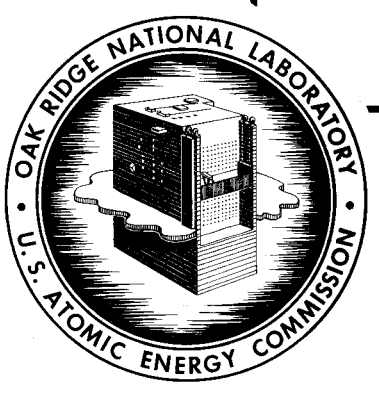
4

ORNL-3054
UC-34 - Physics and Mathematics

THE STRUCTURE OF AN ALGOL TRANSLATOR

A. A. Grau



OAK RIDGE NATIONAL LABORATORY
operated by
UNION CARBIDE CORPORATION
for the
U.S. ATOMIC ENERGY COMMISSION

Printed in USA. Price $1.50. Available from the
Office of Technical Services
Department of Commerce
Washington 25, D.C.

ORNL-3054

Contract No. W-7405-eng-26

Mathematics Panel

THE STRUCTURE OF AN ALGOL TRANSLATOR

A. A. Grau

DATE ISSUED

JAN 23 1961

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee
operated by
UNION CARBIDE CORPORATION
for the
U. S. ATOMIC ENERGY COMMISSION

ABSTRACT

The theory underlying a translator program, which is needed on a computer whenever programs written in the algorithmic language ALGOL 60[10][1] are to be used on that machine, is described. The approach followed is related to the recursive sequential methods outlined by Bauer and Samelson [9]. These methods have been simplified by the explicit use of recursive subroutines based on syntactic skeletons defining the ALGOL language.

Specifications for a translator based on these principles are given in machine-independent form. They include substantial advances over those used in the design of the first ALGOL translator for the ORACLE[12].

---

[1] The numbers in brackets refer to the corresponding items in the bibliography on page 53.

CONTENTS

THE STRUCTURE OF AN ALGOL TRANSLATOR

A. A. Grau

1. Preliminary Considerations

Introduction. In order to use ALGOL [3,10][1] for programming a modern high-speed stored-program computing machine, either a translator program or an interpreter program must be constructed for that machine. A translator program (more simply, translator) is a machine program that converts programs written in ALGOL submitted to it as input into programs in machine language; these are then executed as any other machine programs would be. An interpreter program, on the other hand, is placed into memory with the ALGOL program. Control is given to it, whereupon it takes the ALGOL instructions, one at a time, and translates and executes each in turn. The latter process obviously results in a relatively slow execution time. Because of this and other considerations, major emphasis is placed almost universally on translation rather than interpretation.

A translator for an algorithmic language such as ALGOL is relatively complicated. In recent years considerable efforts have been made to develop a systematic theory of translation which permits a consequent simplification of the translation process. Some powerful principles [9], could find use eventually not only in the translation of artificial languages such as ALGOL,

---

1 Numbers in brackets refer to the corresponding items in the list of references on p. 53.

but possibly also in the machine translation of natural languages.

In this report the theory of translation and detailed plans for an ALGOL translator are presented. The first section is devoted to introductory material dealing with programming languages and representations. Section 2 contains the general theory on which the plans are based, and section 3 contains the specifications for the translator written in a language consisting of ALGOL, augmented by additional primitive language elements. A degree of familiarity with ALGOL is assumed at this point. The self-explanatory nature of ALGOL is such, however, that this need not be comprehensive.

The first ALGOL translator for the ORACLE is based on the principles discussed by Bauer and Samelson. The plans given in this report, while based on the same principles, also benefit from experience gained in the construction of that translator. The refinements and simplifications which are included will be used in the construction of any new translator.

Scope of the Translator. In the design of the translator presented, one goal has been to include the handling of as many features of ALGOL 60, insofar as they are understood and unambiguous, as possible. The practicality of implementation on a machine with small memory such as the ORACLE has, however, also had to be considered. A further aim has been the clear exposition of basic principles.

These considerations have led to the apparent use of a single arithmetic mode in target programs, except in the automatic handling of subscripted variables. In most machines, this can be made to coincide with a floating-point mode of operation. Since quantities which have integral values

automatically yield in most machines integer-valued results under floating-point addition, subtraction, multiplication, and the entier function, this is not a real restriction.

No essential change in the translator is needed, however, if real and integer arithmetic are to be implemented in the target program as floating-point and fixed-point operation, respectively. The type of a variable is made part of its internal identifier. This information is supplied during the processing of type declarations. The principal additions to the translator will be in the macros EXU and EXB (p. 42). In these, additional program is introduced, which tests the mode of each operand, which provides for conversion from fixed-point to floating-point form whenever the operands are of mixed type or if the standard functions are applied to integer-valued expressions, which determines the proper mode of the arithmetic operation, and which tags the address of the result with its mode.

Input and output in the target program do not constitute part of the ALGOL reports. Therefore, no provision for them is included in the description of the translator. By way of illustration, the handling of these in the ORACLE translator is described separately.

The translation table provides for the implementation of blocks. The macros needed are outlined, but not detailed. Processing is discussed in a supplementary paragraph. If it turns out that blocks are little used in ALGOL programs outside of procedures, the effort needed in implementation could well have been directed elsewhere. In the case of the ORACLE, useful features of the ORBIT system [5, 6, 7] are tape files and provision for segmentation.

Provision for including these in an ALGOL translator appear to merit priority. The possibilities of using the block concept for including segmentation in target programs is being investigated.

The translation of procedures involves the fact that the ALGOL 60 report permits the use of recursive procedures in ALGOL programs. These are procedures that may directly or indirectly slave themselves on a new level without loss of information on the old. No provision is made in the language for distinguishing these from procedures which are not recursive and which can therefore be handled in a simpler way. If provision is made for procedures to be recursive a certain amount of manipulation and transfer of information is necessary on entry and exit. This results for short procedures in a considerably slower execution time.

ALGOL was primarily designed for use with numerical algorithms. In numerical work, recursion in a method is usually replaced by iteration in the program. The replacement of recursive concepts by iterative algorithms is easily done by programmers, while the mental processes they use cannot be fully simulated on a machine at this time. For numerical work recursive procedures do not seem to be seriously needed.

On the other hand, recursive subroutines are useful in the construction of translators, as this report will show. If the design of a "bootstrapping" translator, that is, one that can translate itself, is attempted, the implementation of recursive procedures is necessary.

Probably the ALGOL language should eventually incorporate provisions for specifying procedures to be either recursive or nonrecursive so that they

can be handled properly by the translator, with a resulting increase in efficiency of the target program.

In this report the implementation of procedures is limited to those whose parameters are restricted to names. This is no real restriction, since it is possible to program accordingly. If it is desired to use the value of an expression as an actual parameter, the value may be assigned to a new variable before the procedure call. In the procedure call, the name of the new variable is then used as actual parameter. The call of arrays by value may be circumvented by the same expedient. Functions, or procedures with values, are not permitted to change global variables.

Apart from such considerations, the translator described by this report will handle most features of ALGOL 60. This includes with complete generality all types of ALGOL 60 statements, and the new features such as conditional arithmetic and designational expressions. The ordinary array declaration is fully handled. Switch and own array declarations are somewhat limited.

Programming Languages. For the discussion below, it is necessary to make precise the concepts of programming language and alphabet. A programming language consists of a set of symbols, termed the alphabet of the language and a set of rules which state the manner in which sequences of alphabetic symbols called strings, may be formed which constitute valid instructions. The computer-oriented and algorithmic languages generally used in programming are, of course, examples of such languages.

Computer-oriented languages have as their distinguishing feature the fact that in their design primary consideration is given to making available

the facilities of the machine. Among these are the machine languages themselves whose alphabets consist of the symbols 0 and 1 in the case of a binary machine, with possibly the space and a few other control characters added, or the ordinary decimal digits in the case of decimal machines. At the present time, no coding is done in the raw language of a computer, since even so-called "machine coding" in the case of a binary machine is done using either the octal or hexadecimal digits. Even more common are the use of "symbolic" languages in which the machine operations are represented by mnemonic abbreviations using letters of the Latin alphabet and addresses are floating or symbolic. The target language used in this report may be considered as such a language.

Algorithmic languages have as their aim ease of application to the problem to be presented to the machine. Generally these languages are independent to a considerable extent of the hardware of the computer on which they are to be used, and so can be used on a variety of machines of different design. An ultimate ideal would be to state problems to a computer in the language of the technology involved itself. At this time, however, this appears not soon attainable. The alphabet of ALGOL, primarily useful for stating problems that involve numerical algorithms, since much of the language resembles the notation of ordinary algebra, includes the letters of the Latin alphabet, the mathematical symbols (such as +, /, =, etc.), and certain symbols which are needed for stating the dynamic requirements of computation.

In the translation process, it is sometimes necessary to treat a language as if it had a basic symbol set different from its own alphabet. In

this case, it is useful to consider the language considered from this new point of view as actually another language. An example of this is process-ALGOL discussed below.

Representation of ALGOL. For loading an ALGOL program into a machine for translation, the hardware representation designed for that machine is used. It may be pointed out that the hardware language referred to by the ALGOL report necessarily deals with the language employed on peripheral equipment before loading and not to the representation that ultimately will be used within the machine during processing. An example of a hardware language used on the ORACLE is found in the appendix. It is not our purpose to discuss at this point hardware languages further.

The representation used during processing may be related to the internal configurations induced by the loading of the hardware representation. It is not necessary that this be so, however, and in some instances this is not even desirable. Much depends on whether the machine has a large memory or a small memory and in the latter case on whether secondary storage is available.

If a machine has adequate high-speed storage and no auxiliary storage such as magnetic tape or drum, it is usually advisable to carry out the translation process in one pass using the representation induced by the hardware representation of ALGOL, or better, an internal representation isomorphic to the reference, not hardware, ALGOL. That is, there is a one-to-one correspondence between the symbols of reference ALGOL and their internal representations, so that one is able by transliteration alone to pass from one to the other, one

ALGOL symbol at a time, in such a way that it is not necessary to consider the context.

On most machines, secondary storage in the form of tapes and drums is available which permits the processing of programs in as many passes as required. The same can be carried out if there is a sufficient amount of internal memory. In this case it is possible to use, not a representation of either the hardware language or the reference language, but one which regards ALGOL from a simplified point of view. The conversion to this representation is easily carried out in one pass. This will be discussed in the following paragraph.

Process ALGOL. Letters and digits, though alphabetic symbols in ALGOL, do not have individual meaning. They are always parts of strings forming identifiers and numbers. Process ALGOL (p-ALGOL for short) is a view of the language which considers the alphabet to consist of the ALGOL delimiters and, instead of letters and digits, a set of possible internal identifiers. In other respects process ALGOL is identical with ALGOL.

The transition from ALGOL to process ALGOL is accomplished by replacing each ALGOL identifier string, each number string, and each truth value by a uniquely corresponding internal identifier. Numbers and truth values when encountered are converted if necessary and stored. In the final target program they will appear as a list of constants. The original external identifiers are not needed for the subsequent translation process or for the target program. They may be saved for use in diagnosis and print-out.

The use of process ALGOL as the primary vehicle of processing solves the problem also of what is to be done with identifiers of arbitrary length.

No restriction is made in ALGOL on the number of letters and digits which may be used for a given identifier. The use of internal identifiers to replace these permits the latter to be of fixed format. The necessity of handling strings of arbitrary length is therefore restricted to the replacement pass.

Representation for p-ALGOL. The latitude permitted in an actual choice of a representation for p-ALGOL can be used to simplify and facilitate the translation process. Since practically all computers handle information most conveniently in units of machine words, we will limit the representation immediately to one in which each alphabetic symbol of p-ALGOL (that is, each ALGOL delimiter and each identifier) is assigned a specific and unique machine word.

The differences between different identifiers do not have any affect on the flow of control in the translator, though identifiers will not be treated like operations and relations. The representation can make the similarity and the distinction immediately available when needed. Operations likewise for the most part are treated alike, though there are times when the difference between two operations or relations does affect the flow of control in the translator.

The latter is true for instance in the handling of algebraic expressions. In algebraic formulae, part of the bracket structure is understood from context. Thus a + b * c means a + (b * c) and not (a + b) * c. Also, x + y < a * 2 means (x + y) < (a * 2). The convention observed here is generally summarized in the statement that an otherwise ambiguous expression involving binary operations and relations is rendered well-defined by the

rules of precedence governing the operations and relations. Each operation and each relation has a precedence level; in the otherwise ambiguous expression, the operation or relation with the higher precedence is executed first. The precedence level of an operation or relation, since it will influence the flow of control of the translator, should be apparent in the representation.

At this point therefore we adopt the following representation for p-ALGOL. Each symbol of p-ALGOL is represented by a machine word, which for convenience is subdivided into three parts, P1, P2, and P3. P1 will denote the class of symbol to which a given character belongs: the classes are the class of identifiers, the class of operations and relations considered as a single class, and other classes each containing for the most part one or at most a few elements.

P2 will be used to differentiate subclasses. In the case of operations and relations this will denote the set of elements having the same precedence, for identifiers the type of arithmetic.

P3 is used to make the representation of a given element unique within a class or subclass.

For most of the classes of symbols, the class consists of a single subclass and in fact a single element. In these cases, the portions of the word P2 and P3 are left blank and not used in the translation.

2. Theory of Translation

Recursive Definitions. The concepts of ALGOL, as those of other algebraic programming languages, include those of variable, arithmetic expression, statement, and the like. The definition of many of these as given in the report [10] is inductive or recursive. That is, while part of the definition lists types of the structure being defined which can be expressed entirely in terms of entities which have a separate definition and do not contain as substructures structures of the type being defined, another part of the definition consists of rules which govern the construction of more complex examples of the structure from simpler ones.

By way of illustration, "simple arithmetic expression" has the syntactic definition ([10], 3.3.1):

<simple arithmetic expression> :: = <term> |<adding operator> <term>|
<simple arithmetic expression> <adding operator> <term>

"Term" and "adding operator" are defined elsewhere in the report. Any term or term preceded by an adding operator constitutes a simple arithmetic expression. However, given any simple arithmetic expression, a new simple arithmetic expression may be formed by appending an adding operator and a term. Thus, simple arithmetic expressions may contain as constituents other simple arithmetic expressions. It is this last feature in the definition that makes it inductive or recursive.

In any algebraic programming language, "arithmetic expression" must necessarily be defined recursively. In ALGOL, the definition of "statement"

also is recursive ([8], 4.1). In this case assignment statements, procedure statements, dummy statements, and (in ALGOL 58) stop statements are statements; all are defined separately. Other statements have statements as constituents and are built up in one of three ways from them: (1) compounding, (2) forming a conditional statement, and (3) forming a for statement.

The use of recursive or inductive definitions is common in certain branches of mathematics. It is at the heart of the postulational method. The integer-valued function of integers, factorial of n (Fact (n) = n!) is usually defined recursively:

$$\text{If } n = 0, \quad \text{Fact } (n) = 1;$$

$$\text{If } n \neq 0, \quad \text{Fact } (n) = n \cdot \text{Fact } (n-1) .$$

Effect on Translation. It is possible to proceed in many different ways in devising a routine to translate all simple arithmetic expressions. Basically the problem is to decompose the expression into parenthesis-free assignment statements. In order to do this, it is necessary to scan the expression for an "atomic" expression. By this we mean a simple arithmetic expression which consists of at most simple or simply-subscripted variables and a single arithmetic operation. A means for isolating such constituents is devised. Once one has been located and processed as an elementary assignment statement, it is replaced by a simple variable, and the process may be repeated. The finite character of the expression assures us that there is always an atomic constituent within it and that translation may be completed.

The scan described above may be improved. Instead of using multiple scans to locate atomic expressions, a single scan will suffice if in that scan,

information that cannot be handled immediately is stored systematically for subsequent use, and it can easily be determined when an atomic entity has been isolated.

It is at this point that the effect of the recursive nature of the concepts of ALGOL on translation may be considered. Implicitly there must exist within the translator subroutines corresponding to the various types of structures present in the language. Thus a subset of the translator has as its function the processing of simple arithmetic expressions. If we attempt now to design explicitly a subroutine for handling simple arithmetic expressions after the syntactic skeletons given in the report, the routine, if it is sequential as described above, at the outset has a three-way switch. In the first position, we encounter a term and control is sent to a corresponding term subroutine. In the second, we encounter an adding operator and a term. In the third, neither a term nor adding operator is encountered and we are faced with the problem of processing another simple arithmetic expression. It is necessary at this point for the subroutine to be able to slave itself on another level, without discarding the information still needed for final processing on the old level and remembering the point in the subroutine to which control must be sent on exit from the new level. A subroutine which can slave itself recursively on higher levels is generally called a recursive subroutine. It must be emphasized at this point that our conclusions apply not only to the processing of arithmetic expressions, but also to such matters as the processing of variables and statements.

Recursion vs. Iteration. A digression to a somewhat analogous numerical situation at this point is advisable. The factorial function is defined recursively, but it may be computed either recursively or iteratively. The computation of factorial n may from the definition be reduced immediately to the following recursive ALGOL procedure:

```
real procedure Fact (n); value n; integer n;
    if n = 0 then Fact := 1 else Fact := n • Fact (n-1) .
```

In machine coding programmers universally sense the difficulties in implementing in this manner a function recursively defined. In the programming of recursively defined mathematical functions the conversion to the iterative program is almost always made by them with little difficulty and usually results in a superior program. They write instead immediately the equivalent of the following ALGOL procedure:

```
real procedure Fact (n); value n; integer n;
    begin
        w: = 1; for k := 1 step 1 until n do w := w × k;
        Fact := w
    end
```

This iterative procedure is machine-wise definitely preferable since it minimizes storage.

The question naturally arises whether the translation process which by definition also calls for recursive subroutines is also better handled iteratively. In the past, the point of view has often been adopted that the

improvement that obtains in the mathematical case also holds here.

The situation is definitely somewhat different in the case of translation. Two drawbacks to attempting to reduce the programming of the translation of entities recursively defined to iterative procedures are, first, the processing is therefore necessarily non-sequential, and, second, the bookkeeping involved in the analysis may become quite involved. In our example, "term" is also defined recursively in terms of "arithmetic expression." Thus the probing for a true atomic entity may involve a complicated scanning problem. It will be seen that the iterative handling will have at least the same order of complication as the recursive process to be described. A definite advantage of the latter is that multiple passes through the original information are avoided by its systematic storage of needed information.

In the recursive treatment, there will be essentially a recursive subroutine S to handle simple arithmetic expressions and a subroutine T to handle terms. S may, during translation, slave S and T, and T may indirectly slave S. At each stage, this is done in such a way that information on one level required after the work of a slave is completed is stored before entry into the slave. This includes a record of the place in the master to which control will be returned. The various subroutines may be constructed almost immediately from an analysis of the syntactic skeletons defining the corresponding terms. The handling of the information contained in the original program is done sequentially. From the point of view of this method, a translator consists of a set of mutually recursive subroutines each of which

takes its pattern from a syntactic skeleton defining the language.

Push-Down Lists. The use of recursive subroutines in the translator leaves the problem of providing for the systematic storage of the information and flow of control required by the nesting of subroutines in which the translator appears at a given time.

Let subroutine R1 slave at some point in the translation process R2. R2 may be the same as R1 on another level. At the point where entry is effected into R2, the information in R1 that will be needed by it after R2 has done its work may be added to a list in which similar information has been stored from routines slaving R1. If R2 itself requires slaves, similar information will be added by it to this list. On exit from R2, however, the information needed by it and its slaves has been retrieved from the list, and since the work of R2 at this time is finished, the last meaningful material in the list is precisely that stored before R2 was entered.

The type of list thus induced is called a push-down list. In it the information last stored will be the first recalled. Between the two events other information may have likewise been stored and retrieved. It is also clear that by this device, information has been uncovered and becomes available at the moment that it again is needed. The contents of the push-down list at any time consist of all information stored by all subroutines in the current nesting arranged in the order in which they are slaved by each other.

In theory, only one push-down list is required for the storage of information needed by a set of recursive subroutines. In practice, it is generally desirable to split the information into several. We divide the

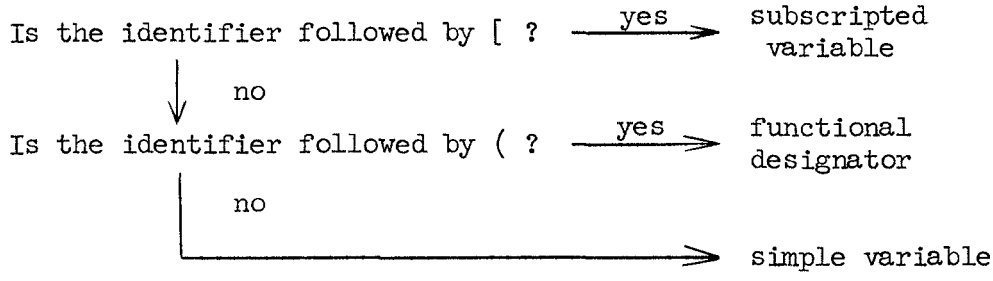
push-down information into two lists containing: (1) the information associated with the point to which control will be returned in each of the subroutines of the nesting, and, (2) all other information. The former list we shall call the control push-down and the latter the auxiliary push-down.

Translation and Syntax. A word is necessary about the relationship of syntactic definitions to the translator. The syntax of ALGOL indicates how valid statements and programs may be constructed in the language. That is, the syntax as given emphasizes synthesis. The problem of translation, however, is that of decomposing a validly written program into its constituent parts. In this connection, syntax must be regarded from an analytical point of view.

In order to apply the syntactical rules to translation, we must derive from them rules of analysis. In the design of an artificial language such as ALGOL the rules governing decomposition were involved, even though the report is written with synthesis as the primary consideration. The latter, of course, alone concerns the user of the language for programming his problem. In the natural languages, the two aspects are also present in syntactic rules, but the situation is much more complicated. It is safe to say that one of the difficulties encountered in automatic machine translation of natural languages is the fact that the rules governing decomposition do not always permit simple expression.

The contrast between the two aspects may be made clear by means of an example. Two important concepts of ALGOL are those of "variable" and

"functional designator" ([10], 3.1 and 3.2). In the report the syntactic descriptions state how valid strings of symbols to denote entities of these types may be built up. In translation, the problem becomes one of recognizing the two types of entities. In a sequential treatment, to which we limit ourselves, either may be present when an identifier is encountered; additional information is required before it is definitely known that the identifier is to be associated with a variable or a function. Accordingly, the rules of syntax lead to an analytical scheme for processing of the following type:



In essence, therefore, syntax hinges on "identifier" in translation while in writing programs for the machine in ALGOL, "variable" and "functional designator" are the primary concepts.

It is apparent therefore that one of the first tasks in designing a translator is the reorganization of the syntax for analytical purposes. When this is done, among the more important concepts which determine the course of processing statements are found to be:

1. Compound statement and block
2. Operand and identifier
3. Expression

4. Go to statement
5. Assignment statement
6. Conditional statement
7. For statement.

Corresponding to each of these we will construct what amounts to a closed recursive subroutine.

In some cases an entity of the type indicated is expected from previous considerations. In particular, an arithmetic or Boolean expression follows necessarily any of the following symbols: (, [:= if. The expression subroutine may be entered on a new level whenever one of these has been encountered and processed. In these cases, the anticipation of a structure permits a simplification of the translator. In other cases, such as for an identifier, entry to the subroutine can only be made after an identifier has actually been encountered in the incoming information.

Summary. Historically, much attention has been focused on iterative methods of translation. The methods used in FORTRAN [2] and the techniques outlined by Rutishauser [1] were essentially of this type. That the true usefulness of recursive methods were overlooked is not surprising in the light of the fact that for numerical processes, the iterative approach appeared always definitely the preferable one.

However, in translation, the reduction of recursion to iteration is not simply counting and forming a loop as it often is in the mathematical recursions. The techniques developed by Bauer and Samelson [8] are based on recursion and not iteration. The relationship of the symbol push-down (which

is essentially our control push-down) to the nesting of recursive subroutines was not stated, but was nevertheless implicit. Once the relationship of the Bauer and Samelson techniques to the explicit use of recursive subroutines based on syntactic skeletons is recognized, it also becomes apparent that for translation, the important and useful methods are those based on recursion and not those on iteration.

3. Specifications for the Translator

General Description. The translator, for which specifications are developed below, uses as its source language process ALGOL. For the processing of ALGOL itself, therefore, an additional program is required which first converts the hardware adaptation of ALGOL used on a given computer into p-ALGOL. This program depends on the machine and the hardware language used. However, it is relatively simple to design so that it need not be discussed here.

The translator is constructed following the theory outlined in section 2. It is, therefore, a collection of recursive subroutines based on syntactic skeletons. Two push-down lists are used for the storage of information. The problem of designing the translator reduces to that of writing the subroutines. A program for the translator is given below, which is in part written in ALGOL itself augmented by additional primitive elements. The basic switch is described by means of a table or matrix. A working translator for a given machine can be obtained by a translation of the program by hand and suitable coding to provide for the switch.

The recursive subroutines are composed of other subroutines, a few dozen in number. In order to avoid confusion, we introduce the term "macro" for any of these. They are essentially of three types: (1) those that manipulate the control push-down list and so determine the flow of control within the translator, (2) those that produce target program, and (3) those that provide for necessary bookkeeping and checking.

Certain lists play a leading role in the translator. Consequently notation is introduced to permit reference to them and their elements as

given in the following table:

List	Name	Nature of List Element	Item	Counter
Source program	Γ	Character of process ALGOL	γ	g
Target program	Π	Machine word or symbolic instruction	π	p
Variable table	Φ	Identifier and corresponding address	Φ	f
Control push-down	Σ	State	σ	s
Auxiliary push-down	A	Miscellaneous information (machine word)	α	a
Label table	Λ	Label, associated address, use address	λ	i
Temporary push-down	H	Address in target program	η	h
Array control list	K	Initial address and dimensions	κ	k

The control push-down Σ to which we have already referred may be used as an example of the way in which the notation introduced in the table is used for any of the lists. Σ denotes the list itself, while its elements $\sigma_1, \sigma_2, \ldots, \sigma_s$ are denoted by subscripted symbols; s is the length of the list at a given point during processing.

In terms of these lists, the function of the translator may be said to be to produce from the source program list Γ the target program list Π of machine or symbolic instructions. The remaining lists are used for storage. The list H is a push-down list in the target program which will be referred to by the translator only through its addresses η_h.

The control push-down determines the flow of control in the translator. It with incoming information determines the macros which will be executed and the order in which they will be executed. That is, the last element of the push-down σ_s and the current incoming symbol γ_g together determine a set of

macros which are to be executed. The relationship of the entities used in Σ to the syntax determining the recursive subroutines which constitute the translator is treated below. The translation process may be described by a double entry table in which the column headings are all possible states σ_s and the row headings all possible incoming characters γ_g. In the field of the table determined by a given state and a given character are listed the macros which are to be executed when the pair (σ_s, γ_g) is encountered. This table is called the translation table or matrix. Most of the fields are empty, since the corresponding combinations of state and character cannot occur in the processing of a validly written ALGOL program. If desired, an error subroutine may be listed in this space. If this is done, a rather complete check of the syntactical correctness of an ALGOL program is possible.

The translation program is basically the following:

```
begin           comment Initialize and set counters to zero--this
                includes at least the ones mentioned here;

                    p := ℓ := 0; g := s := 1;

                    if γ[g] ≠ 'begin' then error else σ[s] := <S0>;

next:               g := g+1;

Process pair:       Execute the list of macros listed in the table
                    under (σ[s], γ[g]);

                    go to next

end
```

In some cases the list of macros will involve transfers not following this normal flow of control. This is in particular true of the macro EOB, which tests for the end of the program. If the end of the program is reached, final exit from

this loop is effected, and the translator stops.

In some cases, a valid combination (σ_s, γ_g) will not call for the execution of any macros, but the normal flow of control around the loop is continued. In such cases, the word "next" found in the table denotes a stall.

The matrix given in this section is restricted to that needed for processing statements. An additional part of the matrix formed on the same principles may be used to process declarations. However, since the latter are in structure relatively bracket-free, they may equally well be handled in some other way.

Control Operations. There are five macros which perform control operations in the translator. These add to, delete from, or otherwise affect the control push-down. They are given with ALGOL-like code below.

1. Entry into a recursive subroutine, Ent (ω):

$$s := s + 1;\ \sigma[s] := \omega$$

2. Establish a new state within a subroutine, Ch(ω):

$$\sigma[s] := \omega$$

3. Exit from a recursive subroutine, Exit:

$$s := s - 1$$

4. Exit from a recursive subroutine and save the incoming character for processing on the preceding level, Rep:

$$s := s - 1;\ g := g - 1$$

5. Transfer to processing of current character and state, PP:

$$g := g - 1$$

The label "process pair" is the point in the translator at which the execution of the set of macros determined by the pair (σ_s, γ_g) and given in the translation table begins. In the first two subroutines, ω is a parameter which is furnished on entry to the subroutine in accordance with the specifications in the table.

Each time that one of the recursive subroutines is entered, a state is added to the control push-down list. Each time an exit is effected, a state is removed. Thus the number of elements in the control push-down at any time is the number of subroutines which are currently nested.

States. The recursive subroutines used by the translator and the states used in them are listed in this section.

1. Compound statement and block

   States: S0, S1, S2, N

   S0 — This state is entered into the control stack on encountering a begin. It remains until terminated by an end, at which time there is a test for end of program.

   S1 — This state is the block state. If a declaration is encountered in the S0 state, the state is changed to S1. It also is terminated only by end, but at that time there is also carried out the end of block manipulation.

   S2 — This indicates the statement state in either a compound statement or in a block. It remains, once it has been placed, for all the statements in a given block. It is removed by end which then is processed against the underlying S0 or S1.

   N1 — The neutral state is needed for comments. It is terminated by a following ; .

   N2 — This is a second neutral state for strings following end. It is terminated by a following ; , end, or else, which is tested also against the underlying state.

2. Identifier and operand

States: 0, I1, I2, I3, I4, P

0 The operand state is entered whenever an operand is expected. Operands are of two kinds: those beginning with an identifier, in which case the state is changed to I1 and those beginning with a left parenthesis, in which case the state is changed to P and a state E0 is added on the next level, since another expression is then expected.

I1 The state I1 is entered into the control push-down whenever an identifier is encountered in the processing of a program.

I2 This state indicates that the combination "I[" has been encountered previously. It assumes control only through being uncovered, at which time it indicates that an additional subscript has been placed in the next available temporary η_h. It is terminated on encountering a right bracket (]).

I3 This state indicates that the combination "I(" has been encountered. When it assumes control (after uncovering) it indicates that the value of a parameter has been placed into the next temporary η_h. It is terminated by a right parenthesis) at which time the procedure is evaluated and the value of function, if any, stored in a temporary, whose address is placed in the auxiliary stack.

I4 This state indicates that the address of the simple or subscripted variable or value of the desired function has been stored. In any case the last entry of the auxiliary storage is directly or indirectly the address of the quantity involved.

P The state P serves merely to keep track of parentheses within expressions. It can be uncovered, i.e., made the control element only when a right parenthesis is encountered, at which time it is deleted and the underlying state assumes control.

3. Expression subroutine

The expression subroutine uses the following states, plus those of the identifier subroutine which it slaves:

E0 E1 E2 E3 CE1 CE2 CE3

3. Expression subroutine (continued)

These have the following meaning:

E0 This state is added to the control stack at any point in the processing where an arithmetic or boolean expression is expected.

E1 When this state becomes the control element (always after uncovering another state) it indicates that an operand has been processed whose address is directly or indirectly stored in the uppermost cell of the auxiliary stack.

E2 When this state becomes the control element (always after uncovering another state) it indicates that k operands have been processed whose addresses directly or indirectly are stored in the k uppermost cells of the auxiliary stack. The k-1 operations that have also been processed are stored with E2 states in the control stack. A binary operation is always associated with each E2.

E3 When this state becomes the control element (always after uncovering another state) it indicates that an operand has been processed whose address is directly or indirectly stored in the uppermost cell of the auxiliary stack. The associated unary operation is stored with the state E3.

CE1 When this state becomes the control (always after uncovering another state) it indicates that the boolean expression following an if has been completely processed and its value is stored in the address stored in the uppermost cell of the auxiliary storage.

CE2 When this state becomes the control (always after uncovering another state) it indicates that the first arithmetic expression following then has been processed, and its value is stored in the address stored in the uppermost cell of the auxiliary storage.

CE3 When this state becomes control (always after uncovering another state) it indicates that the second arithmetic expression following else has been completely processed and its value is stored in the address stored in the uppermost cell of the auxiliary storage.

4. Go to subroutine

States: G L1 L2 L3 CG

G — The state G assumes control (by uncovering other states) when a designational expression (either a label or a switch setting) has been processed in an unconditional transfer. Any end of statement indicator terminates it, at which time the actual transfer order is written in the target program.

L1 — This state indicates that a designational expression is expected. It is changed to CG if an if is encountered in the designational expression. If an identifier is encountered it is changed to L2.

L2 — If a bracket is encountered in this state then a switch setting is being processed, otherwise the previously processed identifier is a label, so that at this point an end of statement indicator is encountered.

L3 — This is the switch setting state. The setting of the switch is determined when a right bracket is encountered. An end of statement indicator will terminate this state by repeat.

CG — This state is entered when a conditional designational expression is encountered. It assumes control on encountering an else. At this point a conditional transfer is written into the program. Then the state is reset to that expected of an unconditional transfer and the final label is treated as such.

5. Assignment statement

States: A1, A2

A1 — The state A1 with E0 is added to the control push-down whenever the character : = is encountered when the translator is in a state S2. Since in an assignment statement an expression is normally expected after this symbol, the state E0 is also added.

A2 — The state A2 is used for multiple assignments. Consequently, it is added only when the character : = is encountered while the translator control is state A1 or A2. As with A1, an expression is expected and thus the state E0 is added immediately. A2, like A1 assumes control only by uncovering. Both are terminated by an end of statement indicator.

6. Conditional statement

States: C1, C2, and C3

C1 The state C1 is added to the control push-down when the translator is in state S2 and if is encountered. Since if is always followed by a Boolean expression, the state E0 is added at the same time. C1 assumes control only when it is uncovered and this can only happen at the time the incoming character is then.

C2 When this is encountered, the state C1 is changed to C2. Since a statement is expected, the state S2 is also added to the control stack. C2 assumes control after the statement has been processed and a character else, end, or ; appears.

C3 The state C3 is set when else is processed in state C2. Again a statement is expected and so S2 is also added to the control. The state is terminated by an end of statement indicator, at which time the code for terminating the condition is written.

7. For statement

States: F0, F1, F2, F3, F4, F5

F0 This state is added in the state S2 when a for is encountered. It is terminated after the variable has been processed and stored and the : = is encountered. At that time the state is changed to F1.

F1 This state assumes control until either step, while, or do is encountered. An expression is processed during this state.

F2 This state copies the increment in a step-until for element. It is terminated by until.

F3 This state is terminated by , or do and completes the processing of a step-until list element.

F4 This state completes the processing of a boolean expression involved in a while list element. It is also terminated by , or do.

F5 This state is induced by the processing of do. It is terminated recursively at the conclusion of the processing of the statement subject to the for clause by an end of statement indicator.

Translation Matrix. In the following pages is given the translation matrix. The column headings consist of all possible states, and the row headings of possible incoming p-ALGOL symbols. For the sake of brevity, incoming characters which cannot form a valid pair with any state on a particular page have been omitted on that page.

The row heading OTHERWISE requires explanation. In some instances, many of the entries in a column are alike. It is possible to minimize the number of table entries by taking advantage of this in listing such an entry only once under OTHERWISE. It is understood then that if a pair (σ_s, γ_g) is encountered in processing with no entry in the column σ_s, the entry listed under OTHERWISE applies. Any column having an OTHERWISE entry will not give rise to an alarm. If a relatively complete check of syntax is desired, this device should not be used. In that case, the matrix may be rewritten by placing the entry now under OTHERWISE in place of all asterisks occurring in the same column since only these combinations should activate the OTHERWISE entry. It is then possible to report an error whenever (σ_s, γ_g) has no entry in the table.

Each of the entries in the table consists of a list of macros. Thus, under the pair (S2, I) is found the entry

STID | Ent(I1)

Whenever this combination is encountered, the macro STID is activated, followed by the execution of the control instruction Ent(ω) (for ω being I1).

TRANSLATION MATRIX

Compound Statement and Block

Assignment

Part 1

γ \ σ	S0	S1	S2	N1	N2	A1	A2
begin	Ent(S2)|PP	*	Ent(S0)	*	*		
for	Ent(S2|PP	*	CLO|Ent(F0)	*	*		
go to	Ent(S2|PP	*	Ent(G)|Ent(L1)	*	*		
if	Ent(S2)|PP	*	Ent(C1)|Ent(E0)	*	*		
stop	Ent(S2)|PP	*	STOP	*	*		
I	Ent(S2)|PP	*	STID|Ent(I1)	*	*		
end	Ch(N2)	EOB|Ch(N2)	Rep	*	Rep	*	*
else			Rep	*	Rep	*	*
;	Ent(S2)	Ent(S2)	Rep	Exit	Rep	*	*
: =			Ent(A1)|Ent(E0)	*	*	Ch(A2)|Ent(A1)|Ent(E0)	
:			LABEL	*	*		
Declarator	*	Ent(Dec)		*	*		
all others				*	*		
comment	Ent(N1)		Ent(N1)	*	*		
OTHERWISE	BBL|Ch(S1)|PP	Ent(S2)|PP		next	next	EV1|Rep	EV2|Rep

TRANSLATION MATRIX

Operand and Identifier

Part 2

γ \ σ	P	O	I1	I2	I3	I4
I		STID|Ch(I1)				
(		Ch(P)|Ent(E0)	PROC|Ch(I3)|Ent(E0)			
[			Ch(I2)|Ent(E0)			
then			*			*
step			*			*
while			*			*
until			*			*
do			*			*
,			*	STV|Ent(E0)	STV|Ent(E0)	*
]			*	STV|SUBS|Ch(I4)		*
)	Exit		*		STV|FUNC|Ch(I4)	*
end			*			*
else			*			*
;			*			*
			*			
ω			*			*
: =			*			*
:			*			*
OTHERWISE			Ch(I4)|PP			Rep.

TRANSLATION MATRIX

Expression

Part 3

γ \ σ	E0	E1	(E2,ω)	(E3,ω)
if	Ch(CE1)|Ent(E0)			
I	*			
(	*			
then		*	*	*
step		*	*	*
while		*	*	*
until		*	*	*
do		*	*	*
,		*	*	*
]		*	*	*
)		*	*	*
end		*	*	*
else		*	*	*
;		*	*	*
ω	Ch(E1)|Ent(E3,ω)|Ent(0)	Ch(E2,ω)|Ent(0)	COMPEX	*
OTHERWISE	Ch(E1)|Ent(0)|PP	Rep	EXB|Rep	EXU|Rep

TRANSLATION MATRIX

Go To Statement

Part 4

γ \ σ	G	L1	L2	L3	CG
if		Ch(CG)|Ent(E0)			
I		STID|Ch(L2)			
[			Ch(L3)|Ent(E0)		
then					Ent(L1)
]				SWITCH|Exit	
end	*		*	*	
else	*		*	*	CONTRA|Ch(L1)
;	*		*	*	
OTHERWISE	TRA|Rep		Rep	Rep	

TRANSLATION MATRIX

For Statement

Part 5

γ \ σ	F0	F1	F2	F3	F4	F5
I	*					
: =	Ch(F1)					
step		Ch(F2)				
until			Ch(F3)			
while		Ch(F4)				
,		A1|C1		A2|C1|Ch(F1)	A3|C1|Ch(F1)	
do		A1|B|Ch(F5)|Ent(S2)		A2|B|Ch(F5)|Ent(S2)	A3|B|Ch(F5)|Ent(S2)	
;						C|Rep
else						C|Rep
end						C|Rep
any other	*	*	*	*	*	
OTHERWISE	Copy V	Copy E1	Copy E2	Copy E3	Copy E2	

TRANSIATION MATRIX

Conditional Expression

Conditional Statement

Part 6

γ \ σ	CE1	CE2	CE3	C1	C2	C3
then	IF|Ch(CE2)|Ent(E0)			IF|Ch(C2)|Ent(S2)		
step			*			
while			*			
until			*			
do			*			
,			*			
]			*			
)			*			
end			*		THEN|Rep	THEN|Rep
else		CC|ELSE|Ch(CE2)|Ent(E0)	*		ELSE|Ch(C3)|Ent(S2)	
;			*		THEN|Rep	THEN|Rep
ω						
OTHERWISE			CC|THEN|CC1|Ch(E1)|PP			

Target Language. The purpose of a translator is to produce target program. Some of the macros have as their function the production of such instructions. In most instances, the instructions produced will be highly machine-dependent and so cannot be fully described here. In order to minimize the effects of this, an essentially one-address machine is assumed for the purposes of the report. However, the arithmetic operations produced in macros EXB and EXU are left in three-address form; in a one-address machine the single target instruction indicated in the plans will have to be replaced by a more extensive complex of instructions. For individual operations that are not machine operations, devices such as subroutine calls may well be used. In this case a final assembly program can incorporate the needed subroutines in the machine program.

The target instructions produced by macros are placed always in a pair of braces ($\{,\}$). The functions that we assume can be performed by our (fictitious) generalized machine are the following:

1. Arithmetic

A := (ωB)	The unary operation ω is applied to the contents of B and the result stored in address A.
A := B ω C	The binary operation or relation ω is applied to the contents of B and C and the result stored in address A. In the case of relations, the result is a Boolean value.
A := B	The contents of B are placed in A.

2. Non-arithmetic

STOP	The machine equivalent of a stop order.
TRA y	Transfer to the address stored in y.
CLA y	Clear the accumulator and add the contents of y.

ADD y	Add the contents of y to the accumulator, converting the addends to fixed-point if necessary. This is used only in address setting.
STA y	Set the address in the target instruction with address y.
TIT y	Transfer if value in accumulator is true to address in y; otherwise proceed.
TIF y	Transfer if value in accumulator is false to address in y; otherwise proceed.
STO y	Store the accumulator in address in y.
NOP	Stall.

Notation in Macros. Where possible, ALGOL 60 is used to describe macros. Description in natural language in the form of comments is added both to clarify the ALGOL program, where present, and also to describe the program needed in the case the latter has not been formulated because of machine dependence and other considerations.

Some notation to augment ALGOL 60 is used:

C(E)	The value of a variable, whose address is stored at an address which is the current value of E. This permits indirect addressing. An important case is the assignment v := C(v2) In this case v is assigned the value of the variable whose address is the current value of v2.
v := <α>	Here α is a symbol such as a state or a character of p-ALGOL. The statement means that the internal representation of the symbol in question is assigned to v as its value. This is machine dependent.
v := ⎧α⎫	The string representing the target instruction α is assigned to v, which in this case is a string variable.

(α, β, γ) — A list entry in the label table, where α is a label, β is the target program address associated with the label in the current block, and γ is a target program address at which the label is used. When the entry in the table is first made, either β or γ is blank. The amount of space allotted to an entry is machine dependent, and may be several words.

--- — Indicates that part of the information is to be furnished at a later time in processing.

Two ALGOL-like delimiters have been added to aid in the processing of for statements and procedures. These are used in corresponding statements:

SJ L — This statement is eventually translated into the target equivalent subroutine jump to the address corresponding to the label L.

SSE L — This statement is eventually translated into the target instructions to set a subroutine exit in the address corresponding to the label L.

Macros. The macros required in the translator, other than those that determine control, are listed below. They are grouped under headings corresponding to the subroutines in which they find their primary use. Each is identified by a label consisting of a mnemonic abbreviation which is used in the table. As parts of an ALGOL program for a translator, they must be used as procedures, i.e., closed subroutines; in the interests of brevity, however, the necessary procedure heading and the enclosing begin-end parentheses have been omitted.

1. Compound statement and block.

BBL: — comment This carries out the operations at the beginning of the processing of a block. Much of this deals with the bookkeeping involved in lists containing local variables and labels;

EOB: comment This carries out the corresponding operations at the end of the processing of a block. Among other things, permanent (or symbolic) addresses can now be assigned to all labels within the block local to it. Certain counters are reset. Finally this tests for the end of the program;

s := s-1; if s = 0 then go to end of program;

STOP: comment Write a stop order in target program;

p := p+1; π[p] := $\left\{\text{STOP}\right\}$;

2. Designational.

LABEL: comment Enter a label and its associated address in the label table;

ℓ := ℓ+1; λ[ℓ] := (α[a], p+1, ---); a := a-1

TRA: comment Write instruction for transfer in target program;

p := p+1; π[p] := $\left\{\text{TRA ---}\right\}$;

if α[a] is not sentinelled then begin

λ[ℓ] := ℓ+1; λ[ℓ] := (C(α[a]), ---, p); end

a := a-1;

SWITCH: comment This computes and sets address in a go to switch statement;

p := p+1; π[p] := $\left\{\text{CLA C}(\alpha[a])\right\}$;

p := p+1; π[p] := $\left\{\text{ADD } \eta_h\right\}$; h := h-1;

p := p+1; π[p] := $\left\{\text{STA p+1}\right\}$; set sentinel in α[a];

CONTRA: comment This writes a conditional transfer;

if α[a] is sentinelled then begin

π[p] := $\left\{\text{STA p+2}\right\}$; end;

p := p+1; π[p] := $\left\{\text{CLA } \eta_h\right\}$; h := h-1;

p := p+1; π[p] := $\left\{\text{TIT ---}\right\}$;

if α[a] is not sentinelled then begin

ℓ := ℓ+1; λ[ℓ] := (C(α[a]), ---, p); end

a : a-1;

3. Assignment.

EV1: p := p+1; $\pi[p]$:= $\left\{\text{CLA } \eta_h\right\}$; h := h-1; a := a-1

EV2

EV2: p : p+1;

if $\alpha[a]$ is sentinelled then begin

$\pi[p]$:= $\left\{\text{STO } C(\eta_h)\right\}$; h := h-1 end

else $\pi[p]$:= $\left\{\text{STO } C(\alpha[a])\right\}$; a := a-1

4. Operand and identifier.

STID: comment This stores the incoming "identifier" in the auxiliary push-down.

a := a+1 ; $\alpha[a]$:= $\gamma[g]$;

SUBS: comment The location of the element of an array is computed here. This depends on the k subscripts whose values are stored in $\eta[h-k+1]$, $\eta[h-k+2]$, ..., $\eta[h]$ and the information vector of the array A which is stored beginning at address m in the translator (or more generally in the target program); m itself is stored in $\alpha[a]$.

If the information vector consists of the k dimension m_i and the (theoretical) location A[0, ..., 0], the address may be computed in the target program by the Horner scheme:

address $A[i_1, \ldots, i_k]$:= address A[0, ..., 0] +

$(\ldots((m_i \times i_1 + i_2) \times C\, m_2 + i_3) \times C(m_3) + \ldots) \times m_{k-1} + i_k$;

This is stored by the target program in $\eta[h-k+1]$; h := h-k+1; $\alpha[a]$ is sentinelled.;

PROC: comment This procedure may be designed to deal with the parameter list that will follow in accordance with declarations.;

FUNC: comment A subroutine entry is written at this point to the closed subroutine corresponding to the called procedure. The actual code will depend on the method of subroutine entry used in the machine. Linkage will be in terms of the temporary level and information covered in declarations.;

STV: comment Store value of expression in the next available temporary. This is required whenever an identifier state is combined with an end of expression indicator;

```
if α[a] ≠ <η[h]> then begin

if α[a] is sentinelled then begin

      p := p+1; π[p] := { CLA  [h] } ; h := h-1;

      p := p+1; π[p] := { STA p+1 } ;

      p := p+1; π[p] := { CLA --- } end

else begin

      p := p+1; π[p] := { CLA C(α[a]) } end  h := h+1;

p := p+1; π[p] := { STO η[h] }  ; α[a] := <η[h]>  end;
```

5. Expression.

EXU: comment This writes code in the target program for the execution of a unary (arithmetic or boolean) operation;

```
if α[a] ≠ <η[h]> then h := h+1;

p := p+1; π[p] :=

        { η[h] := (ω C(α[a])) } ;

α[a] := <η[h]>
```

EXB: comment This writes code in the target program for the execution of a binary (arithmetic or boolean) operation or relation;

```
if α[a] ≠ <η[h]> then begin if α[a-1] ≠ <η[h]>

          then h := h+1 end else if α[a-1] = <η[h-1]>

          then h := h-1;

p := p+1; π[p] :=

        { η[h] := C(α[a]) ω C(α[a-1]) } ;

a := a-1; α[a] := <η[h]>
```

COMPEX: This incoming arithmetic or boolean binary operation or relation is tested for precedence against the one stored in the control push-down σ[s]. If the latter does not have lower precedence, it is executed;

```
if prec (σ[s]) ≥ prec (γ[g]) then begin

        EXB; Rep end

else begin Ent(E2, γ[g]); Ent(0) end
```

6. For statement.

Copy v, Copy E1, Copy E2, Copy E3:

These building blocks have as their functions the copying of the character γ[g] into a reserved space for four strings denoted by v, E1, E2, and E3. These are then used to construct strings which constitute ALGOL-like statements which are processed by the translator.

Clo: comment This is used to clear counters for the above sub-routines and in addition q := q+1;

A1: comment This processes a list element of the type v := E1. The following string is constructed from the lists in the spaces allotted for v and E1:

```
"v := E1; SJ Mq;"
```

This is then processed as if it were part of the ALGOL program.;

A2: comment In the same manner, a list element of type v := E1 step E2 until E3 leads to an expression of the type after u := u+1:

```
"v := E1; Lu: if (v-E3)*E2 ≤ 0 then begin

 SJ Mq; v := v+E2; go to Lu end;"
```

This is then also processed as if part of the program.

A3: comment This processes a list element of the type v := E1 while E2. The string generated and processed in this case after u := u+1 is

```
"Lu: v = E1; if E2 then begin SJ Mq; go to Lu end"
```

B: comment This constructs a transfer past the subroutine for the statement subject to the for clause and a subroutine entry to it;

u := u+1; p := p+1; π[p] := $\left\{\text{TRA ---}\right\}$; ℓ := ℓ+1;

λ[ℓ] := (Lu, ---, p); a := a+1; α[a] := <Lu> ;

u := u+1; p := p+1; π[p] := $\left\{\underline{\text{SSE}}\ \text{---}\right\}$; ℓ := ℓ+1;

λ[ℓ] := (Mq, p, ---); ℓ := ℓ+1; λ[ℓ] := (Lu, --- p);

a := a+1; α[a] := <Lu>

C: comment This constructs the exit from the subroutine enclosing the statement subject to the for clause.;

p := p+1; π[p] := $\left\{\text{TRA ---}\right\}$; ℓ := ℓ+1;

λ[ℓ] := (C(α[a]), p, ---); a := a-1; p := p+1;

π[p] := $\left\{\text{NOP}\right\}$; ℓ := ℓ+1; λ[ℓ] := C(α[a]), p, ---); a := a-1

C1: comment This clears the counters for Copy E1, Copy E2, and Copy E3;

7. Conditional.

IF: comment At this point, a test is made on the Boolean value;

p := p+1; π[p] := $\left\{\text{CLA } \eta\text{[h]}\right\}$; h := h-1;

p := p+1; π[p] := $\left\{\text{TIF ---}\right\}$;

α[a] := <p> ;

ELSE: comment The proper transfers are set following the first statement or expression of a conditional statement;

p := p+1; π[p] := $\left\{\text{TRA ---}\right\}$; ℓ := ℓ+1; λ[ℓ] := (---, p+1, ((α[a]))

α[a] := <p>

THEN: comment This stores information concerning transfer previously coded in the target program;

ℓ := ℓ+1; λ[ℓ] := (---, p+1, C(α[a])); a := a-1

CC: comment This serves merely to adjust the temporary and auxiliary counter in the case of a conditional expression;

if $\alpha[a] = <\eta_h>$ then STV;

a := a-1; h := h-1

CC1: comment This readjusts the temporary and auxiliary counters.;

h := h+1; a := a+1; $\alpha[a] := <\eta[h]>$;

One-Pass Translator. In the case of a translator where only one pass is desired and adequate memory space is available, p-ALGOL will not be used. In that case, the matrix can be enlarged and macros added to take care of the operations otherwise executed in a prepass. The additions to the matrix and the additional macros are summarized below. A considerable number of additional entries in the matrix will be required.

γ \ σ	Any applicable state[1]	IC	NC1	NC2	NC3
letter	Init|Copy|Ent(IC)	Copy			
digit	Init|Copy|Ent(NC1)	Copy	Copy	Copy|Ch(NC3)	Copy
10	Init|Copy|Ent(NC2)		Copy|Ch(NC2)		
+		*	*	Ch(NC3)	*
-		*	*	Copy|Ch(NC3)	*
OTHERWISE		IDT|Rep	NUMT|Rep		NUMT|Rep

Macros:

Init: i := 0

Copy: i := i+1; ν[i] := γ[g];

IDT: comment Pack the identifier string ν[1], ν[2], ..., ν[i]. Check the resulting identifier against the identifier table. If not there, assign an internal identifier or symbolic (real or pseudo) address I, which by STID will be placed in the auxiliary push-down.; γ[g] := <I>;

---

[1] Any state which has an entry for (I, σ[s]) in the translation matrix.

NUMT: comment Convert number string $\nu[1]$, $\nu[2]$, ..., $\nu[i]$ to machine representation. Check against number table. If not there, assign internal identifier or symbolic address I, which by STID will be placed in auxiliary push-down; $\gamma[g]$: <I>;

Detection of Syntactic Errors. If the form of the matrix that is used for processing is the one without the OTHERWISE feature, syntactic errors can be detected concurrently with the attempt to translate. Whenever a pair $(\sigma[s], \gamma[g])$ is encountered for which no entry is in the table, there is a syntactic error in the program. If, therefore, entries that diagnose and report errors are placed in all such otherwise vacant fields of the matrix a rather complete error monitor is possible.

For a machine with a relatively small memory, it may not be possible to have the additional code required by the error monitor in the memory at the same time as the processing portion. In other cases, translators are written by a group consisting of more than one programmer, and it is advisable to divide up the work into independent parts that can be constructed simultaneously. In either case, it is advisable to carry out the check for syntactic error in a separate pass independently of processing. This pass should, of course, precede actual translation, and can be carried out in process ALGOL.

The structure and coding of such a pass can be patterned after the main processing pass. The matrix without the OTHERWISE feature is used. In the fields where there are entries in the processing pass only the control macros are kept. Care must be taken with some other macros such as COMPEX, IDT, and NUMT where control functions are carried out within the macro itself.

Here the control must be abstracted from the rest and retained. The control push-down will operate exactly as in the processing pass. The auxiliary push-down is not used. A program will be considered validly written if the control function is properly executed throughout.

If such a diagnostic pass precedes processing, no loss of information results and considerable space is saved if the matrix used in processing contains the OTHERWISE feature.

Implementation of Blocks. If a program consists of a single block, the foregoing description of a translator is complete. No declarations will be found beyond the heading of such a block.

However, blocks will naturally occur also at least in the procedures which are implemented in the system. This language requires some additional planning. The simplest case is that in which no recursive procedures are permitted in the program, and consequently blocks will also not be recursive. In this case it is necessary only to provide for the storage of information concerning the memory requirements of the variables of the containing blocks. A subroutine within the target program may be used to provide for storage allotments to arrays within the block which do not have fixed dimensions. No adjustment will be needed on exit from the block, so that the problem of performing necessary operations on exit in the case of recursive blocks does not arise.

A block may be used recursively if it is part of a recursive procedure. If the dimensions are fixed, the own array can adequately be handled. If the dimensions are variable, complications are introduced into handling the arrays;

copy operations will be called for whenever the dimensions change. Such copy operations can be supplied, at the cost of increased running time. Whether the trouble is merited depends very much on the use to which the translator will be put in a particular installation.

Tables which must be handled in push-down fashion in the translation are the label table, since labels are local, and the identifier table. Local variable may be properly handled by the simple device of restricting the search of the identifier table to the portion to which the local variable is local. This can be done by making part of the identifier a serial number which indicates the depth of nesting of the block to which the variable is local, and then searching only to this level. An alternative is to search only from the end of the table when an identifier is encountered so that the last entry with this identifier is encountered first, this being the local one.

Labels are entered in the order found. At the end of the processing of a block the final addresses of all labels local to that block are known and therefore are assigned. Any labels to which assignments cannot be made are kept in the table, but the table can be shortened at this point. Alternatively the assignment of unspecified transfer orders may be made in the table, but not in the program. In a final pass the addresses are set in the target program, or the assignment may be delegated to a subsequent loading routine.

Procedures. We outline here in brief the translation of procedures and procedure calls which may be recursive, but which have only names as parameters. A further simplification can be made if arrays are considered always as global to the procedure.

The restriction of parameters to being only names is not severe. In the case of simple values, it is possible to write ALGOL statements which assign

the values of the desired expressions to new variables, whose names are then used as parameters in the procedure call.

Basic to the processing of procedures is the use of a reserve push-down H* in the target program. At any procedure call, the current contents of all cells of H ($\eta[1], \ldots, \eta[h]$) are copied into the reserve push-down, along with h and the address to which control is normally to be returned at the end of the procedure execution. In addition, h is then set equal to zero. In effect, therefore, at the beginning of each procedure execution the temporary push-down H is empty, and therefore no special means for translating are necessary within the body. At the conclusion of the procedure execution, the contents of H are restored. Any remaining elements of H due to the execution of the procedure must be moved to the end of the push-down list in this.

On an exit from the procedure other than the normal one, the return address will not be used. The contents of the temporary push-down must, however, be restored.

The handling of procedures in this way permits recursive subroutines. Since recursion in some cases can be of arbitrary depth, in the case of machines with small high-speed memory, it will be desirable to include in the manipulation of the reserve push-down H* the storage and retrieval of information from a secondary storage such as drum or tape. Corresponding to each formal parameter in the procedure heading will be a link-word and a cell of the push-down H. In the translation of a call, the addresses and pseudo-addresses of the actual parameters are placed in the link-words. In the translation of a procedure heading and body, provision is first made for transferring

the information in the link words into fixed storage locations assigned in H. Then in the rest of the translation, the addresses indirectly in H are associated with the formal parameters. Indirect addressing is necessary at this point, but causes little trouble even in machines which do not directly provide for it. Whenever a formal parameter is encountered in the body of the procedure, its address must be set from the contents of the corresponding definite cell of H. In the case of arrays, the information will of course be removed one step further, since the information stored in H in that case is itself indirect.

Some complications still arise when the parameters are chained from a procedure to one that it slaves. This can be worked out on similar lines if desired, or it can be prohibited.

Input and Output in Target Programs. Since ALGOL 60 does not include provision for input and output statements, such facilities must be designed to be used with the language by the group constructing a translator. This paragraph may be useful in such a design. Two ways of handling simple input and output suggest themselves immediately. The first uses the procedure approach; in this, certain identifiers are reserved for use with input and output procedures, whose bodies are not written in ALGOL. The second uses additional ALGOL-like delimiters for input and output.

The first ALGOL translator for the ORACLE used at Oak Ridge National Laboratory uses the following additional delimiters:

1. read
2. read array
3. punch

4. cr

The first three of these are used with a list of variables, names of arrays, or arithmetic expressions, respectively to form a statement having one of the following forms:

read v1, v2, ..., vk

read array a1, a2, ..., ak

punch E1, E2, ..., Ek.

The function of the first two is to assign the values appearing on a paper tape under the paper tape reader to the variables and arrays designated in the order written. Values are assigned to the elements of an array in lexicographic order of the subscripts.

The punch order transfers the current values in order to paper tape. The delimiter cr constitutes a statement which punches on paper tape a character that in printing activates the carriage return mechanism.

In general, more elaborate output format provisions are definitely desirable with an ALGOL system, but the simple output given here can be used until such formats can be designed and programmed. All of the output statements (with the exception of cr) are handled during translation by a subroutine entry with suitable linkword. In final compilation, a library subroutine is compiled into the target program.

References and Bibliography

[1] H. Rutishauser, "Über automatische Rechenplanfertigung bei programmsteuerten Rechenanlagen," Z. Angew. Math. Mech., 31 (1951), p. 255.

[2] J. W. Backus, et al., "The FORTRAN Automatic Coding System," Proc. Western Joint Computing Conference, Los Angeles, Calif., 1957.

[3] "Preliminary Report--International Algebraic Language," edited by A. J. Perlis and K. Samelson, Communic. Assoc. Comp. Mach. 1 (1958), no. 12, pp. 8-22.

[4] H. H. Bottenbruch, "Übersetzung von algorithmischen Formelsprachen in die Programmsprachen von Rechenmaschinin," Zeitschr f. math. Logik und Grundlagen d. Math. 4 (1958), pp. 180-221.

[5] A. A. Grau, et al., "Programmer's Manual, ORACLE Binary Internal Translator (ORBIT)," ORNL CF-59-9-20 (Sept. 22, 1959).

[6] A. A. Grau, Multi-Segment ORBIT Programs, ORBIT Memo 2, ORNL (1959).

[7] A. C. Downing, Magnetic Tape Storage Files, ORBIT Memo 3, ORNL (1959).

[8] Structure of the first ALGOL translator at the Institute für Angewandte Mathematik, Mainz, Germany, 1959. Unpublished notes.

[9] F. L. Bauer and K. Samelson, "Sequential Formula Translation," Communic. Assoc. Comp. Mach. 3 (1960), no. 2, pp. 76-82.

[10] "Report on the Algorithmic Language ALGOL 60," edited by Peter Naur, Communic. Assoc. Comp. Mach. 3 (1960), no. 5, pp. 290-314.

[11] The Structure of ALCOR, Institut für Angewandte Mathematik, Mainz, Germany, 1960. Multilithed report.

[12] ORACLE ALGOL Translator 1 (Preliminary Report). Multilithed Mathematics Panel Report., ORNL, Sept. 1960.

Appendix

ORACLE Hardware ALGOL

1. The character apostrophe (') is reserved for use as a delimiter indicator. It will have the punch pattern and keyboard location previously occupied by %.

2. Any delimiter given in reference language by an English word or phrase (underlined or italicized) is represented by the same word or phrase enclosed by apostrophes. Examples:

Reference:	go to	procedure	for
ORACLE:	'go to'	'procedure'	'for'

3. Five single-character delimiters of the reference language which were previously not ORACLE characters have been made available. These, with the characters that they replace and whose punch patterns they assume, are:

ALGOL character:	$_{10}$	[	]	↑	;
Previous character:	'	α	β	σ	Δ

4. For the following delimiters, substitutions are made:

Reference	ORACLE	Reference	ORACLE	Reference	ORACLE
$<$	'ls'	$\wedge$	'and'	$\div$	'div'
$\leq$	'lseq'	$\vee$	'or'	$\times$	*
$>$	'gr'	$\neg$	'not'	(	"
$\geq$	'greg'	$\equiv$	'equiv'	)	"
$\neq$	'nteq'	$\supset$	'implies'		

5. Any single-character delimiter not included above is the same as in reference language.

6. While identifiers may be of any length, only the first five characters have meaning to the ORACLE translator.

7. Number strings must be limited to ten digits.

8. The upper and lower case punch characters are used as needed to change case. The reader stop character is used to terminate each paper tape.

9. The ORACLE format characters, tabulator, backspace, carriage return, punch stop, and breakpoint, are ignored by the translator. They may be used as desired. The space is likewise ignored except in strings enclosed by a pair of quotes (").

10. The ORACLE characters which are not among those referred to above must not be used in punching ALGOL programs.

Acknowledgement. The attention of the author was first directed to the recursive sequential methods outlined by Bauer and Samelson [7] by F. L. Bauer while he was associated with the Laboratory in the autumn of 1959. The unpublished specifications for a translator [9] designed at Mainz, Germany, by M. Paul were a source of valuable information soon thereafter.

Thanks and credit is due also to H. H. Bottenbruch, with whom were held many stimulating and worthwhile conversations on a number of the topics covered in this report, and to E. J. Schweppe and D. R. Fitzwater, Iowa State University, for their critical reading of a preliminary draft.

ORNL-3054
UC-34 - Physics and Mathematics
TID-4500 (15th ed.)

INTERNAL DISTRIBUTION

1. C. E. Center
2. Biology Library
3. Health Physics Library
4-5. Central Research Library
6. Reactor Division Library
7-26. Laboratory Records Department
27. Laboratory Records, ORNL R.C.
28. A. M. Weinberg
29. L. B. Emlet (K-25)
30. J. P. Murray (Y-12)
31. J. A. Swartout
32. E. H. Taylor
33. E. D. Shipley
34. A. S. Householder
35. C. P. Keim
36. W. H. Jordan
37. S. C. Lind
38. F. L. Culler
39. R. S. Livingston
40. A. H. Snell
41. A. Hollaender
42. M. T. Kelley
43. P. M. Reyling
44. K. Z. Morgan
45. T. A. Lincoln
46. C. S. Harrill
47. H. E. Seagren
48. C. E. Winters
49. D. Phillips
50. M. L. Nelson
51. J. A. Lane
52. R. A. Charpie
53. M. J. Skinner
54. R. S. Cockreham
55. G. J. Atta
56. H. H. Bottenbruch
57. L. L. Bumgarner
58. H. P. Carter
59. E. C. Cooper
60. R. R. Coveyou
61. N. M. Dismuke
62. A. C. Downing
63. Manuel Feliciano
64. Walter Gautschi
65-84. A. A. Grau
85. ORNL - Y-12 Technical Library, Document Reference Section

EXTERNAL DISTRIBUTION

86. Division of Research and Development, AEC, ORO
87-696. Given distribution as shown in TID-4500 (15th ed.) under Physics and Mathematics category (75 copies - OTS)