TECHNIQUES DEPARTMENT

Dear Bob:

This letter is being sent to you both as Techniques Editor of the *Communications* and as an interested party. I found the POL-UNCOL-ML report in the August issue very interesting since we have arrived at just about the same conclusions. Furthermore I have developed an UNCOL, a version of which we will use when we put the RUNCIBLE successor on the Univac. The enclosed paper describes the language in the jargon of the August report.

The similarity of the language with Polish notation has led us to begin a rigorous characterization of it. It seems that one might fruitfully define transformations on strings in the language to optimize a space-time functional for a given machine (e.g., the "653 function" which will tolerate a few more instructions here and there if the result is less execution time through greater use of the core storage).

I submit the paper for publication and for your comments.

Sincerely,
Mel

PROPOSAL FOR AN UNCOL

MELVIN E. CONWAY, Case Institute of Technology

This discussion contains a proposal for a universal computer-oriented language (UNCOL) to be used as a common path between the problem-oriented language (POL) and the machine language (ML) in compatible automatic programming systems. (See Ref. 1.)

The totality of desired characteristics of a useful UNCOL cannot be completely known because of the possible emergence of a machine whose programming method is basically different from those currently used, but at the present time at least, a computer-oriented language is one which describes a computation in terms of a sequence of operations, each of which operates on information stored in machine registers or alters the sequence of operations.

The language described herein has been motivated by practical considerations[1] in the programming of RUNCIBLE[2], an existing IT-to-650 compiler, for the Univac 1. The memory size of the Univac 1, on which the compilation will take place, demands that there be at least one intermediate language which can be stored on tape during the compilation process. Experimentation with several intermediate languages has indicated that the one which provides the greatest opportunities for subsequent economization of machine code resembles the code of a single-address computer, and the simpler this computer (i.e., the fewer the registers and instructions available) the greater are the opportunities to translate this intermediate code into tight machine code. This happy fact implies that universality of the intermediate language (in the sense that it is not biased toward any computer) goes hand-in-hand with efficiency of the resultant machine code. Therefore the proper design of an UNCOL will not only aid intercommunication among machines but will result in the production of better programs.

The technique of programming in this UNCOL, which I shall call SML (simple machine language), is to reduce a computation to a sequence of minimal arithmetic or logical operations, to transfer the arguments of each operation to standard storage positions, and to express each operation as a subroutine linkage ("execute"). There are only two instructions in SML, a memory-to-memory transfer and an execute instruction.

Programs refer to four types of memory: general (G-) storage, temporary (T-) storage, argument (A-) storage, and result (R-) storage. These are distinguished by the roles they play in the computation.

Variables which appear in the POL and constants are held in G-storage. Hence the operation Y←—(B*C)+D finds B, C, and D in G-storage and stores the result Y there.

Intermediate results in a given calculation which have not been given a name in the POL are stored in T-storage. For example, if the sequence of Y←—(B*C)+(D*E) is

1. Execute B*C
2. Store the result
3. Execute D*E
4. Execute the addition
5. Store the result in Y,

then the store operation of step 2 will be into T-storage.

A-storage holds the operands (arguments) of all operations. Hence "execute B*C" assumes that B and C have been put into A-storage.

The result of every manipulative operation (as distinguished from a comparison operation which has no "result") is found in R-storage. Although existing POLs do not demand it the possibility of multiple output operations is included in the language.

Each operation is preceded by the memory-to-memory transfers necessary to load A-storage and succeeded by the memory-to-memory transfers necessary to empty R-storage. The storage registers (except for G-storage) are denoted by the Roman letters T, A, or R subscripted by a non-negative integer. In SML a memory-to-memory transfer is represented by a pair of memory locations. For example, (T_0, A_1) means "read out of T_0 into A_1" or "T_0——→A_1" and is conventionally coded as "load T_0, store A_1".

Addresses in G-storage are given any name desired except, of course, for names like T_0 and A_1. If appropriate, these names will be taken from the POL-coding. In addition there is a class of G-addresses denoted by a star (*) whose name indicates the contents of the memory location. For example, (*1.0, A_1) means "read a floating-point 1 into A_1". Starred G-addresses may appear as the left member only of a transfer pair.

An operation is denoted by the letter "X" (execute) followed by parentheses enclosing the name of the operation: X((A_0)*(A_1)——→R_0, floating point). In order to reduce the number of possible operations there are conventions on the use of A- and R-storage. The n arguments of an n-ary operator will be placed into the set (A_0, . . ., A_{n-1}) and the correspondence is specified by the usual order in which the arguments of the operator are written. The results are similarly lined up in R-storage. For example the numerator of a divide goes into A_0 and the denominator goes into A_1. With these conventions the above example is equivalent to X(floating point *).

Some readers may have been disturbed by the fact that the allocation of G- vs. T-storage depends on the POL. The simplification rule stated below removes part of the objectionableness of this practice, in that superfluous temporary storage not mentioned in the POL will not appear in the UNCOL. The other half of the problem is more difficult and indeed arises whenever one tries to describe a given computation equivalently in two different languages: since many POLs may specify too much temporary storage (e.g., any three-address system, or even a Fortran-type language with too-finely-divided statements) how can this inefficiency be kept from getting into the UNCOL? The answers to this question are to be found partially in the design of the POL and partly in the algorithms of the POL-to-UNCOL generator, which should perform the currently avoided task of scanning a program as a whole for redundancies.

Certain seemingly obvious simplifications of SML must be avoided, whereas others are acceptable. The criterion for acceptability of a simplification rule rests on the way it will complicate the algorithms for conversion of SML into ML. For example, if Y←——(B*C)+D is scanned from the right the corresponding SML program is

(D, T_0)
(C, A_1)
(B, A_0)
X(*)
(R_0, A_0)
(T_0, A_1)
X(+)
(R_0,Y).

One acceptable simplification rule eliminates all transfers of the form (G-address, T_k) by replacing the subsequent (T_k, A_n) by (G-address, A_n). In our example the SML program resulting from such a simplification is

(C, A_1)
(B, A_0)
X(*)
(R_0, A_0)
(D, A_1)
X(+)
(R_0, Y).

The following example illustrates that using the above rule for R-address (rather than G-address) would be an unacceptable simplification:

Y←—|B|+|C| should generate

(C, A_0)
X(magnitude)
(R_0, T_0)
(B, A_0)
X(magnitude)
(R_0, A_0)
(T_0, A_1)
X(+)
(R_0, Y).

The sequence

(C, A_0)
X(magnitude)
(R_0, A_1)
(B, A_0)
X(magnitude)
(R_0, A_0)
X(+)
(R_0, Y) is incorrect for two reasons:

1. on a given object machine A_1 may be a register which is destroyed by the magnitude operation, and
2. the algorithms for translation from SML to ML are greatly simplified by the assumption that preceding each n-ary operation there will be precisely n transfers of the form (−, A_k); (k = 0, . . ., n − 1). Such mistakes can be avoided by adopting the rule that A- and R-storage are not usable across an operation.

So far we have discussed only replacement statements (equations). SML must contain several other types of information. Somewhere (probably at the beginning to simplify the job of the SML-ML translator) there should be a list of storage requirements for arrays. Certain points in the SML program should be tagged (just as statements have their numbers) and the tag should carry information as to whether or not it will be used as an entrance point into the instruction sequence. Finally, there are certain operations of the "jump" type: "jump to the point in the program whose tag is in A_0 (if $A_1 = A_2$)". The quantity in A_0 may be the result of a calculation or it may be a name which was carried over from the POL. In the latter case there may be good reason not to carry around the statement names in the ML program: the SML-ML translator will have to make such decisions. There are other operations all of whose arguments may not appear explicitly in the ML program; "get the doubly-subscripted variable whose name is in A_0, whose matrix width is in A_1, and whose subscripts are in A_2 and A_3, such that the subscripts of the upper-left-hand element are in A_4 and A_5"; "get the next item in the serial input file whose name is in A_0". The latter implies the existence of some file designs at the beginning of the program. When a name is put into A-storage it comes via a transfer from starred G-storage: (*input file 3, A_0). Thus the translator can go back in the instruction list and find out the name.

The methods used to turn the POL into SML of course depend on the input, but this much can be said: because of the regularity of SML the job will be simpler than that of turning our symbolic coding for an existing machine.

More can be said about the SML-ML translator. The way we are programming RUNCIBLE for the Univac, the translator will consist of two parts: 1) SML to inefficient ML; 2) inefficient ML to good ML. (ML here means symbolic ML, UNISAP[3].) The "inefficient ML" will work but will be wasteful of space. The chief job of part 1 is to decide what operations can be done by the hardware and do not require a reference to the library, and to translate, for each operation, A- and R-storage into appropriate machine registers. Part 2 keeps a table of the contents of the machine equivalents of A-, R-, and T-storage and weeds out about as much coding as a human programmer could, subject to the restriction that he may not permute operations.

I have tried to describe in rough terms an UNCOL, whose suitability for describing arithmetic computation will be ascertained in our current flow-charting of a compiler which uses it. The suitability of SML for *all* computation is open to question, but assuming some kind of communication (regarding what are legitimate operations) between the POL-SML phase and the SML-ML phase, the complete generality of SML is very plausible.

Notes

1. Share Ad-Hoc Committee on Universal Languages, *The Problem of Programming Communication With Changing Machines*, Communications of the ACM, Vol. 1, No. 8, August 1958.
2. Computing Center Staff, *RUNCIBLE 1*, Computing Center No. 1008, Case Institute of Technology, August 1958.
3. Conway, M. E., *UNISAP*, Computing Center No. 1009, Case Institute of Technology, August 1958.

ON THE EQUIVALENCE AND TRANSFORMATION OF PROGRAM SCHEMES

IU. I. IANOV

Doklady, AN USSR, vol. 113, No. 1, 1957, pp. 39–42
Translated by MORRIS D. FRIEDMAN, Lincoln Laboratory

Editor's Note: This is a translation of the first of two related Russian articles. The second article will be published in a later issue of the COMMUNICATIONS.

When programming for universal automatic computers, (logical) program schemes (PS) are used [1]. Inasmuch as a PS is not determined uniquely by an algorithm, questions of the equivalence and also of the identical transformation of the PS arise. In this note, the PS are considered as specific listings of the order of completion of the operators and of the logical conditions depending on the values of the logical (binary) variables.* Hence, the operators are considered as elementary objects to which a specific capacity to alter the values of the logical variables is attributed.

DEFINITION 1. The symbols $|\_\!\__i$, $\_\!\_|_j$ with the natural subscripts i, j will be called the left and right strokes, respectively. We will call the ordered set of the logic-algebra function α and the left stroke $|\_\!\__i$ the logical condition $\alpha|\_\!\__i$. We will call the operators $A_1, A_2 \ldots$ and the logical conditions, elementary expressions (EE).

DEFINITION 2. The finite line composed of operator symbols $A_1, A_2, \ldots,$ logical conditions $\alpha|\_\!\__i, \beta|\_\!\__j, \ldots$ and right strokes $\_\!\_|_i, \_\!\_|_j, \ldots$ such that one and only one right stroke $\_\!\_|_i$ with the same subscript i is found in this line for each left stroke $|\_\!\__i$ with the subscript i which enters into this line and, conversely, one and only one left stroke with the same subscript is found for each right stroke $\_\!\_|_i$, is called a program scheme.

We will consider that each entry of the operator into the PS is independent.

* Assuming the values 0 and 1, where let 0 correspond to "false" and 1 to "true".