THE ALCOR PROJECT

K. Samelson and *F. L. Bauer*

Gutenberg University, Mainz, Germany

ALCOR (meaning ALgol COnverteR) is the name of a cooperative group of computing centers, and computer manufacturers. The members of this group have agreed to use the same restricted version of ALGOL, based on essentially the same methods of translation. For coding, a single hardware representation (5 channel paper tape) is used. In this way exchangeability of programs for different machines is insured.

The history of the ALCOR group is intimately connected with the development of ALGOL. In a certain sense both could be said to begin with Rutishauser's paper on automatic programming (9) first published in 1951, which for the first time gave a systematic, although somewhat complicated, approach to formula translation. It came too early, however, and remained unnoticed.*

Stimulated by Rutishauser's paper, the computer group of the TH Munich in late 1955 started studying formula translation. This led to the design, by Bauer and Samelson (1), of a formula controlled computer with a very simple and effective control mechanism.

A diagram of the simplest form of this machine is given in fig. 1. The input language, in ALGOL terminology, consists of simple arithmetic expressions (without exponentiation) with numbers only as operands, and with *right* hand side assignments like $(a+b) \times (-a+b) \Longrightarrow e$.

Characters are written on the keyboard 1 and produce coded signals to the predecoder 5. This separates numbers from operators $\times : + - ( )$.

Numbers are sent, as they come in, through numbers converter 7 into the numbers cellar 11. A cellar here is a store of the type which lately has been called pushdown-store or stack, working on the "last in first out" principle.

Operators go to operations converter 8. There, each character coming in from the predecoder is compared with the topmost element in operations cellar 12 (initially empty). The evaluation

*The same happened to two other papers on essentially the same subject which ought to be mentioned here: Lehmann 1953 (7), and Bohm 1954 (4).

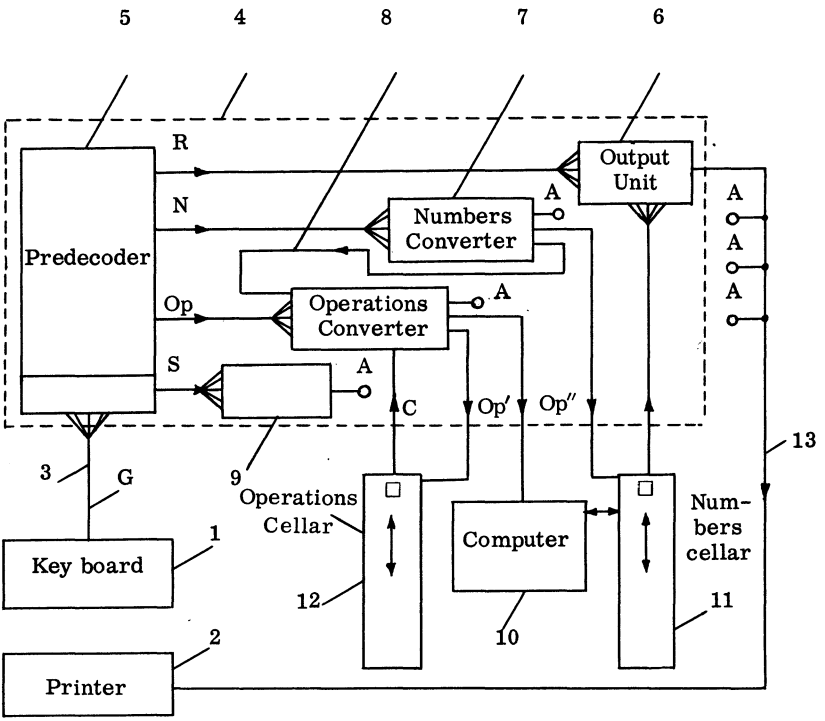

FIG. 1.

is done by means of a decoding matrix. The effects can be described in terms of precedence orders of the characters involved as listed below:

Order	incoming characters	cellar element
0	(	
1	× :	× :
2	+ −	+ −
3	)	(

The closing parenthesis never enters the cellar.

If the incoming character has lower order than the cellar element, it is simply pushed into the cellar. If the order is equal, the cellar element is transmitted to the computing unit 10 for execution, and is replaced in the cellar by the incoming character. If the incoming character has higher order than the cellar element, the latter is again transmitted to the computing unit for

execution. At the same time, contents of the operations cellar move up, and the process of comparison is repeated with the new cellar element. Closing and opening parenthesis simply cancel out upon meeting.

Execution of an operation means that the computing unit extracts the two top elements of the numbers cellar, executes the operation indicated with these operands, and returns the result to the numbers cellar.

An example is given in fig. 2. Here the top field gives the characters of a statement processed from left to right. The two fields below give the corresponding contents of operations cellar OC and numbers cellar NC. c and d denote the intermediate results, o denotes the number zero which is introduced at the beginning of expressions in order to take care of possible unary operators + or −.

The system described can be extended (and has been on paper) in quite a number of ways which we need not discuss in detail. The machine as designed immediately interpreted and evaluated formulae. However, it was clear from the beginning that the process could be adapted easily to generate programs simply by being applied to the names (addresses) of numbers instead of the numbers themselves, and producing instructions instead of executing operations, with the numbers cellar for intermediate results added to the generated program.

The basic principle remains unchanged: The characters of the program are sequentially compared to the cellar elements and moved into the cellar until a complete (explicit or implied) bracket structure is detected. This is extracted, processed, and replaced

S 1	S 2	S 3	S 4	S 5	S 6	S 7	S 8	S 9	S 10	S 11	S 12	S 13	S 14	S 15	S 16	
(	a	+	b	)		×	(	−	a	+	b	)		→	e	
⊠	(	(	+	+	(	⊠	×	(	−	−	+	+	(	×	⊠	⊠
	⊠	⊠	(	(	⊠		⊠	×	(	(	(	(	×	⊠		—OC
			⊠	⊠				⊠	×	×	×	×	⊠			
									⊠	⊠	⊠	⊠				
o	o	a	a	b	c	c	c	o	o	a	−a	b	d	d	e	o
⊠	⊠	⊠	⊠	a	⊠	⊠	⊠	c	c	o	c	−a	c	c	⊠	⊠
				⊠				⊠	⊠	c	⊠	c	⊠	⊠		—NC
										⊠		⊠				

FIG. 2.

by the result of the processing. Then the scanning is continued.

With this translation principle as a background, the ZMMD group (Zürich-Munich-Mainz-Darmstadt), the nucleus of the later ALCOR group, was set up in 1957 to develop a common language, translation philosophy and translation method.

The group was in agreement from the beginning, partly as a result of the experiment in machine design, to devise a language and translator system as an efficient programming tool using available machine facilities to the fullest extent possible. This ruled out any kind of interpretive system, in particular for floating point machines like PERM and ERMETH, and indicated a very judicious use of closed subsidiary subroutines. The target was, and always has been, an easy mathematically oriented language, and fast translation into efficient machine programs.

After publication of the first ALGOL report, in the fall of 58, three people of ZMMD institutes (Schwarz from Zürich, Seegmüller from Munich and Paul from Mainz) were brought together to program a first test model of ALGOL translators for their respective machines (ERMETH, PERM, Zuse Z 22). These first models were interpreters because programming was considered to be simpler. They were finished by the end of 58, were running in January 59, and were soon converted to program generators.

At the ICIP conference in Paris, June 1959, a report on the group's translation methods was given in the symposium on automatic programming (2). This marked the start of the expansion of ZMMD into ALCOR. Several computing centers (Regnecentralen Copenhagen, University of Bonn, Mailüfterl Vienna, Oak Ridge National Lab) showed interest, and decided to join ZMMD on a cooperative basis. They were given the available ZMMD material on translation methods, and began their own work on translators. The first manufacturer, Siemens & Halske AG, soon followed suit.

The cellar method outlined above for arithmetics proved to be suitable for ALGOL in general since practically all delimiters, apart from their operative meaning, implied a bracket structure amenable to the cellar treatment.

The essential feature of this method is, as mentioned in publications (2,11), that each in-coming character is matched with the topmost cellar element, and that the pair determines the ensuing action. The cellar system therefore can be considered to be a kind of automaton, where input (incoming character) and state (cellar element) determine new state and output. Originally a

decoding matrix, with incoming characters on the one side, cellar elements on the other side, as inputs, was used to describe the choice of action for each pair.

In the early days, especially when the figures on the size of the FORTRAN I translator, and its translation times on the fast IBM 704 became known, it was thought that on the relatively small machines at the disposal of the group decoding speed was essential, and a programmed decoding matrix or multiswitch seemed to be the fastest method. For this reason, and on account of its simplicity, ZMMD descriptions of the cellar method in 59 were still in terms of this matrix which had become rather outsized. The publications (2,11), however, which barely mentioned the matrix, should have made clear that the manner of decoding was a question of programming expediency for particular machines, and not a matter of principle.

In fact, all ALCOR members in 59 started looking for ways to reduce the matrix in size, with mostly similar results which only in part were published. First to be mentioned is A. A. Grau (6) who explicitly introduced the concept of syntactic states, and pointed out that the system could be considered to be a set of mutually recursive subroutines. Lucas (8) tried to deduce the translator mechanism from the syntactic structure. Paul and Petry independently introduced the direct use of the precedence order for decoding described at the beginning (unpublished) which can be extended to practically all operative delimiters with the precedence ordering shown in Table I.

The translator mechanism hitherto described (and including resolution of the for statement and conditional statement and expression by means of simpler ALGOL elements) serves to decompose ALGOL statements into a bracket free ordered sequence of instructions of fixed format (e.g. three address or single address instruction).

With a high degree of justification, we could call this the end of ALGOL translation. For the output of decomposition is a fully sequential language of simple macro-instructions. This language could serve as input language for an interpretive system or, alternatively, for a standard compiler system generating machine programs.

In both cases the processing system has nothing to do with ALGOL. However, in a translator designed to produce straight machine programs such a division, although simplifying programming by clearly separating tasks, would be uneconomical.

TABLE I

Order	Incoming character	Cellar element
0	(| [| := | if | for | goto | array | switch | procedure | value | '' | begin | segment begin |	
1	↑	↑
2	× | /	× | /
3	+ | −	+ | − | $-_1$
4	< | ≦ | = | ≠ | ≧ | >	< | ≦ | = | ≠ | ≧ | >
5	¬	¬
6	∧	∧
7	∨	∨
8	⊃	⊃
9	≡	≡
10	) |] | , | : | ; | then | else | step | until | while | do | end.	(| procedure call | standard function call |] | := | if | thenE | thenΣ | elseE | elseΣ | for | for:= | step | until | while | do | goto | begin | begin Δ | begin array | segment begin | array | array[| array: | switch | switch:= | procedure | type procedure | procedure(| type procedure(

Furthermore, what comes after decomposition makes up the bigger part of the detail work to be done for the individual machine. Therefore, we shall consider it briefly.

As mentioned before, operations required in the macroinstructions usually are not directly available in machines, and have to be translated into machine terms. This begins with arithmetic operations on different number types. Type handling very much depends on the machine, and within the ALCOR group different methods are used. The extremes probably are the

translator ALCOR Z 22 on one side, ALCOR S 2002 on the other side.

On the Zuse Z22, arithmetic operations must be done by subroutines. Therefore types are handled dynamically: every quantity stored carries a type characteristic, and the arithmetic subroutines evaluate this characteristic, carry out the operation accordingly, and attach the correct characteristic to the result.

The Siemens 2002 is a floating point machine, and real type arithmetics is naturally identified with floating point arithmetics. As for integers, after long discussions we came to the conclusion that for us they were most important as subscripts. Therefore they are internally identified with machine addresses which necessarily means fixed point numbers. Thus we have two different types of arithmetics which are incompatible, and type handling has to be completely static. This means that the type analysis has to be done completely during translation, with type transfer instructions in case of mixed arithmetic. As a consequence some restrictions, e.g. for actual parameters in procedure statements, had to be made. This we accepted since we considered the running time of the generated program to be far more important.

With respect to relations, and much more so with the Boolean operations, the situation is similar: in part at least they have to be represented by (open or closed) subroutines.

Next to decomposition the major task in ALGOL translation is the storage allocation as induced by the block structure. The block structure itself is a prime example of a bracket structure to be handled by the cellar method, both on the program and on the data storage side.

Conceptually this is very simple and elegant as long as the *own* is excluded: all the variables and arrays declared in each block are put on top of what is already present after the begin, and are taken out again after the corresponding end, and the numbers cellar for intermediate results is always at the momentary end of the sequence.

In practice, however, we have again the problem of static treatment (during translation) versus dynamic treatment (at run time). In a largely interpretive system, fully dynamic treatment as described by Dijkstra (5) is possible and very sensible.

The situation is different for a generating system. For dynamic treatment essentially means that data storage for each block is a relatively addressed sequence, with the point of reference dif-

ferent for each block, and determined by the running program. This can be handled effectively, if a sufficient number of index registers is available so that to each block a register of its own can be assigned. Otherwise, index registers would have to be simulated or shared, which is costly both in computing time and instruction storage, or the block data would have to be readdressed at each entry which also is rather tedious.

Therefore in our ALCOR translators two different data stacks were introduced: a static stack, comprising the simple variables and certain subsidiary quantities, which could be completely assigned and addressed at the end of translation, and a dynamic stack for arrays, (where some address calculation usually has to be done anyway) with a "free storage indicator" or stack pointer handed over from block to block. It should be mentioned here that this dynamic stacking of arrays was standard operating procedure in the subroutine organization of the ERMETH and on PERM since the mid fifties.

Handling of blocks immediately leads us to the delicate problem of how to handle procedures. Conceptually, dynamic stacking again is the immediate answer to the problem of data storage, since it means that at each procedure call, the momentary free storage is available to the procedure, and no data storage need be reserved. However, due to the admission of non-local quantities in procedures, we have the same problem of dynamical relative addressing as with blocks in general. Therefore it was decided, to assign static data storage to each procedure separately within the block containing the procedure, which of course rules out recursive procedures. The waste of static storage, in conflict with our original cellar principle, was considered regrettable.

There was no great concern, however, over recursive procedures. These recursive functions were originally introduced by McCarthy in order to formulate elegantly the theory of computability. They were considered by us to be of very little value in a language intended to describe actual rules of computation since they, in effect, do not describe recursive computation of the function value, but a recursive build-up of the explicit rule of computation for the function value.

In conformity with the general emphasis on efficiency of generated programs, methods to evaluate subscripts of subscripted variables recursively and using index registers were considered rather early (11) since these were deemed to be the most important problem of optimization.

Schemes of this type were incorporated in the translators for PERM by Seegmüller, and for ERMETH by Schwarz.

The extent of the language adopted by the ALCOR group after lengthy deliberations in 1960 was described in the ALGOL Manual of the ALCOR group by Baumann (3).

The main restrictions against ALGOL 60 are:

(1) no conditional Boolean expressions since these essentially duplicate the propositional calculus which is the better known of the two

(2) no "conditional designational expressions" since they are unnecessary

(3) no while-element in the for clause since this essentially duplicates the step until element

(4) no *own* since neither need nor definitions are clear

(5) no recursive procedures.

In addition there are rules for the use of procedures which are essentially programming rules.

(1) type procedures are called only by function designators. They serve only to compute the function value, and no side effects of any kind (recomputation of actual or global parameters, side exits) are permitted.

(2) In proper procedures called by procedure statement the function character is underlined by a standard heading of the form

P(x,y) results: (u,v) exits: (l,m)

separating different classes of parameters. This separation has to be used also in procedure statements.

These rules are intended to keep the notation close to standard mathematical form, and to ensure easy reading of programs. The ALCOR group considered both these requirements to be fundamental postulates with respect to the language.

To conclude, we come back to the ALCOR group to give a short survey of its present status.

The following translators were designed on the basis of ALCOR plans and are now running:
ALCOR-ERMETH, -MAILUFTERL, -ORACLE, -PERM, -S 2002, -Z 22, -Z 22 R.

They are all of the "load-and-go" type requiring no intermediate print-out although none of them except the ORACLE-translator has magnetic tapes at its disposal.

A translator for the Telefunken TR 4 has been programmed in Munich and will become operative with the machine. Smaller translators exist for DERA in Darmstadt and for the Zuse Z 22 (ALCORETTE).

The membership of the group at present is as follows:
Institut für Angewandte Mathematik der Eidgenössischen Technischen Hochschule, Zürich
Rechenzentrum der Technischen Hochschule München,
Institut für Angewandte Mathematik der Universität Mainz,
Institut für Praktische Mathematik der Technischen Hochschule Darmstadt,
Zentral-Laboratorium der Siemens & Halske AG., München,
Institut für Angewandte Mathematik der Universität Bonn,
IBM—Forschungsgruppe Wien,
Oak Ridge National Laboratory, Oak Ridge, Tenn.,
Telefunken GmbH., Backnang,
Zuse KG., Bad Hersfeld,
Dr. Neher Laboratory of the Netherlands Postal- and Telecommunications Services, Leidschendam,
Standard Elektrik Lorenz AG., Informationswerk, Stuttgart
IBM—Deutschland, Sindelfingen

Here, the last three should properly be called associated members since they developed, or are in the process of developing, their translators completely on their own. Their membership therefore concerns mainly use of the common hardware representation, and adoption of ALCOR-conventions concerning the use of ALGOL.

The ALCOR group is essentially an experiment which is not finished yet. Its success so far has been very encouraging although not perfect. In particular, the problem of active cooperation between members has not been solved to full satisfaction. This is due mainly to the severe shortage in manpower and funds in the University institutes which all had a full computing and teaching load to cope with. Thus every institute had to go ahead on its own to build its translator as the manpower situation permitted. The result was that the translators (all essentially one man jobs) show slight deviations. Thus there is still room for improvement, as was to be expected in a group consisting of members as individualistic as university institutes usually are.

In spite of these minor faults, however, we feel that we, in the ALCOR group, have already proved our point: it is possible to set up a uniform system which allows free exchange of programs and information between machines of types as widely different as e.g. Zuse Z 22, PERM, ERMETH and Siemens S2002 without seriously impairing the efficiency of the individual machine.