at least some of the machine-dependent aspects of a program, such as input and output characters. If it cannot represent all the characters used in all present and future POLS, then UNCOL's universality is open to question. An UNCOL designed to accommodate those machine-dependent parts of programs expressed in current POLs, for current machines, might not encompass the capabilities of future POLs and machines.

It remains to be seen whether a specific realization of UNCOL can produce, in machine language, programs having satisfactory operating efficiency. Since life situations are rarely all black or all white, it is probable that UNCOL will provide satisfactory efficiency in some circumstances and not in others. For programs which in POL form are essentially machine-independent (and especially where the data can be described in machine-independent form), and for programs which can be readily separated into their machine-dependent aspects, the conversion of such programs into relatively efficient machine code should not be difficult. For programs which are originally expressed in a POL heavily dependent on a particular type of machine, it may not be possible to translate them into machine-independent form (and thence into machine code) without undue degrading of their efficiency of execution.

The success of UNCOL also depends on the ability of a translator mechanically to analyse into appropriate constituents a program expressed in a POL. We are unable to say at the moment what these appropriate constituents are, other than to note that the machine-dependent aspects must be distinguished from the machine-independent ones.

With our present state of knowledge, it seems inappropriate to state categorically that UNCOL is, or is not, feasible. It was the conviction of Holt and Turanski (1960) that UNCOL was "unrealistic and unworkable." It is likely that their conviction resulted from their having dealt primarily with programs that were heavily dependent (particularly in terms of the data description) on the characteristics of a specific machine. As we discussed, the success of UNCOL depends crucially on the ability to segregate the machine-dependent from the machine-independent aspects of a program. They perhaps felt that separating a programmed procedure into machine-independent and machine-dependent parts either was impossible or, if it were possible, would result in intolerably poor operating efficiency of the resulting program. In addition, they dealt largely with programs which taxed the capabilities of the machines they had available. Hence they probably felt that intolerable operating inefficiencies would almost inevitably be introduced by an UNCOL based on currently available techniques.

Steel (1961) attempts to ease the task of devising an UNCOL by limiting the scope of UNCOL to a set of current computers ("priority class machines"). He conceives UNCOL as a machine language for an "UNCOL machine" having the composite characteristics (but not the idiosyncrasies) of the priority-class machines. A source program in UNCOL form is that program in the machine language of the UNCOL machine. A realization of the program on some real machine is to be done by simulating the UNCOL machine on the real machine. The source program might be either compiled or interpreted for the real machine. (The interpreter in this case should be relatively simple since it is imitating the UNCOL machine on a real machine having similar characteristics.) This plan might indeed be workable for priority-class machines, but, as Steel admits, it will almost certainly be unsatisfactory for machines outside the priority class.

Development of UNCOL

Having talked about the general approach to an UNCOL, we must consider the next steps to be taken. The next steps appear to be these:

(1) to analyse the constituents of computer programs, in order to see what is the general nature of concepts to be expressed by UNCOL (the author has made some attempts to do this, but no results have been published);
(2) to collect or devise a representative set of programs to be used as a guide to developing UNCOL expressions;
(3) to enumerate the things which UNCOL must

express, based on the results of steps (1) and (2) above, as well as on a study of the capabilities of some current advanced programming languages;
(4) to suggest a suitable notation;
(5) to examine the problems of constructing translators.

It is by no means clear at this point that a satisfactory UNCOL as outlined in this paper is possible. There appears to be no way to decide whether this UNCOL is possible except to try to construct it. If an UNCOL can be developed along the lines presented here, it offers the distinct possibility of being a high-level programming language which has more capability and convenience than any of the current "advanced programming languages" such as ALGOL.

The author does not pretend to offer in this paper any solutions to problems concerned with UNCOL, but he hopes that he has made clearer the natures of some of the problems involved.

References

HOLT, A. W., and TURANSKI, W. J. (1960). "Man-to-Machine Communication and Automatic Code Translation," *Proc. of the 1960 Western Joint Computer Conf.*, p. 329.
IBM (1958). FORTRAN II Reference Manual for the IBM 704 Data Processing System.
MCCARTHY, J. (1960). "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I," *Communications of the A.C.M.*, Vol. 3, p. 184.
NAUR, P., *et al.* (1960). "Report on the Algorithmic Language ALGOL 60," *Communications of the A.C.M.*, Vol. 3, p. 299.
SHARE AD-HOC COMMITTEE ON UNIVERSAL LANGUAGES (1958). "The Problems of Programming Communication with Changing Machines," *Communications of the A.C.M.*, Vol. 1, No. 8, p. 12; and Vol. 1, No. 9, p. 9.
SIBLEY, R. A. (1961). "The Slang System," *Communications of the A.C.M.*, Vol. 4, p. 75.
STEEL, T. B. (1960). "UNCOL, Universal Computer-Oriented Language Revisited," *Datamation*, Vol. 6, No. 1, p. 18.
STEEL, T. B. (1961). "A First Version of UNCOL," *Proc. of the 1961 Western Joint Computer Conf.*, p. 371.
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

Microanalysis of Socioeconomic Systems, by ORCUTT, GREENBERGER, KORBEL and RIVLIN, 1961; 425 pp. (Harper Brothers, New York, $8.)

This work is in five parts. The first part is introductory and outlines the problem of building a satisfactory model of an economic system based on the social unit of a single individual. This is the problem considered in this book. The second part gives the statistical details of the demographic problem proposed, and the third part discusses some extensions to the problem.

The fourth part gives the details of computer programs to solve such problems, and has a very detailed appendix on the generation of random numbers, the generation of which is of great importance in the stochastic processes involved. The fifth part outlines some conclusions reached from experiments making use of the programs, and suggests further possible extensions to this field of research.

Demography is the study of the statistical behaviour of the population of a country. Here, at the micro-economic level, each person is defined by a group of data about such things as age, sex, and marital state, at a given moment. These basic units are then combined with other units representing other people on a semi-random basis, to produce a number of other larger family units. These family units are assumed to be capable of making decisions about the purchase of durable commodities, higher education, travel, and other economic problems. One of the main objects of the research is to simulate the behaviour of these decision-making units in the consumer section of the American economy.

The first step is to simulate the behaviour of these individual and family units by computer programs which imitate, on a stochastic basis, the birth and death of individuals, and marriage and divorce among families. Several chapters in the second section are devoted to the problem of getting reliable estimates for the probabilities that a given individual will be born, die, marry, or have a child in a given month. These probabilities are then used to set up a large sample for initial data, representing about five thousand families.

The simulation of the economic behaviour of these family units depends not only on these demographic features of a population, but also on their status in the total labour force, their hire purchase debts, their assets, their demand for luxuries, higher education, and travel. Simple models incorporating these factors are introduced in Part III.

Part IV is the section most interesting to professional computers. This section contains a general treatment of the simulation of these large-scale micro-economic models, using a very large high-speed computer. In this case, an I.B.M. 704 with four magnetic-tape decks was used. Full details are given, with flow diagrams of the actual course of the calculation, for this model of the consumer section of the U.S. economy, which was set up in the preceding sectors.

The connection with Markov processes is noted, but to bring the problem within practical bounds, a Monte Carlo process is used to generate the behaviour of any household at any given time. All households in the initial data are processed on magnetic tape, and this produces successive predictions of the future state of the population at monthly intervals.

The last section discusses further the problems of obtaining specific predictions from such micro-analytic models, by this approach based on simulation.

The work is invaluable to anyone concerned in the organization and automatic processing of demographic data, and should be of great interest to those engaged in building economic models of any kind, and to all those who have to build programs for projects with large quantities of data.

LUCY JOAN SLATER.