STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

ALGOL Bulletin
Supplement nr. 10

ALGOL-60 Translation

by

Dr. E.W. Dijkstra

November 1961

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

CWI BIBLIOTHEEK
3 0054 00077 3227

An ALGOL 60 Translator for the X1 *)

Introduction

An ALGOL 60 Translator for the X1 has been working in the Computation Department of the Mathematical Centre in Amsterdam since June 1960. This is naturally not the result of our first attempt. While the problem was yet new to us we began a few times by treating relatively simple tasks, but every solution we then found turned out later to be inadequate in more complicated cases. When the few times were past us we attacked the whole problem from the other side and subsequently subjected our new approach to, and tested it against, the most difficult situations imaginable. The basic form of this approach has not changed since, although its working out gave rise to various improvements. For example, the method of reference to anonymous intermediate results, which we had taken over directly from the old projects, now turned out on closer consideration to require too much storage space for the object program. It was clear how we had to improve this, and the modification could be carried out within two weeks. We got the idea of this modification during a discussion which our group had in Copenhagen with Messrs. J.Jensen, P. Mondrup and P. Naur of "Regnecentralen", Copenhagen.

We naturally chose this way of approaching the problem in the hope that, once a really satisfactory solution had been found for fundamentally complicated tasks, the working out in detail would no longer prepare unpleasant surprises for us, and in each case an elegant solution would present itself as if automatically. This hope has been fulfilled beyond all expectations: thus it turned out to our great joy and surprise that the translator would deal with certain existing extensions of ALGOL 60 without more ado. And it is this experience which prompts us to publish something about our project; it is also relevant that our solution is not only valid for the X1: it can be carried through with any good computer.

---

*) Published in German in two parts
MTW, 2, 1961, pp 54-56 and MTW, 3, 1961, pp 125-119

Presuppositions and Intentions

Our solution is for "good computing machines", where by "good" we want to mean that we are completely free to determine how the computer should be used, this in contrast with machines for which considerations of efficiency force us in practice to a special manner of use, that is, force us to take account of specific properties and peculiarities of the machine.

We shall therefore suppose that a sufficiently large homogeneous store is available. Thus we shall not be concerned with the problems that arise as soon as we want to make efficient use of a machine with a fast store of restricted extent together with a large but slower store. Indeed we undertake to subdivide the store, for whose cells we presuppose constant access time, as advantageously as possible.

We further suppose that the arithmetic unit is so fast that we may permit ourselves to use well-chosen subroutines to carry out the required arithmetic, logical and organisational operations from which the object program is constructed. Thus all the special possibilities of the order code hardly find any expression in the structure of the object program: they should nevertheless be fully utilised in the complex of subroutines which is "played" by the object program.

We are fully aware that we only attain a profitable use of the store and an extraordinary flexibility of the object program at the expense of a certain prolongation of the calculating time, and we can imagine that for some computer which is still in use today one cannot accept this delay. There are twofold reasons why we nevertheless made this choice, one of principle and one practical.

The reason of principle is that as a scientific Institute we would rather devote our time to the development of a programming technique which we expect to be realised in the near future than a technique for which this is not so.
On the one hand we have good reason to suppose that the percentage of machines for which our technique is suitable will grow, and one can more easily permit oneself to pay the price of a certain delay. On the other hand we anticipate that increased flexibility and the release from all kinds of restrictions of secondary importance which have normally to be observed while programming will come more

and more to be appreciated as valuable. This last could well be of decisive significance for the question of whether in the coming years one will be in a position actually to make full use of the rapidly increasing calculation capacity.

The practical reason is that the machine of the Computation Department of the Mathematical Centre, i.e. an X1 (provided by N.V. Electrologica of Amsterdam), fully satisfies the requirements which were made by our approach. Since the order code of the X1 only includes fixed point operations the floating point operations have in each case to be carried out by subroutines. This anyway makes the relative delay, compared with other solutions, considerably smaller. Besides this, thanks to the rapid subroutine mechanism, we can introduce quite short subroutines into the complex, a fact of which good use can be made.

Considerations of this kind have led us to try for a complete ALGOL 60 translator. We have become so consistent in this respect that we occasionally seemed to require of our translator that it should be able to cope even with situations for which it is questionable whether they were foreseen during the compilation of the "Report on the algorithmic language ALGOL 60": in the ALGOL 60 to be processed here we became accustomed "to allowing in principle everything which is not explicitly forbidden" (naturally on the condition that it has a clear unambiguous meaning). Thus in the first place it is our conviction that as soon as ALGOL 60 becomes accepted refined programmers will exploit the possibilities of ALGOL 60 in just the same way as they do now with machine codes. Besides this we are afraid that mutual interchangeability of ALGOL programs will soon become an illusion if everyone who makes a translator allows himself the option of leaving out everything from the language that does not suit him personally. We have however at one point not remained true to this principle. The declaration own cannot be applied unrestrictedly: it cannot be used during recursive use of a procedure and for array declarations the use of own dynamic upper and lower bounds for the indices is excluded (see [1], 5.2.2., example 2).

Arithmetic

With regard to arithmetic the structure of the object program is rather conventional: algebraic expressions are evaluated with the help of an accumulator stack (see e.g. [2]). In the following we use the symbol: {X} for stating the contents of the storage cell with address X, and the inverse symbol: }x{ for stating the address in which the quantity x is found. We denote the accumulators in the stack by v_0, v_1, v_2, ... etc.

Thus the evaluation of the assignment statement x :=a-b×(c+d)+e takes place in the elementary steps:

v_0 := }x{ ;
v_1 := a;
v_2 := b;
v_3 := c;
v_4 := d;
v_3 := v_3+v_4;
v_2 := $v_2 \times v_3$;
v_1 := v_1-v_2;
v_2 := e;
v_1 := v_1+v_2;
{v_0} := v_1;

Each accumulator v_i occupies a number (in our case 4) of consecutive storage cells. An accumulator contains either a number or an address, together with a declaration as to which one of the two. Secondly each accumulator contains a real-integer indication, which in the first case refers to the number in the accumulator itself, and in the second case to the content of the storage cell whose address is contained in the accumulator. (An explicitly named variable of type Boolean or integer occupies one storage cell, but one of type real occupies two.) All arithmetic operations start by investigating the real-integer indications of the two operands: if they differ, the real representation of the number in integer representation is first formed. The inverse transformation (including rounding) is introduced when a value

which is formed from arithmetic in real representation is assigned to a variable of type integer, e.g. in:
"; integer n; n:=a/b;" .

The system of the accumulator stack is clear: if a new number is called up from the store it is transferred to the first free accumulator and the number k of filled accumulators is increased by one. An arithmetic operation is always carried out on the numbers in the two last filled accumulators, and the number k thereby reduced by one. If the changes of k were done by the translator the current value of k would have to be represented in the object program in the specification of every action; at the cost of a negligible loss of time we could store the object program much more compactly if we left the appropriate changes of k to the arithmetic subroutines. The object program then takes the following form (assuming suitable declarations):

Numbers of orders			change of k
2	TRAx	TAKE REAL ADDRESS	+1
2	TRRa	TAKE REAL RESULT	+1
2	TIRb	TAKE INTEGER RESULT	+1
2	TRRc	TAKE REAL RESULT	+1
2	TFRd	TAKE FORMAL RESULT	+1
1	ADD	ADD	-1
1	MUL	MULTIPLY	-1
1	SUB	SUBTRACT	-1
2	TRRe	TAKE REAL RESULT	+1
1	ADD	ADD	-1
1	ST	STORE	-2
17	Total		0

In our organisation the "addressless" operations ADD, MUL, SUB, ST (and the others, DIV etc.) require one order (a subroutine jump) in the object program, the "addressed" operations like TRA and TRR require two orders. In consideration of this we have also introduced the addressed versions of the commonest operations (+, -, x and /) to shorten the object program, and the form of the object program now runs as follows:

Numbers of orders			change of k
2	TRA x	TAKE REAL ADDRESS	+1
2	TRR a	TAKE REAL RESULT	+1
2	TIR b	TAKE INTEGER RESULT	+1
2	TRR c	TAKE REAL RESULT	+1
2	ADF d	ADD FORMAL	+0
1	MUL	MULTIPLY	-1
1	SUB	SUBTRACT	-1
2	ADR e	ADD REAL	+0
1	ST	STORE	-2
15	Total		0

One might imagine that the number of accumulators used and the number of orders could be reduced by referring to the variables in a different order. Thus in order to carry out the component operation $v_2 := b \times (c+d)$ one might suggest:

2	TRR c	TAKE REAL RESULT	+1
2	ADF d	ADD FORMAL	+0
2	MUI b	MULTIPLY INTEGER	+0
6	Total		+1

In this way 6 orders suffice, whereas our solution requires 7. The translator does not carry out this abbreviation since it is not in general allowed. During the evaluation of the formal variable d a procedure might be performed which as a subsidiary function altered some non-local variables, including possibly value of b. In the above abbreviation the new value of b would then be used, although ALGOL 60 requires that the old value of b should be used in the evaluation of x, because expressions have to be worked out from left to right.

For the same reason the operation STORE is performed without an address. In this connection one should note that the ALGOL program:

```
";i := 5;  x [i]  := i := 7;"
```

leaves the array element x [7] unaltered, but performs the assignment x [5] := 7.

Arrays.

The reference to the cells in which the parameters of the storage mapping function are stored functions as the "address of an array"; the operation TRA in the next example is, as its name already indicates, carried out by the same subroutine that also puts the address of an unsuffixed variable of type real into the next accumulator. If x is the designation of an array of type real with two indices, then a reference to the array element x [i, j+k] in the object program (assuming suitable declarations) is coded as follows.

Number of orders			change of k
2	TRA x	TAKE REAL ADDRESS	+1
2	TFR i	TAKE FORMAL RESULT	+1
2	TIR j	TAKE INTEGER RESULT	+1
2	ADR k	ADD REAL	+0
1	IND	INDEXER in this case:	-2
9	Total		+1

The operation IND (INDEXER) is addressless and takes a subroutine jump in the object program. In this process the successive accumulators, beginning with the last filled, are tested to see if they contain a number. If so then depending on the real-integer indication the transfer to integer may be introduced: all index values are essentially integral. In passing, the number of index values we find is counted: this process terminates as soon as an accumulator is found which contains an address (in our case the address } x {). With the help of this address the required storage mapping function can now be found and in this accumulator the address of the required array element is left by IND; the index k is reduced in such a way that this accumulator now becomes the last filled. The net result of the above program is thus:

"v_k := } x [i, j+k] {; k := k+1;"

If we are only interested in the address of the array element - e.g. when it appears on the left of a becomes-sign ":=" in the ALGOL text - we let it stay here. If on the other hand we are interested in the value of the array element there follows also the addressless operation TAR (TAKE RESULT), which is (and may

only be) called if the last filled accumulator v_{k-1} contains an address.
The effect of TAR is thus given by:

$$"v_{k-1} := \{ v_{k-1} \}"$$

It is clear that in this way indices are allowed to be arbitrarily complicated expressions.

Simultaneous Assignments

The operation ST (STORE) was described above; it is given by:

$$" \{ v_{k-2} \} := v_{k-1}; \quad k := k-2;"$$

The index k is here decreased by two because the contents of two accumulators are finally processed. In the so-called simultaneous assignment we wish to assign a result once formed to several variables. To this end the operation STA (STORE ALSO) is introduced, given by:

$$" \{ v_{k-2} \} := v_{k-1}; \; v_{k-2} := v_{k-1}; \quad k := k-1;"$$

The operation STA begins to work with the two last filled accumulators like ST; then the result is pushed back by one accumulator and k is only reduced by one, so that the result still remains in the last filled accumulator. Thus e.g.

"x := y [k [j]] := h := q;" gives:

Number of orders			
2	TIA x	TAKE INTEGER ADDRESS	+1
2	TIA y	TAKE INTEGER ADDRESS	+1
2	TRA k	TAKE REAL ADDRESS	+1
2	TFR j	TAKE FORMAL RESULT	+1
1	IND	INDEXER in this case:	-1
1	TAR	TAKE RESULT	+0
1	IND	INDEXER in this case:	-1
2	TIA h	TAKE INTEGER ADDRESS	+1
2	TRR q	TAKE REAL RESULT	+1
1	STA	STORE ALSO	-1
1	STA	STORE ALSO	-1
1	ST	STORE	-2
17	Total		0

Procedures and Blocks

Every procedure has the properties of a block; conversely every block which according to the ALGOL text is not a procedure can be considered as a (parameterless) procedure which is only called at one place. Since our translator does this we shall use the words "procedure" and "block" interchangeably without distinction in what follows.

We have described above how the arithmetic is determined in the object program. It is of interest to note here that not only are the individual accumulators not mentioned explicitly but it is also nowhere explicitly specified where the accumulator stack is to be found in the store.
For this there is good reason: it means that the object program has elsewhere the responsibility to decide in the first place where the stack is to be localised. It has in fact not only the responsibility but also the freedom to alter this decision during the running of the program. It will exploit this freedom so as to use the store as advantageously as possible and so as to make it possible for procedures to call each other or themselves a number of times. Note that this number can only be determined dynamically and is thus essentially unknown during translation. This decision mechanism comes into action each time a procedure is called, and it is this mechanism that we shall now describe.

The arithmetic complex is controlled by four administrative "state quantities" which are held in four stores reserved for this purpose. These are:

AP = ACCUMULATOR POINTER
WP = WORKING SPACE POINTER
PP = PARAMETER POINTER
BN = BLOCK NUMBER

The quantity AP plays the role of the above-named index k of the v_k; that is, AP is the starting address of the first free accumulator. Since each accumulator occupies 4 storage places, the increase "k := k+1" given above (during the filling of the next accumulator) corresponds in the subroutine complex to the

operation "AP := AP+4".

As the examples show, all acumulators which are filled during the execution of a statement are again freed during the course of this execution, in other words, after the lapse of the statement AP has again the same value as at the beginning. Thus in a series of statements of one and the same block the quantity AP assumes a constant value in between the statements: this constant value is moreover held in the quantity WP, which thus specifies the "beginning of the working store" during the execution of a block.

The processing of algebraic expressions is organised in such a way that when a complicated subexpression is to be calculated the value of this subexpression, however complicated it may be, is always finally written into that accumulator which initially was the first free one. In the meantime a number of the next accumulators are used temporarily for the evaluation of the subexpression.

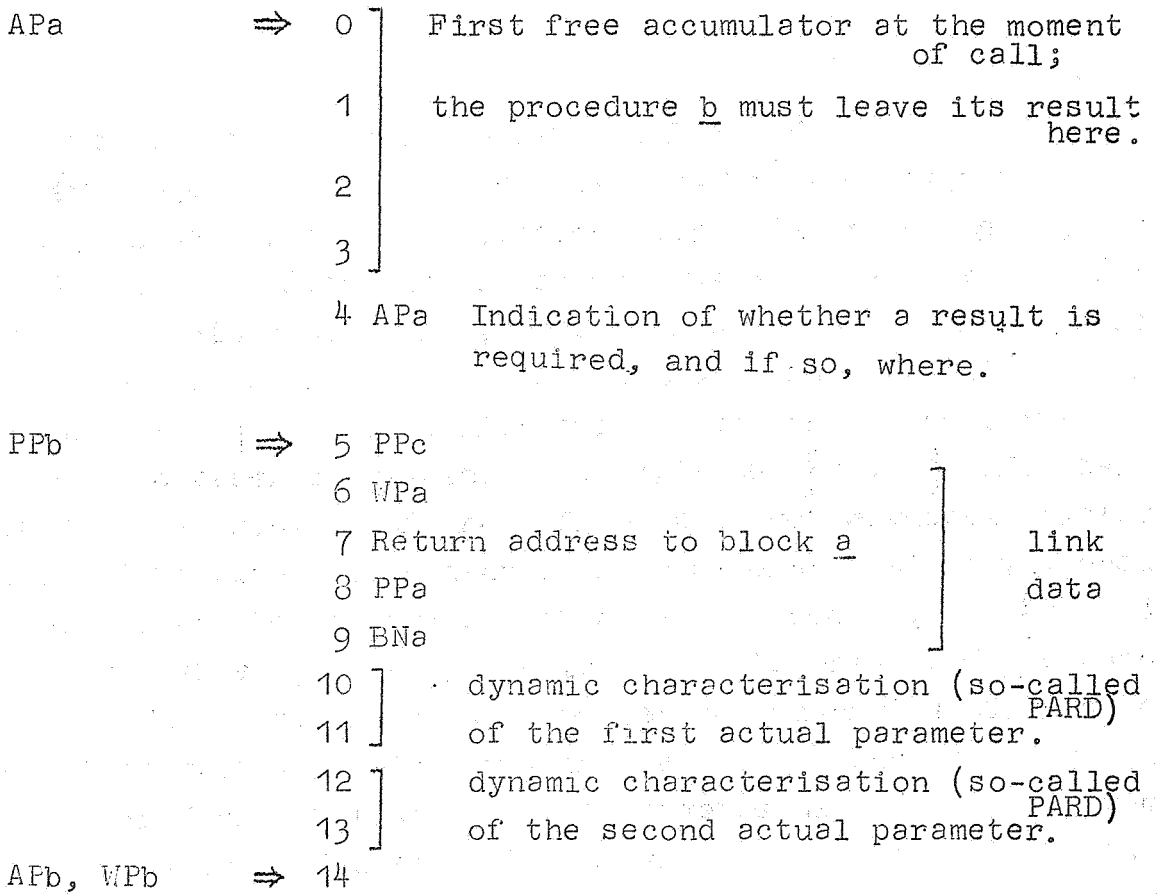
The idea occured of applying the same technique to cases where a part of an expression is given by a (function) procedure. The latter is basically likewise a "complicated subexpression": the single difference is that the calculation procedure for this subexpression is defined elsewhere (and with greater freedom), namely in the procedure declaration. In other words, each procedure should be constructed in such a way that it works in that part of the stack which begins with the cell indicated by the value of AP at the moment of the call. This value is held in the quantity PP: the PARAMETER POINTER thus continually specifies the place in the stack where the currently active block began to work. (The PP value which corresponds to the single activation of the main program is unimportant.) Like the quantity WP, PP is also constant during a (particular) execution of a block.

This holds also to a greater degree for the last administrative quantity BN: in fact for a given block BN always assumes the same value. For each block the corresponding value of BN is determined once and for all on purely lexicographical grounds during the translation: BN specifies namely for each block by how many blocks it is (lexicographically) enclosed.

Consequently BN is zero for the main program. At the (single) entry to each block the translator inserts instructions into the

object program which give to BN the value corresponding to the block. The quantity BN plays a part in the process of reference to non-local variables (see below).

Suppose that block a contains an expression in which a (function) procedure is called. Let the procedure body be block b, and let this be lexicographically directly enclosed by block c. In other words, the activation (function designator) of block b lies within block a and the definition (procedure declaration) of block b lies in block c. (Note that a = b or a = c is allowed; b = c is of course excluded). We denote the values of the state quantities current at the moment that block b is called in block a by APa, WPa, PPa and BNa. We now give a picture of the stack when the passage from block a to block b has just been completed, and on the assumption that block b is a procedure with two formal parameters.



Above is given the stack picture which is generated by the call mechanism ETMR (EXTRANSMARK RESULT). This first increases AP by 4 to reserve an accumulator for the result of the procedure and the beginning address of this reserved accumulator is

stored in the next cell.

Now ALGOL 60 also allows the call of this procedure to occur outside of an expression, that is as an independent procedure statement. In this case the calling program has no interest in the value which the procedure will assign to its own identifier; the increase of AP by 4 is suppressed and to indicate this situation the next cell is filled with a negative number (= -0). This takes place in the procedure activation mechanism ETMP (EXTRANSMARK PROCEDURE) which evokes the following occupation of the next cells in the stack:

APa	⇒	0	-0	Indication that a possible result is not required
PPb	⇒	1	PPc	
		2	WPa	
		3	Return address to block a	
		4	etc.	
		5	.....	
		6	.....	

(ETMP and ETMR are two different entries to the same activation program ETM: after a few orders they run together.) For assigning the value to the procedure identifier itself (see [1], 5.4.4) the object program has at its disposal the special subroutine STP (STORE PROCEDURE VALUE) - or the analogous subroutine STAP (STORE ALSO PROCEDURE VALUE). In this case the mechanism STP examines the content of the cell PPb-1; if $\{PPb-1\}$ is positive, then $\{PPb-1\}$ specifies the beginning address of the accumulator reserved for this result and the content of the most recently filled accumulator is transferred there; if however $\{PPb-1\} = -0$, then this transfer is suppressed, since the procedure b has evidently been called by ETMP. (The fact that ALGOL 60 permits a function procedure to be activated by ETMP we have found to be particularly useful for Boolean procedures.) The next cell, which is reserved for PPc, is left blank by ETM; this cell is filled by the mechanism SCC (see below) at the start of the procedure. The next four cells are filled with data which relate to block a, the block which we are temporarily leaving; these data are at this moment immediately available, they make it possible correctly to continue the calculation in block a on completion of the procedure b. Further, the static characterisa-

tions (so-called PORD's), as they appear in the program text of block a, are translated into dynamic characterisations (so-called PARD's). (If an actual parameter is a simple variable, the PARD contains the physical address of the variable. If an actual parameter is an expression, this is expressed in the form of a subroutine and the corresponding PARD will contain the beginning address of this subroutine. An actual parameter can primarily be given as an address (if it is an "output parameter", i.e. as a possibly suffixed variable) or as a numerical or logical value (expression); these and other data which may change from call to call are expressed by ETM in the PARD's). Each PARD occupies two cells in the stack and ETM increases AP and WP up to the first free place.

The control of the X1 jumps after completion of ETM to the beginning address of block b, since the next actions are dependent on the particular procedure which is now activated. If local quantities are declared in this block, the next cells in the stack are reserved for them. As soon as the procedure is called it is known how much store these local quantities require on this occasion, and the procedure starts by increasing APb and WPb by this amount, before the arithmetic proper begins. But this means that the local quantities of block b in the text of the object program can only be (and are) localised with respect to the current value of PPb.

Before the increase of AP and WP described above the action SCC (Short Circuit) is first performed, under control of the value BNb of the block number, which the translator has given to the block b. The action SCC makes the state quantity BN equal to the given number (so in our case BN := BNb), and records in the still empty cell specified by the current value of PP (in our case PPb) the PP value belonging to the first block which lexicographically encloses block b (so in our case the value PPc is put in the stack). The action SCC is able to find the value PPc since the block number of block c is known to it: in fact BNc = BNb-1.
The required value PPc can be found with the help of BNc in the so-called DISPLAY (see below); the action SCC is necessary in order later to be able to guarantee that the Display can be adjusted at every block transfer.

The Display.

During the translation all local variables which are to be stored anywhere at all in the stack are localised with respect to the PP value of the block in which they are declared. The value of this PP cannot be known during translation, since it is determined anew at each activation of the block during the calculation. On the other hand the block number certainly is known during translation. Hence each local variable is characterised during translation and in the text of the object program by its position p with respect to PP and the block number n, both belonging to the block in which the variable is declared. During the execution of a block the arithmetic complex of subroutines, in order to be able to find the local and non-local variables in the stack, must have access to the PP values of the most recent as yet incomplete activation of the block itself, and of the blocks which enclose this block lexicographically, respectively. Now these PP values appear in order of block number in the so-called Display; this is a series of storage cells which play the part of index registers. To determine the physical address of a variable it is necessary to refer to the Display: if the variable is characterised by position p and block number n (see above), then the required address is found by adding to p the content of position n of the Display. For each block the non-local quantities are declared in lexicographically enclosing blocks, whose block numbers are therefore lower. It is thus in general necessary, for the correct execution of a block, that the Display is correctly filled up to and including the cell specified by the current value of the block number. Every time that the correct filling of the Display becomes uncertain the Display is adjusted up to the current block number by the action UDD (UPDATE DISPLAY).

We can now understand that the action SCC is always possible for non-formal procedure statements: when a procedure is called, all its non-local variables are declared in blocks which enclose not only the procedure declaration but also the procedure statement. In other words, when in the above case block b is activated the action SCC can find the required value PPc in the Display, in fact at the location BNb-1; secondly the newly introduced value PPb is written into the Display at the location BNb.

The action SCC is necessary in order later to be able to carry out the action UDD, which among others forms a part of the RETURN mechanism at the end of a procedure. The control then returns to a block and obtains from the link data the information about PP and BN of the block to which it returns; BN then specifies where in the Display this PP value is to be introduced. With this PP value as initial value of x, by repeated execution of

$$x := \{ x \}$$

x takes on the PP values which have to be introduced into the Display in decreasing order of block number.

Now the variables which occur in the calculation may be localised in two ways: statically or dynamically.

All quantities which are declared in the main program are localised statically; likewise "own" variables are localised statically. Static localisation means that the translator determines the physical addresses where these variables are stored, and that in consequence each reference to such a variable in the text of the object program contains the associated address. (The static localisation of own variables is the origin of the restrictions of our translator mentioned earlier. It is by the way not sufficiently clearly described in the ALGOL Report [1] what the consequences of the concept own should be in the case of recursive use.)

Dynamic localisation is the localisation described above of variables in the stack with respect to the PP value associated with the block.

This has the consequence that the operations that need an "address" of a variable arise in fivefold manner in the arithmetic complex. Thus for example for the operation TAKE RESULT which puts a new number in the next accumulator we have the following five versions:

TRRD	TAKE REAL RESULT DYNAMIC
TRRS	TAKE REAL RESULT STATIC
TIRD	TAKE INTEGER RESULT DYNAMIC
TIRS	TAKE INTEGER RESULT STATIC
TFR	TAKE FORMAL RESULT

As shown, the operation TFR only occurs once; the distinc-

tion between static and dynamic is dropped since the PARD of a formal parameter always lies in the stack and is thus dynamically localised. Neither do we distinguish here between real and integer: this last is determined by the object program at the moment that the actual parameter is transmitted (see below).

Actual and Formal Parameters

The establishment of an actual parameter in the text of the object program is done where possible in one word. This word (called a PORD) consists of three parts, a (15 bits), t (2 bits) and Q (2 bits).

The two bits of t specify whether the 15 bits of a must be interpreted as a static or dynamic address, and in the latter case further whether the actual parameter as it stands in the procedure statement is already formal (the "handing on" of a formal parameter). For a non-formal actual parameter Q has the following meaning:

Q = 0 : a is the address of a variable of type real
Q = 1 : a is the address of a variable of type integer
Q = 2 : a is the beginning address of a procedure (a subroutine) with or without a numerical result.
Q = 3 : a is the beginning address of an (implicit) subroutine with an address as result.

As soon as an actual parameter is too complicated to be fully characterised as above by one word, this actual parameter gives rise to a so-called implicit subroutine in the object program: the PORD then contains the beginning address of this implicit subroutine, together with the specification of whether the result is an adrress. If the actual parameter is a suffixed variable, then the corresponding formal parameter within the procedure may perhaps stand on the left side of a becomes-sign; and for this reason this implicit subroutine which this actual parameter defines yields the address and not the value of the variable. This situation is indicated in the PORD by Q = 3; to any other implicit subroutine corresponds a PORD with Q = 2.

The PORDS are translated by the call mechanism (TRANSMARK) into so-called PARDS - thereby among other things dynamic addresses are converted into physical ones. The PARDS, which occupy two

words each, are stored in the stack following the link data (thus at the locations PP+5, PP+7 etc.). For procedures the PARDS play the part of formal parameters; if for example a procedure wants to use the second parameter there will appear in the text of the object program a reference to the PARD with dynamic address PP+7. The first word of a PARD is derived from the PORD, the second contains the PP value and the block number belonging to the block in which the corresponding procedure statement is given; only when the PORD specifies that a parameter which is already formal is transmitted are both words of the corresponding PARD transported by TRANSMARK. The second PARD word has no meaning in the cases Q = 0 or 1; if Q = 2 or 3, then PP and BN from the second PARD word are used as initial values for the process UDD on every occasion that the procedure requires the execution of the subroutine. In order to be able to guarantee the evaluation of such a complicated parameter or a formal procedure, the Display must contain the same PP value for the lower block numbers as at the moment of call; this adjustment of the Display is carried out by UDD under control of the second PARD word.

Analysis of a PARD takes place in each of the following operations:

TFA	TAKE FORMAL ADDRESS
TFR	TAKE FORMAL RESULT
ADF	ADD FORMAL RESULT
SUF	SUBTRACT FORMAL RESULT
MUF	MULTIPLY FORMAL RESULT
DIF	DIVIDE FORMAL RESULT

If this analysis finds that Q = 0 or 1 it is soon finished. If however the analysis finds Q = 2 or 3, then to obtain the required result an expression or a procedure must be evaluated, in principle of unrestricted generality. But this means that for the six above-named operations recursive activation must be possible. To the above six are added two further mechanisms for the activation of a formal procedure, namely FTMP (FORM TRANSMARK PROCEDURE) if no result is required, and FTMR (FORM TRANSMARK RESULT) if one is required. The possible recursiveness of these mechanisms requires three more places in the stack than

ETMP and ETMR respectively.

Concluding Remarks

In the foregoing a survey is given of the structure of the object program, or rather an overall impression of the operations with the help of which the translator has to formulate the object program. The subroutine complex which executes these operations does not differ essentially with regard to storage requirement and speed from a normal complex for floating point operations. The structure of this complex is also in large part conventional: only the problem of activating new blocks or terminating their activity has given rise to program constituents which are, at least by our present standards, rather complicated.

The particulars given above are described in the first place for those readers who have concerned themselves more or less intensively with the construction of a translator. But it will also be clear to the reader with more general interest that the making of an ALGOL translator is a relatively simple job if the translator may formulate the object program in operations cut out for the problem. It was possible thereby for the translator, which contains about 2500 orders, to be written in a few months by two people - namely by J.A. Zonneveld and the author. I would like a few special properties of the translator not to go unmentioned.

In the first place the translator is to a large degree independent of the method chosen for the representation of the ALGOL text (hardware representation). Each time the translator requires the next ALGOL symbol it calls a subroutine which has to provide the next ALGOL symbol in a fixed internal representation. There have to be as many different versions of this subroutine as there are representations to be processed. Punched tapes with 7 or 5 channels are available.

The code in which the object program is punched exhibits the same form of flexibility. All references to the subroutine complex which are to be at the disposal of the object program are numbered, and for these cases the translator only punches the number. The punched tape with the object program has to be read in by a special simple read-program, which is provided beforehand with the data which it has to substitute in place of the numbers.

At present various subroutine complexes have been developed, working to various degrees of precision. It is intended to develop further complexes which although working rather more slowly will offer the possibility of automatic program testing. All these complexes may be manipulated by the same object program tape: one has only to specify the required complex when the object program is read in.

Perhaps the most important point of our method of translation is that we do not discriminate on the combination of two consecutive delimiters (as described e.g. in [2]) but on each individual delimiter. The fact that we do not have so-called transition matrix but a discrimination vector has probably contributed not inconsiderably to the reduction of the size of the translator.

The name list is organised as a stack. At the beginning of a block its local names are added to the name list, and as soon as the translation of the block is complete these names are again struck off the name list (by suitable lowering of a pointer). Thanks to the complete bracket of the ALGOL language it is moreover not necessary to introduce more than one stack. Algebraic expressions, bracketed conditional expressions and/or statements, bracketed for -statements and procedure declarations can all be translated with the above universal stack. The processing of the for-statement, which is admitted without any restriction, is made much easier by considering the domain of the for-statement as a block (it is forbidden e.g. for a goto-statement to lead into it from outside.)

The various lists which the translator builds up during its work are laid one after another in the working stores: if one of the earlier lists grows too much the following lists are shifted up. The translation only stops through lack of storage when the entire extent of the working store would not suffice. Thus the translator also, as well as the object program, uses the store as appropriately as possible.

So as to let the translation proceed as fast as possible the interrupt facility is fully used and input and output take place in parallel with the translation: data transfer from the input to the translator and from the translator to the output occurs through cyclically arranged buffers which absorb the

variations in the speeds of processing and production.

I owe the greatest thanks firstly to my colleague Mr. J.A. Zonneveld, who made the intensive cooperation from first to last a fruitful pleasure, and secondly to Professor van Wijngaarden, who made many constructive contributions in the first months, when we three had to determine the tactics to be followed. A further word of great gratitude is due to almost all the staff of the Computation Department of the "Mathematisch Centrum" for their extensive and accurate work.

Translation by M.Woodger,

October 1961.

References

1 "Report on the Algorithmic language ALGOL 60". J.W. Backus et. al.

2 "Sequentielle Formelübersetzung", K. Samelson, F.L. Bauer. Elektronische Rechenanlagen 1 (1959), Vol 4.

Making a Translator for ALGOL 60. *)

I do not feel myself entitled to give complete prescriptions how to make a translator for ALGOL 60, for the problem of translator construction has two aspects. On the one hand we are faced with ALGOL 60, on the other hand we are usually confronted with a particular machine that has to perform the computations described by the ALGOL program. (And as a rule this same machine has to perform the translation.) As certain machine properties may present specific difficulties in bridging the gap between a process description in ALGOL 60 and its actual execution, I do not claim to be able to treat the subject in its entirety. In principle I shall restrict myself to my actual experience, i.e. making an ALGOL 60 translator for the X1, the computer of the Mathematical Centre at Amsterdam. As a result I shall not touch the problems that arise as soon as a machine with a two-level store has to be used efficiently. I shall point out alternative solutions as they present themselves and include the improvements we discovered after our translator had been finished.

Before one can start making a translator which is fed with an ALGOL program and has to produce the so-called "object program", one has to decide what the structure of the object program will be, because only then the task of the translator becomes well defined. What I call the "object program", has also been described as "an equivalent program in machine language", but I prefer not to use the last description, not being convinced that machine language will be the most appropriate language. I therefore ask you to consider the object program as an equivalent description of the process, more adapted to the requirements of the machine which has to do the actual computation, than the source description in ALGOL 60.

The object program is built up from a (limited) number of well chosen operations, each explicitly supplied with the appropriate number of parameters (may be equal to zero). Whether these operations will be written out in full in the object program or whether they will be denoted by a code number or a subroutine jump depends largely on the structure of the order code of the machine itself and the amount of storage space one is willing to provide for the storing of the object program. As the X1 is a fast fixed point computer, nearly all standard operations in the object program are denoted by subroutine jumps in our case.

---

*) Published in the APIC-Bulletin no 7, May 1961.

Furthermore we must be willing to face the desirability (and to act accordingly) of including in the object program certain operations, which do not correspond to something, explicitly prescribed in the ALGOL program. The ALGOL program, for instance, does not say a word about storage allocation for the variables. The declarations announce, which identifiers will be used for variables of all types, but it is left to the organisation that realizes this computation, to decide where the variables are to be stored. This is what is meant by storage allocation. Part of this job can be done during translation but for the sake of economy as regards the using of the working store, it may be desirable that the object program does some part of the allocation job at run time, adapting the actual allocation to the conditions every time they change.

But if the object program is not necessarily written in machine language and, furthermore, certain implicit tasks, such as storage allocation, may be postponed until execution time, one might well raise the question, whether the preceding paragraphs did not reduce the task of the translator to next to nothing. To remove this doubt we should direct our attention to those functions that certainly do belong to the task of the translator.

We have, for instance, the so-called "priority rules". In the statement

```
x:= a + b * c;
```

the execution of the multiplication must precede that of the addition. Another way of specifying this order of execution is

```
x:=(a +(b * c));
```

and we may regard the priority rules as a convenient mechanism for reducing the number of brackets needed. But the translator must evidently be aware of the priority rules and follow all their consequences. But for every ALGOL program this analysis needs only to be done once and is therefore regarded as one of the tasks of the translator.

The next point we raise is the analysis of the bracket structure. The function of a bracket pair may be regarded as "shielding an expression from its surroundings". By putting an arbitrarily complicated expression between brackets, it may play the role of a simple variable in a (larger) expression; on the other hand, an expression between brackets can be evaluated as such, independent

of the way the result will be used. If we start to scan the formula

x * (a +(1 + y)/(1 - y) - sig)

from left to right, we find an x, which is the first factor of a product. The second factor starts with an opening bracket. The multiplication must be postponed and, what is more, we can temporarily forget about the multiplication until we have found the corresponding closing bracket. It is obviously essential for the correct evaluation of such a formula to find the implicitly given one to one correspondence between opening bracket and "compensating" closing bracket. The determination which brackets form pairs, a job that implies some form of counting, can be done once and for all and is therefore a successful candidate for the translation stage.

There is another point: which opening bracket belongs to which closing bracket is defined by the lexicographical order, and before we can go on, we have to explain what we mean by the term "lexicographical".

An ALGOL program is a linear sequence of symbols, fed to the translator in a well defined order ("from left to right"). This order is called "the lexicographical order" in contrast to the dynamic order, i.e. the order in which operations are to be performed when the program is executed.

The lexicographical and dynamic order are closely related to one another because in principle expressions are evaluated from left to right and statements are performed in the order in which they are written. But the language includes a number of mechanisms for loosening the close connection between the two orders: priority rules, brackets, conditional expressions and statements, for statements, go to statements etc.

Let us now consider a piece of ALGOL program of the following structure: "if B1 then begin S1; if B2 then go to A else S7; S2 end
else begin S3; S4 end;
S5;
A: S6"

consisting primarily of a conditional statement followed by the two statements S5 and S6, the last one being labeled with the label A. Dynamically, the evaluation of the Boolean expression may have one of two successors, either the statements following the following then or the statement following the corresponding else. But finding the corresponding else implies, because the first alternative is a

compound statement, finding the symbol end corresponding to the symbol begin, that immediately follows "if B1 then".

This example shows that the dynamic successor to the evaluation of B1 is defined in a rather implicit way in the case in which the logical value of B1 turns out to be false. A useful task for the translator is to provide the object program with a more direct link (read: a jump order) to the second alternative.

To stress the fact that the definition of "corresponding" opening and closing parentheses is a purely lexicographical matter, we included a go to statement leading out of the first compound statement. If both B1 and B2 are true, we "enter" - dynamically speaking - the first compound statement through its begin but we never "pass" its end.

Closely related to the nested structure of ALGOL 60 - i.e. bracket pairs occurring inside bracket pairs - is the multiple use of the same symbol in different meanings. Let us confine our attention to those parts of the ALGOL program in which the statements occur, i.e. we disregard the declarations and the procedure headings for the sake of simplicity. We consider three different bracket pairs:

1) the square brackets enclosing the subscripts of a subscripted variable
2) the parentheses enclosing the actual parameters of a procedure statement or a function designator
3) the symbols for and do, enclosing the "for list elements".

If the translator encounters as part of a statement the symbol "," this comma must be enclosed lexicographically by at least one of the bracket pairs mentioned and its interpretation depends on the type of the innermost enclosing bracket pair.

It is quite probable that the system that will realize the computation as described by the ALGOL program will have to make a distinction between these different commas. If so, this distinction can be made during translation because it can be done once and for all and, furthermore, is defined by the lexicographical structure of the text to be translated. We shall show how the translator can make the distinction easily, provided we introduce a four-valued translator variable, specifying the "state of comma interpretation". Calling this state variable CI, we have say,

CI = 0, everywhere, where no comma may occur

CI = 1, comma separates subscripts

CI = 2, comma separates actual parameters
CI = 3, comma separates for list elements.

A last typical translator function is connected with the identifiers. As stated explicitly in the Report, identifiers have no inherent meaning, i.e. they could be replaced by something else. We had better make use of this freedom by substituting for the names used in the ALGOL program other names, but more suitably chosen.

If an identifier occurs somewhere in a statement, this identifier has a meaning, but only thanks to the fact that the same identifier has been declared to have this meaning. If one wants to find the relevant declaration, one has to scan the declarations at the beginning of the block in which the statement occurs. Either we find a declaration concerned with the identifier in question, or not. In the first case we have found the declaration we wanted, in the second case we scan the declarations at the beginning of the next lexicographically enclosing block, etc. (If we scan the declarations at the outermost block and still do not find a declaration for this identifier, then we may assume the identifier to be declared in the "universe" in which every program is embedded: the use of this identifier presumes some a priori knowledge. This is the case for the standard functions sin, cos, arctan, etc.)

It is clear from the above that finding the corresponding declaration may be a rather time consuming process, involving a lot of scanning. However this correspondence is unique: the translator could do a useful job by establishing this correspondence in a more direct way.

Before I can give a sketch of the translation, I must "choose" a structure for the object program, I must choose a machine and its order code. This we can always do, because if our specific machine does not have the required features built in, we can use it to simulate our chosen machine. We presume that our object machine performs its arithmetic in what is called a stack, a push down list or a nesting store. It allows us to write down the computation of

$$A + (B - C) * D + E$$

in the following form:

```
TAKE A          v0:= A
TAKE B          v1:= B
TAKE C          v2:= C
SUBTRACT        v1:= v1 - v2
TAKE D          v2:= D
MULTIPLY        v1:= v1 * v2
ADD             v0:= v0 + v1
TAKE E          v1:= E
ADD             v0:= v0 + v1
```

In this description we use two kinds of orders: the order TAKE (with the "address" of a variable) that fills a new v, and the arithmetic operations (without address), that always operate on the two "youngest" v's, leave the result in the oldest of the two and leave the youngest one free. All these operations work under control of an implicit administrative variable (a so-called "stack pointer"), which points to the next free v. The operation Take implies an increase of the stack pointer, the other operations imply a corresponding decrease of the pointer. This process description - the reverse Polish notation - gives rise to a straightforward scheme for storing and using the anonymous intermediate results that occur during the evaluation of an algebraic expression. It demands the presence of a stack (for the v's) at run time.

Once this form of object program has been chosen, the task of the translator becomes a little bit more defined, at least as far as the translation of expressions is concerned. We shall show how the translation of expressions written in ALGOL 60 into the reverse Polish notation can be performed by a translator and even by a translator which has to operate under rather severe restrictions as regards working space available during translation.

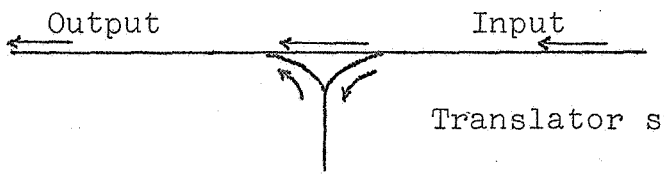
We aim at what I should like to call "immediate translation ", i.e. a translation process that reads the ALGOL program from begin to end, simultaneously producing - say, punching out - the corresponding object program. In other words, we do not assume the presence of a memory large enough to store the complete ALGOL 60 program nor the complete object program. In the first case we should be able to do all kinds of scanning of the ALGOL text, in the second case we should have the possibility of making corrections in a piece of object program produced a certain time ago. The translation process to be described is much less demanding as regards working space: in

fact it only stores information as long as it may be needed during translation. The storage requirements of this translation process are not strongly dependent on the length of the program to be translated; they are more a measure of its intricacy.

To describe the rules of precedence we assign priority numbers to the delimiters, e.g.

0 begin [(if for
1 end]) then else, ;
2 :=
3 ≡
4 ⊃
5 ∨
6 ∧
7 ¬
8 < ≤ = ≥ > ≠
9 + -
10 neg */ ("neg" represents the so-called unary"-"operation)
11 ↑

The translation process shows much resemblance to shunting at a three way railroad junction of the following form

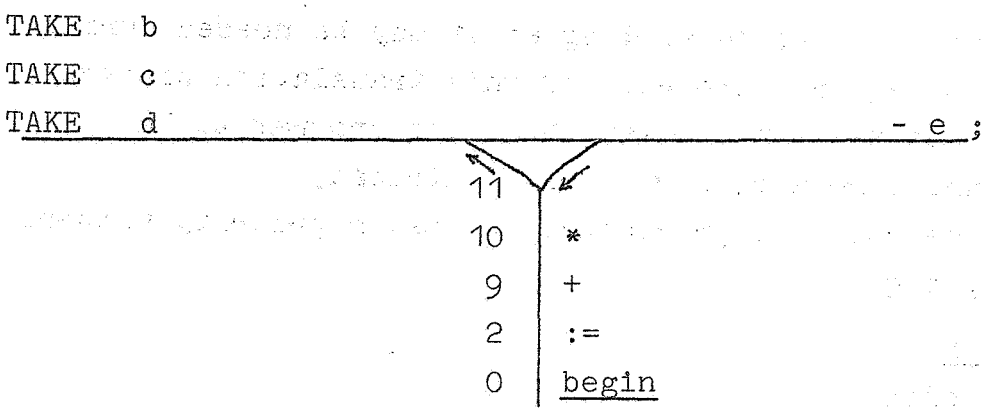


At the right the symbols of the ALGOL text come in in order from left to right, at the left the successive orders of the object program are produced. The rule is that incoming identifiers are sent to the output in the form of a TAKE order ("TAKE address of" if the identifier occurs to the left of the ":=" symbol, otherwise "TAKE value of"). Incoming operators receive their priority numbers and are then sent to the translator stack, but before the latter happens, operators in the translator stack are transported from it to the output as long as their priority number is greater then or equal to the proirity number of the new operator. For instance, at a certain stage of translation

"begin x:= a + b * c ↑ d - e;"

gives the following picture:

TAKE x (i.e. TAKE address of x in the next v)
TAKE a



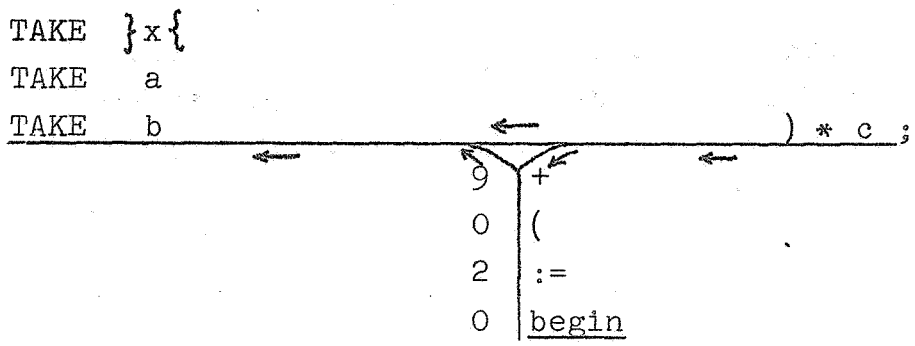
Identifiers are transported to the object program and operators, with their priority numbers attached to them, are dumped in the stack. We now consider the minus sign with priority number =9. Before this is entered in the stack, ↑ , * and + are removed in this order, giving rise to the orders "TO THE POWER", "MULTIPLY" and "ADD". Then follows the order "TAKE e" and when the ";" with priority number = 1 has been read, the two final orders "SUBTRACT" and "STORE" appear. The semicolon, being only a separator, need not be stored in the translator stack.

Up till now the priority numbers in the translator stack increased monotonically. The function of brackets is to interrupt this monotony. If an opening bracket "("is encountered in the ALGOL text, no emptying of the translator stack takes place, but the opening bracket is put on top of the translator stack with a priority number = 0, thus shielding all postponed operations until the corresponding closing bracket has been encountered. When a closing bracket is read from the ALGOL text, a priority number = 1 is assigned to it and the transport of operators from the translator stack to the object program takes place under control of the "new" proirity number 1. When this process stops, we must find the corresponding opening bracket (with priority number = 0) on top of the stack. Now the closing bracket, however, is <u>not</u> put into the stack like the other operators: the corresponding opening bracket is removed from it instead and translation goes on.

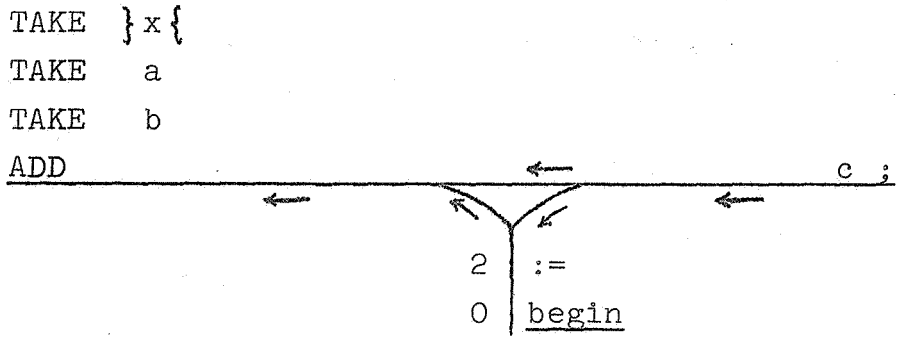
This is illustrated by the example

"<u>begin</u> x:=(a + b) * c;"

which gives rise to the following picture



and after the processing of the closing bracket



This way of processing brackets is perfectly sound. For its justification I should like to refer to a remark made earlier, that the function of a bracket pair is to shield its contents from its surroundings.

Now we shall show that "forgetting the surroundings, when an opening bracket is encountered" can be extended to include besides postponed operations, states of the translator as well. In expressions we have three kinds of opening brackets:

1) "("as arithmetic opening bracket
2) "("as opening bracket, announcing an actual parameter
3) "["as opening bracket, announcing a subscript

The translator can easily detect whether the opening bracket "(" is algebraic or not. It is a parameter bracket, if it follows immediately upon an identifier; otherwise it is an algebraic bracket.

On account of an opening bracket, the state variable CI (Comma Interpretation, see above) is redefined: it becomes = 0 for an algebraic opening bracket, = 1 for a square opening bracket and = 2 for the opening parameter bracket. But the previous value of CI, pertaining to the surroundings of the bracket pair, is to be preserved, because CI must be restored to it, when the corresponding closing bracket is processed. The obvious place to store it is in the translator stack: as soon as an opening bracket is found, a number of state variables (CI and others, left unmentioned) can be

dumped in the translator stack, before the opening bracket is put on top of them and then the state variables are redefined. As these state variables are stored in the translator stack in a fixed order, restoration at the processing of the closing bracket is a well defined process.
The redefinition CI:= 0 at the occurrence of an algebraic opening bracket removes the ambiguity when a closing bracket ")" is met: if CI = 0, then it is an algebraic closing bracket, otherwise CI = 2 and it marks the end of a last actual parameter. If it is desired to count the numbers of actual parameters or the number of subscripts, a counter value can be stored, set and restored in the same manner as the state variable CI. Note that one counter is sufficient: one counts either subscripts or parameters, but never both at the same time.

The translation of for statements and conditional statements, which can both be nested, uses the translator stack in an analogous way to store all the information that may exist simultaneously in as many incarnations as such statements occur inside one another. If we consider the statement (labels are included for the purpose of description):

"if B1 then A1: begin if B2 then A2: S1 else A3: S2 end
else A4: if B3 then A5: S3 else A6: S4;
A7: "

then the object program starts to evaluate the logical value B1. Then a conditional jump order to the point labeled A4 must be given: where in the object program this point will be is unknown at that time. Here we meet the problem of the so-called "future reference". The only thing we can do is to leave the address part of this conditional jump order undefined for the time being. But the translator makes a note of the address, where this undefined jump order in the object program has been produced. This note will be used, when translation has reached the point with label A4. Then a control combination, containing the address of the undefined jump order, can be inserted on the output tape, and for the read program that reads in the object program it is an easy matter to fill in the address digits of the conditional jump order.

But this note, specifying the address of the incomplete conditional jump order, originates when the symbol then is encountered

and must be kept until the translator has reached the corresponding else. In this range, however, another conditional statement may occur - as in our example - and this is the reason why such a note is stored in the translator stack. Just before the point of entry labeled A4 an unconditional jump to A7 must be produced. This second forward reference can be treated along exactly the same lines. A slight improvement of the above technique is to fill the address portions of these forward reference jump orders with a special, recognizable marker and we shall now give the reason for this.

Up till now we had forward reference jumps to points in the program which were (in principle) anonymous. But the problem of a forward reference also arises if we have a number of goto statements to a label that is still to come. The first time that a forward reference to such a label occurs, we produce a jump order with the chosen marker as its address digits and the address of this unspecified jump order is stored by the translator coupled to the label. At the next forward reference to that same label, we produce an undefined jump order but use its address digits to specify the place of the previous forward reference jump to that same label. And coupled to the label the translator only keeps a record of the address of the last forward reference to it. When the translation of the labeled statement actually starts, the control combination described above is inserted on the output tape. The reaction of the input program for the object tape to this directive becomes a little bit more complicated. The directive specifies an address where "the present place of storing" must be substituted. Before filling this in one checks whether the marker occupies these bits. If so, the process stops; if not, we find a new address where the present place of storing must be substituted, and so on.

In the case of a backward reference the translator can produce the definite jump order at once.