A FIRST VERSION OF UNCOL

T. B. Steel, Jr.
System Development Corporation
Santa Monica, California

Summary

UNCOL--UNiversal Computer Oriented Language--is being designed as an empirical, pragmatic aid to the solution of a fundamental problem of the digital data processing business: automated translation of programs from expressions in an ever increasing set of problem oriented languages into the machine languages of an expanding variety of digital processing devices. By application of a program called a generator, specific to a given problem language, program statements in this problem language are transformed into equivalent UNCOL statements, independent of any consideration of potential target computers. Subsequently, without regard to the identity of the original problem language, the UNCOL statement of the problem is processed by a program called a translator, which is specific to a given target computer, and the result is an expression of the original problem solution in the machine language of the desired processor. The advantage of this apparent complication over the current procedure of employing a program called a compiler for direct transformation from problem language to machine language is evident when one examines the number of languages and machines involved and the not inconsiderable expense of translation program construction. If there are M problem languages and N machines, then M·N compilers are required and only M+N generators and translators.

In order to arrive at sensible specifications for UNCOL, certain limitations in its scope are essential. Accordingly, UNCOL is designed to cope with only those problem language and machine language characteristics that can reasonably be expected to enjoy general use in the next decade. Any broader approach shows promise of leading to elegant, impractical results.

A glance at the preliminary specifications for UNCOL shows a language akin to a symbolic assembly language for a register-free, single address, indexed machine. The specific commands are few in number and simple in construction, depending on a special defining capability for the expression of more elaborate instructions. The data referencing scheme is complex, allowing the application of a signed index to the primary address, and permitting both the primary and index parts to be delegated to an indefinite level.

Each item of data, either input or calculated, must be described as to type, range precision and the like by special data descriptive syntactical sentences in the language. These descriptions, additionally, provide information concerning ultimate storage allocation as well as indicators of contextual meaning for the explicit commands.

Supplementary to the instructions and data descriptions are certain declarative sentences, inessential to the explicit statement of the problem solutions being translated, designed to provide information useful to the translator in the introduction of computational efficiency into the object program.

Introduction

One of the harsher facts of life in the data processing world, against which none interested in the effective operation of a digital computing installation dare wear blinders, is the existence of pressures toward diversity. Basic hardware always appears in a variety of shapes, sizes and configurations. New machines arrive on the scene with disconcerting regularity. Since 1952 the number of different types of computers built each year has oscillated about a mean of thirty, and subsequent to 1955 better than sixty per cent of these machines have been commercially built, general purpose devices available to any and all having the inclination and the money.[1] It follows that a problem of steadily increasing magnitude is found in the question of inter-machine translatability of programs.

At the same time, an expanding frontier of applications as well as growing sophistication of machine language has led to the proliferation of the "easy-to-code" problem oriented languages--POLs. Considerable difficulty is attendent to the employment of these languages in view of the effort necessary to maintain an adequate supply of current tools. It is well known that the task of producing translations from problem oriented languages to real machine languages is far from trivial. Until quite recently this translating was done by people--we call this "machine language programming." Now, however, with the exception of a few special cases not relevant to this thesis, the responsibility for performing these translations is being assumed by automation. As a result, requirements appear for processing programs--compilers--that are both expensive and time consuming to produce.

This capital investment in a translation program would be well advised were the concern solely for one problem oriented language and one machine language. However, as indicated above, there is a large variety of machine and problem languages. Thus the required investment increases multiplicity--one compiler for each pairing of problem language and machine language. In addition, the first derivative of development in both machines and problem languages is sufficiently positive that there is a strong tendency toward obsolescence prior to use and the day is not yet with us when programmers will write compilers on lunch hour

just for drill. A principal objective of the developments described below is a reduction in the time and money required to live with comfort in this dynamic environment.

UNCOL--UNiversal Computer Oriented Language--is as its name suggests, a language--occasionally referred to by the irreverent as an electronic Esperanto. Conceptually, UNCOL is a linguistic switchbox where problem oriented languages are transformable into UNCOL and, in turn, UNCOL is transformable into specific machine languages. If one is presented with M problem languages and N machine languages, M+N transformations are required in the UNCOL world, while M·N such transformation programs are necessary in the classical case. Elementary mathematical insight shows that when M and N are greater than two it is game, set and match to UNCOL. For the sceptic, Figure I is illustrative of the conventional situation and Figure II the UNCOL schema. A more detailed examination of this position, its rationale and its implications may be found in the literature.[2,3,4]

UNCOL is advertised as a practical solution to the problem described above and, as such, need not be defended against charges that it cannot handle this or that special case--usually one that was designed specifically for the discomfort of the UNCOL builders. A more legitimate indictment is that of unworkability in practice.[5] To date, debate on this matter has generated more heat than light. Indeed, this appears to be a question that is answerable only by appeal to experience. The empirical evidence for a meaningful UNCOL will come from a definition of the language and the subsequent development of transformation programs linking through UNCOL a sufficient variety of problem oriented languages and machines.

It is clear that the initial step in this program is the establishment of a trial language. In order to find a point of departure it is essential to circumscribe the problem and determine a bound beyond which this first version of UNCOL shall not penetrate. To this end a priority class of machines has been selected. This class may be roughly characterized as those general purpose digital computers having a capacity at least as great as an IBM 701 which are currently available or may be expected in a commercial version before 1968. The selection of 1968 as a cutoff point is based on an estimate of the validity of current projections of the characteristics of hardware with which UNCOL must cope. If one attempts to penetrate beyond this point all bets are off on the breadth of applicability of UNCOL as it is now conceived.

Advancement in the problem oriented language technology does not introduce the same difficulty. By designing UNCOL sufficiently similar to the languages of the available computers, it is clear that any problem language for which there is a known compiling algorithm can be put into UNCOL form by a generator program that simply performs this algorithm. If no such algorithm is at hand, then construction of a compiler is impossible in any event and the applicability of UNCOL becomes moot. It follows that the key questions revolve around the feasibility of constructing effective translators for the transformation of UNCOL statements into equivalent expressions in the several pertinent machine languages.

The point of view best adapted to providing an informal exposition of this first attempt at an UNCOL is to consider it in the light of an assembly language for a generalized machine. Thus we shall see that UNCOL has a capability for describing data both syntactically and semantically: i.e., with regard to both structure and meaning. There is a scheme for naming items of data analogous to conventional symbolic addressing and an associated indexing procedure permitting reference to data items standing in some ordinal relationship to a distinguished item. A set of verbs in the imperative mood provides a suitable collection of commands for the language. The objects of these verbs are data names. Verbs are of two kinds, primitive and defined, primitives corresponding to instructions in basic machine language and defined verbs playing the rôle that is assigned in conventional assembly systems to macro-instructions. Finally, there are declarative sentences whose function is the transmittal of information about the algorithm being stated. It is these declarative elements that hold the key to the introduction of efficiency in the object codes by the translation algorithm.

Syntax of Data Description

The internal representations of data vary so widely from machine to machine that any attempt to extract useful common description procedures is doomed to failure. Indeed, the occurrence of both binary and decimal machines precludes an approach of this type. A common factor, however, exists in another direction. All of today's computers, as well as any machine whose construction is sufficiently imminent to be relevant, communicate with the outside world in terms of linear arrays or strings of individual characters. While it is painfully true that no standard character set is in existence,[6] it is equally true that no difference in kind occurs from one set to another. As a consequence, the union over all known and contemplated character sets will yield a satisfactory universe of basic marks.

The guiding principle in establishing this universe of characters is the desire to permit an hypothetical computer to accept input and provide output in the normal orthography of any language, natural or artificial, that is in common use anywhere in the world for scientific, scholarly and commercial purposes, provided there exists the technological capacity to employ such a machine. Practical restraint imposes two limitations which cause the selected set to fall short of this ideal but the only measurable loss is esthetic. Those natural languages that are written ideographically or employ syllabaries must be excluded in order to

keep the size of the universe manageable. In the case of mathematical symbolism, a somewhat more ruthless action is called for. Typographical distinction between various levels of subscripting and superscripting must be limited in some manner as it can theoretically go to infinity. The most reasonable course seems to be an insistence on a shift indicator whenever the level is carried beyond the first. Failure to allow any geometry in symbolism is too restrictive and application of some limit above the first seems dreadfully arbitrary.

A synthesis of conversations with printers, (human type), random sampling of mathematical and scientific texts and examination of symbol lists such as are found in large dictionaries, has let to the development of the character set summarized in Table I. Occurrence of more than twenty-six letters in the various roman and italic alphabets is a consequence of the inclusion of compound marks such as letters with accents, the cedilla and the umlaut. One could hardly abbreviate the angstrom unit without the "Å".

UNCOL Character Universe

Alphabet	Number
Roman capitals	55
Roman small capitals	55
Roman lower case	56
Italic capitals	55
Italic lower case	56
Greek capitals	25
Greek lower case	28
German capitals	28
German lower case	30
Cyrillic capitals	35
Cyrillic lower case	35
Hebrew	23
International phonetic	45
Arabic numerals	10
Mathematical symbols	175
Commercial symbols	50
Punctuation	15
Special symbols	35
Grand total	811

Table I.

Allowing three size-position variants of each basic mark--main line, subscript, superscript--a total count of 2433 distinct characters obtains. This is not at all as wild as it might seem at first glance. The author has personally seen some printed outputs from various computers with 196 of the listed basic shapes. It is perfectly feasible to attach a device such as a Verityper to many of today's computers. It would be a good wager that the above set is conservative.

Every finite expression constructed as a linear array of the characters in the UNCOL universe is a conceivable input or output to a program described in UNCOL. In any given case, however, this input and output will be limited to expressions of certain well-defined forms, composed from a subset of the character universe. The nature of these legal forms and the character subset are a function of the problem language alone. In the event that the object computer does not possess the ability to handle the full character subset for either input or output, a transliteration must be established by the UNCOL to machine language translation program, according to predetermined rules, dependent only on the UNCOL character universe and the target computer. Thus, this character universe does for the subproblem of character incompatibility exactly what UNCOL itself purports to do for the general problem of language incompatibility.

UNCOL must have the capability to state definitions of character subsets and legal expression forms in structural terms. In order to accomplish this, a formal language, interpretable as syntax, is required. As a base this language includes the first order predicate calculus with identity.[7] Variables are interpreted as ranging over all finite expressions composable from the UNCOL character universe. Primitive names are specified for each of the characters and the operation of concatenation of expressions is introduced, symbolized by the arch, "⌢". Table II gives a summary of this notation.

∿	denial	not ...
∨	disjunction	... or ...
∧	conjunction	... and ...
→	implication	if ... then ...
↔	equivalence	... if and only if ...
⋀	universal quantification	for all ...
⋁	existential quantification	for some ...
=	identity	... equal to ...
≠	diversity	... not equal to ...
⌢	concatenation	... followed by ...

Table II.

The names of the characters are arbitrary. By taking them in some reasonable order, a number can be assigned to each character and these numbers can be used as subscripts to a general symbol to construct graphics for the names of the characters. For illustrative purposes and subsequent use in this paper, let us establish names for some small subset of characters, say a portion of the ALGOL character set. Table III is an example of a set of appropriate names.

Names for some ALGOL Characters

S_0	for "0"	S_{62}	for ","	S_{67}	for "10"
...					
S_9	for "9"	S_{63}	for "."	S_{68}	for "#"
S_{10}	for "a"	S_{64}	for ":"	S_{69}	for "+"
...					
S_{61}	for "Z"	S_{65}	for ";"	S_{70}	for "-"
		S_{66}	for ":="		

Table III.

Proper combination of the names listed in Table III above with the arch notation gives a mechanism for spelling. For example. the name of the expression "3.14159" in the structural-descriptive form envisioned here is:

$$S_3 \frown S_{63} \frown S_1 \frown S_4 \frown S_1 \frown S_5 \frown S_9$$

In addition to the ability to express names of explicit inscriptions, use of variables and the logical mechanisms outlined in Table II permits the characterization of kinds of expressions. By way of illustration, let us examine a series of definitions terminating in the explication of a form described in ALGOL as a number.[8] Recalling that variables, which will here be denoted by Greek letters, stand in place of the names of unspecified expressions, we have:

$$(\alpha B \beta) \quad \text{for} \quad \alpha = \beta \vee \bigvee_\gamma (\beta = \alpha \frown \gamma), \tag{1}$$

which is to be read as "the expression α begins the expression β" is defined to be "either α is equal to β or for some γ, α followed by γ is equal to β." Similarly, we have:

$$(\alpha E \beta) \quad \text{for} \quad \alpha = \beta \vee \bigvee_\gamma (\beta = \gamma \frown \alpha). \tag{2}$$

A digit is any one of the signs "0" through "9", so:

$$D\alpha \quad \text{for} \quad \alpha = S_0 \vee \alpha = S_1 \vee \ldots \vee \alpha = S_9. \tag{3}$$

It should be evident that no breach of rigor has occurred through the use of dots in definition (3) as the reader can fill in the missing clauses at will.

An unsigned integer is a string of characters each of which is a digit. Thus we have:

$$Ui\alpha \quad \text{for} \quad D\alpha \vee \bigwedge_\beta (\beta B \alpha \rightarrow \bigvee_\gamma (\gamma E \beta \wedge D\gamma)), \tag{4}$$

and it follows immediately that an integer is an unsigned integer with or without a prefixed sign.

$$I\alpha \quad \text{for} \quad Ui\alpha \vee \bigvee_\beta (Ui\beta \wedge (\alpha = S_{69} \frown \beta \vee \alpha = S_{70} \frown \beta)). \tag{5}$$

A decimal fraction is an unsigned integer preceded by a decimal point, viz:

$$Df\alpha \quad \text{for} \quad \bigvee_\beta (Ui\beta \wedge \alpha = S_{63} \frown \beta). \tag{6}$$

Similarly, an exponent part is

$$X\alpha \quad \text{for} \quad \bigvee_\beta (I\beta \wedge \alpha = S_{67} \frown \beta), \tag{7}$$

and by collecting the parts and performing the obvious, the definitions of decimal number, unsigned number and number follow immediately as:

$$Dn\alpha \quad \text{for} \quad \bigvee_\beta \bigvee_\gamma (Ui\beta \wedge Df\gamma \wedge \alpha = \beta \vee \alpha = \gamma \vee \alpha = \beta \frown \gamma), \tag{8}$$

$$Un\alpha \quad \text{for} \quad \bigvee_\beta \bigvee_\gamma (Dn\beta \wedge X\gamma \wedge \alpha = \beta \vee \alpha = \gamma \vee \alpha = \beta \frown \gamma), \tag{9}$$

$$N\alpha \quad \text{for} \quad \bigvee_\beta (Ui\beta \wedge \alpha = \beta \vee \alpha = S_{69} \frown \beta \vee \alpha = S_{70} \frown \beta). \tag{10}$$

The above example is indicative of the method one would use in describing the legal forms of data for any problem oriented language. Caution must be exercized in formulating such definition schemes, however. The mechanism outlined above has the capability to describe the syntactical structures of very powerful languages such as the logical language of "Principia Mathematica." In such languages non-constructable items occur and may be syntactically defined. For example, our scheme is capable of defining the notion of non-theorem, clearly non-constructable and even undecidable unless one assumes consistency. In order to circumvent trouble from this source, a set of rules for generator writers must be given which limits the complexity and character of the definitions; rules which the careful framer of definitions will follow instinctively.

Semantics of Data Description

The rather elaborate procedure outlined in the preceding section is sufficient to give the necessary syntactical description of data forms, but it provides no mechanism for introducing the meaning of the defined concepts. It is all very well for a translator to know that a particular juxstaposition of characters is given a specific name, say "number", but this information is of small value unless the translator is also aware of the appropriate mathematical and logical interpretation of this string of marks in whatever contexts it may occur. Provision of this information is accomplished by adjoining a semantic counterpart to the syntactical structure already at hand.

The basic principle involved in this scheme is quite simple. A new primitive operator is introduced which appears only in contexts where it is followed by the name of some expression. This structure is interpreted to mean "the meaning of the following expression." In addition, a notation capable of indicating the various different kinds of data upon which machine instructions can act directly must be provided. As the only types of data involved in computers in the UNCOL priority class are approximations to real numbers and binary patterns, this extra mechanism is relatively insignificant. It is now possible to relate, in a well defined manner, all structures described by the syntax to entities upon which the basic imperatives of UNCOL can operate.

In view of the desirability of having UNCOL versions of the various translator and generator programs, it is important that the semantical mechanism be able to handle the data of these programs which includes statements in the UNCOL language itself. By adopting appropriate rules of formation for the syntactical and semantical statements, it is quite possible to do this and avoid the ambiguities which seem fated to arise. A full explication of these rules is too lengthy for this paper and fragments of the rules are not very meaningful.

Referencing Scheme

The UNCOL referencing scheme--for those who prefer the analogy to a symbolic assembly language, the addressing scheme--is necessarily complicated, including an arbitrary level of delegation or indirectness. This last is essential in view of the existence of both problem oriented languages and machine languages having this kind of capability. For example, if the problem language being generated is a list structure language such as the various Information Processing Languages of Newell, Simon and Shaw,[9] and the elaborate scheme of pointing down a list to find a data item were to be interpreted out in the UNCOL statement of a problem, it would be virtually impossible for a translator to reconstruct this list structure for machine language. In the event that the object machine had the capability of mechanizing some portion of the list structure commands by hardware, the loss in efficiency of the resulting program would be unacceptable. As machines with this character are clearly feasible and under contemplation, UNCOL seems required to shoulder the burden of providing for them.

As many problem oriented languages are designed to handle ordered arrays in a particularly straightforward manner, it is essential that UNCOL have an indexing capability in its referencing scheme. For reasons analogous to those outlined above for multi-level delegation, it is desirable to permit indexing the same freedom.

In order to display the entire scheme some notation is required. We denote basic names by capital roman letters. These serve as names for both primary and index variables, as no distinction is made between these types of entities in data description. We enclose the index name in parentheses and append it as a suffix to the primary. Delegation is indicated by an asterisk.

Delegation of addressing can occur in three basic forms and all combinations of these forms is possible. These are: (1), indexing prior to delegation; (2), indexing subsequent to delegation, and; (3), delegation of the index. If there is no indexing specified, then delegation at the initial level is of the simplest kind. In addition, the delegation may be modified by a level limit, denoted by an appropriate integer following the asterisk indicating delegation. This limit is interpreted to mean the number of levels to which the delegation is carried, independent of the indicated delegation of intermediate names. When delegation has been carried out, willy-nilly, to the indicated level, the delegation process proceeds normally from there unless the original name was marked by a limit modifier, denoted by a prime following the limit integer. In this latter case action is terminated regardless of the status of the name at the given level. Implicit in the above description is the fact that any element to which addressing has been delegated is taken to include all its modifiers as well as the associated index name.

When a string of delegated elements is linked together; i.e., when a delegated element points to a delegated element, which points to a delegated element, etc., a principle of postponement is applied--you step down one more level when required and you do not step back up until everything below is specified. As each name may have an associated index, this leads to a tree searching procedure which terminates only when the tip of each branch is reached.

The following examples will clarify the situation. A(I) means "take A and apply I." A*(I) means "finalize A and then apply I." A(I)* means "take A, apply I, and finalize the result." A*(I)* means "finalize A, apply I, and finalize the result." The index variables themselves may be delegated in the same manner and with the same freedom. Thus, A*(I(J*(K)*)*)* is an allowable combination.

The level limit, which may also be complex in structure, has a clear interpretation. For example, the expression A*5 means "delegate exactly five levels and treat the resulting element as a new address." On the other hand, if a limit modifier is applied, the situation is as follows. A*5' means "delegate exactly five levels and treat the resulting element as the finalized address."

Table IV below gives a set of detailed examples of the UNCOL referencing scheme. In each case, the value called "result" is the ultimate quantity to which the entire schema refers.

UNCOL Referencing Scheme

Name	Value	Name	Value
a	A*	a	A*(I*)*
A	B	I	J
B	result	J	K
		A	B
a	A(I)	B+K	C
I	J	C	result
A+J	result		
		a	Z(V*)*
a	A*(I)	V	U*(K)
I	J	K	P
A	B	U	W
B+J	result	W+P	X
		X	Y
a	A(I*)	Z+Y	T
I	J	T	L*(M)
J	K	M	3
A+K	result	L	N
		N+3	R
a	A(I)*	R	result
I	J		
A+J	B		
B	result		

Table IV.

Imperatives

The fundamental imperative verbs of UNCOL--the instructions of the language--are quite elementary and their meaning will be obvious at once to anyone who has coded for a single address digital computer. It should be observed, however, that no registers of any kind are implied by the command structure, contrary to a conventional machine language, although it would seem at a quick glance that there is at least an analogy to an accumulator. Specification of such a register is necessary only when questions of precision are implicit in the meaning of the command. In UNCOL these questions are answered in the data description sentences, not in the instructions. For ease in studying the situation, however, a generalized and exceedingly pliable accumulator-like gadget may be assumed without creating a difficulty.

An element of context enters into the meaning of the instructions. As is the case with conventional machinery, the instructions are presumed to follow one another in a linear sequence. This is so obvious that it is often overlooked in an analysis of the situation. Because of the capability for modification of data context, it is quite important to keep in mind this sequential flow.

The exact meaning of certain instructions, such as the arithmetic commands, are clearly dependent on the context of the data. Thus a translator for a machine having both fixed and floating arithmetic instructions would have to examine the description of the data called for by the address of the UNCOL command in order to determine which kind of machine language instruction to employ. In addition, each arithmetic command can be tagged with indicators, telling the translator whether the command is to be interpreted as employing the data direct, taking the absolute value (where relevant), or complementing the data first.

With the above preliminaries, the following list of basic UNCOL commands should be easily understood.

TAKE: obtain the value by the address.

ADD: compute the sum (arithmetic or logical as the case may be) of the result of the previous instruction and the value named by the address.

SUBTRACT: compute the difference (arithmetic or logical) between the result of the previous instruction and the value named by the address.

Notice that in both arithmetic instructions the actual operation performed is a function of the data description. In the event that a translator should encounter an instruction and data description combination which implies a mixture of incompatible data, an error has occurred, either in the original problem language statement of the problem or, worse yet, in the coding of the generator. Such a situation would be quite unsafe if UNCOL were being conceived as a language in which humans would code, but as things are it would not seem to be too serious.

The meaning of the commands MULTIPLY and DIVIDE should be evident by analogy with the above descriptions, once the meaning of logical division is explained in some satisfactory fashion. The command REMAINDER permits the gathering of the remainder upon division. It needs no operand.

The command REPLACE is the analogue of the "STORE" instructions of conventional machines. It provides a facility for altering the value which is assigned to a given variable.

All decision making in UNCOL form will be reduced to COMPARE, a command with two objects or operands. These two operands are compared, first to second, and the result of the comparison is remembered--in a real machine this would be by setting a toggle.

In order to implement the branching function, it is necessary to give the UNCOL imperative statements identifiers. These names are, however, distinct from any data names and cannot be used as the operands of arithmetic instructions. No instruction can be modified by an UNCOL statement. Were this permitted chaos would reign. It would become trivial to demonstrate the existence of untranslatable sequences.

Certain instructions--branching instructions of both unconditional and conditional variety--have as their operands the names of UNCOL statements. There are two unconditional branch instructions. The more complex of these, ENTER, is described below. The other, GOTO, is a simple unconditional branch to the statement whose name is the operand of the GOTO statement.

The conditional branch commands are all based on the remembered result of a COMPARE. They are:

IF $<$ GOTO IF $=$ GOTO IF $>$ GOTO

IF $\leq$ GOTO IF $\neq$ GOTO IF $\geq$ GOTO

The meaning of these should be evident.

The ENTER command, together with its complement RETURN, is used for the purpose of transferring control to a subroutine and keeping a record of where the entry occurred. It employs a list, called the "entry list", which must be established by the translator. This list contains the statement names of each ENTER which has been encountered on a last-in-first-out basis. Upon completing the action of a subroutine, a RETURN is given and the appropriate point of return is determined by examination of the last item on the entry list. This procedure permits a subroutine to use itself.

Also used with ENTER are the commands WITH and RESULTS. These follow the ENTER and allow the specification of input and output parameters for subroutines. Use of an additional modifier for the referencing scheme--a relative delegation indicator--permits the delegation of an address relative to the current head of the entry list. Then the list is stepped by one on a temporary basis (for the duration of the address determination) and delegation proceeds. This permits a parameter to be called from the top of a nest of subroutines, which is perhaps recursive, without requiring the asking routine to know anything about the nature of the nest. This is quite important for the proper treatment of certain problem oriented languages.

There are cases, particularly in problem languages designed for writing compilers and the like, where a given data item is operated upon in more than one context by the same program. In order to allow an override of the context that is specified by the data description, an instruction for explicitly determining the context is provided. It is SET CONTEXT and its object is a code specifying context. One of these codes will be called "normal" so that a return to the usual situation is possible.

The UNCOL commands are rounded out by a pair which open Pandora's Box wide! They are DEFINE and END DEFINITION. These serve as a pair of brackets that enclose definitions of complex instructions in basic UNCOL terms. The purpose here is analogous to that found for the use of macro-instructions in conventional assembly programs, but its application is inverted. In the usual case there complex instructions are designed to improve the source language program by making it more compact and easier to construct. In the UNCOL case it is the object language that is the prime beneficiary. These definitions permit a translator to recognize units of program larger than one command and, if the object machine has an instruction for that function, considerable gain in efficiency may be obtained.

Declaratives

Not much can be said about the declarative sentences of UNCOL at this stage. Their general character will be similar to such statements as the following: "The next 20 UNCOL statements form a loop." The information content of these declaratives will be largely a function of the availability of flow information in the problem oriented language. In order to determine the proper kind of statement to include, it is necessary to construct some experimental translators and discover empirically what questions the translator would like to have answered in order to introduce efficiency in the object code.

Work currently under way in an effort to plan and flow chart translators to a variety of different machines with objective of uncovering suitable declaratives. Perhaps six months is not too soon to expect some results.

Recapitulation

In the phrasing of H. G. Wells we observe, "The past is but the beginning of a beginning ..." The version of UNCOL outlined in the preceding paragraphs is the result of extensive cogitation and conversation but its only justification will be the test of workability and that is still to come. Perhaps the whole approach is doomed to failure and oblivion, and then again, perhaps not. One thing at least is sure; Experience and not dialectic will be the judge.

References

1. Bourne, C.P., and Ford, D.F., "The Historical Development, and Predicted State-of-the-Art of the General-Purpose Digital Computer," Proc. of W.J.C.C., 17 (May 1960), 1-21.

2. Mock, O., et al., "The Problem of Programming Communications with Changing Machines: A Proposed Solution," Comm. A.C.M., 1:8 (Aug 1958), 12-18; 1:9 (Sept 1958), 9-15.

3. Steel, T.B. Jr., "UNCOL," Datamation, 6:1 (Jan Feb 1960), 18-20.

4. Steel, T.B. Jr., "UNCOL: The Myth and the Fact," Rev. in Automatic Programming. (in press).

5. Holt, A.W. and Turanski, W.J., "Man-to-Machine Communication and Automatic Code Translation," Proc. of W.J.C.C., 17 (May 1960), 329-339.

6. Bemer, R.W., "Survey of Coded Character Representation," Comm. A.C.M., 3:12 (Dec 1960),639-642.

7. Hilbert, D. and Ackermann, W., "Mathematical Logic," Chelsea, New York, 1950.

8. Naur, P., et al., "Report on Algorithmic Language ALGOL 60," Comm. A.C.M., 3:5 (May 1960), 299-314, esp. #2.5.1.

9. Shaw, J.C., et al., "A Command Structure for Complex Information Processing," Proc. of W.J.C.C., T-107 (May 1958), 119-128.

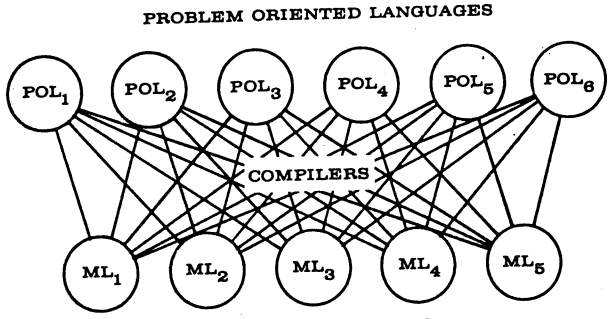

FIGURE I

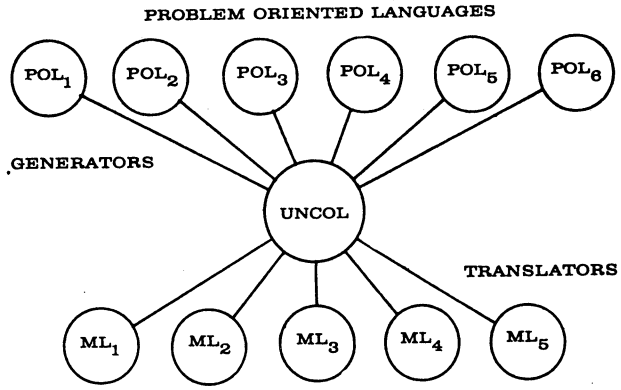

FIGURE II