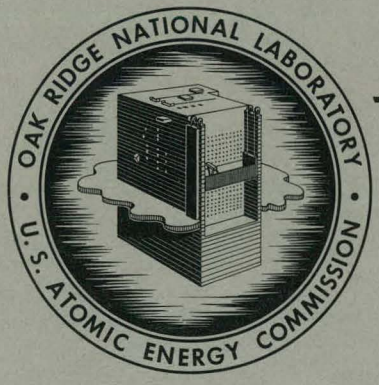
ORNL-3592
UC-32 – Mathematics and Computers
TID-4500 (28th ed.)

AN APPROACH TO ALGOL TRANSLATION

A. A. Grau
L. L. Bumgarner



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

DISCLAIMER

ORNL-3592

Contract No. W-7405-eng-26

Mathematics Division

AN APPROACH TO ALGOL TRANSLATION

A. A. Grau
L. L. Bumgarner

JUNE 1964

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

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

CONTENTS

Foreword

The principles discussed in The Structure of an Algol Translator (ORNL-3054) [19] have since the appearance of the Algol report [30] been applied at ORNL to the design of a translator for the Control Data Corporation 1604 machine [7] and at Duke University to the International Business Machines 7070 machine [2]. The present report covers the revision and extension of [19] based on the experience at ORNL. The translation table has been reformulated and expanded; new material, particularly on storage allocation and the implementation of procedures and blocks, has been furnished. Additional references to the fast-growing literature on the subject have been added.

On storage allocation and the implementation of procedures, acknowledgment is made to conversations with Professor K. Samelson, of Institut für Angewandte Mathematik, Mainz, Germany, during which fruitful ideas were germinated.

AN APPROACH TO ALGOL TRANSLATION

L. L. Bumgarner A. A. Grau

ABSTRACT

The approach to Algol translation described in this report is an extension and elaboration of that given in The Structure of an Algol Translator (ORNL-3054). In machine-independent form are given specifications for a translator for Algol 60. These specifications have been obtained in the design and construction of The Oak Ridge Algol Compiler for the Control Data Corporation 1604 (ORNL-3460). They may also be used with little modification to give an Algol translator for any present-day sequential machine.

---

1. Problems of Implementation

1.1. Introduction

In order to use a problem-oriented language such as Algol [29] [30][1] for programing a modern high-speed stored-program computing machine, either a translator program or an interpreter program, or a combination of the two, must generally be constructed for that machine. A translator program (or simply, translator) is a machine program that converts programs written in source language submitted to it as input into programs in machine language; these are then executed as are any other machine programs. An interpreter program, on the other hand, is placed into memory with the source language program. Control is given to the former, whereupon it takes the source language instructions, one at a time, and translates and executes each in turn. This process

---

[1] Numbers in brackets refer to the corresponding items in the list of references on pages 79-81.

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.

Translation and interpretation represent extreme approaches to the handling of the language. The possibilities inherent in an explicit combination approach will not be investigated here. Even when translation is emphasized, much having the character of interpretation must be done in the final machine program though it may not explicitly be called that.

A translator for an algorithmic language such as Algol has a relatively complicated structure. 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 of the principles evolved this way should prove useful not only in the translation of formal languages such as Algol, but also in the machine translation of natural languages into each other.

In this paper are presented a theory of translation for bracket-structure languages and moderately detailed plans for an Algol 60 translator based on that theory. A degree of familiarity with Algol is naturally assumed throughout, but the self-explanatory nature of Algol is such that this need not be exhaustive.

The first Algol translator for the Oracle [15] was based on principles discussed by Samelson and Bauer [34] and influenced by specifications drawn up at Mainz [31]. The principles outlined in this report are revisions and reformulations which have the benefit of the experience gained in the design and construction of that translator and

the problems which had to be solved in attempting to apply the original techniques. They have been used in the design of an Algol translator for the Control Data Corporation 1604 [7].

1.2. Hardware Representations of Algol

Practically no machine at this time has as a subset of its peripheral hardware symbol set the Algol reference character set. It is therefore necessary to design and use a hardware representation for Algol if Algol is to be used on that machine. The hardware language used on the Control Data 1604 is typical of such adaptations (see Appendix). The latter by mutual agreement is the ALCOR hardware language for machines using 48-character set peripheral equipment.

Experience in the use of Algol at Oak Ridge and elsewhere suggested certain principles which should be followed in the design of a hardware language [15]: (1) If a reference language symbol is also a hardware symbol, the latter is used to represent it; (2) if it is not, a substitution of a string of hardware symbols is made, and (3) one hardware symbol is reserved for "escape" usage. This philosophy has also been suggested by a report of users of KDF9 computers of the English Electric Company, Ltd. The basic advantages of these principles are that (1) the programmer may write in reference Algol, and if desired, the task of transliteration may be referred to the key punch operator with the instructions: "If a symbol in the manuscript exists on the keyboard, use it; if not refer to the card containing the rules and make the appropriate substitution." [14], and (2) if more hardware characters become available, the modification of the hardware language is obvious.

1.3. Intermediate Language Forms

The representation of Algol used during translation may, of course, be related to the internal configurations induced by the hardware representation. It is generally better to use a representation which is isomorphic to an adaptation of the reference language rather than to the hardware language: the representation is so chosen that there is a one-to-one correspondence between the internal patterns and the reference Algol symbols, so that it is possible by simple transliteration alone to pass from one to the other.

Theoretically, the problem of translation is to convert from reference Algol to target language. Practically, reference Algol must be converted into hardware language by hand (either by a programmer or keypuncher), and then hardware language must be reconverted to an internal representation of reference language or a useful adaptation of it; translation proper then consists of converting this into target language. It will be noted that a number of language forms actually are in view. It is advisable to carefully distinguish between these, by referring to them by distinctive names. The practical source language used within the machine may be called process Algol, or p-Algol.

Letters and digits are alphabetic symbols in Algol which do not have individual meaning; syntactically they occur only as parts of strings that constitute identifiers, labels, strings, and numbers, which are basic syntactic units. The alphabet of process Algol may be the Algol delimiters and a set of internally generated identifiers, called entities to distinguish them from the original Algol identifiers. Entities correspond to identifiers, labels, numbers, truth values, and

strings in Algol. Of these, identifiers, labels, and numbers in their original form are made up of letters, digits, the decimal point, and the delimiter $_{10}$ which are not retained as part of the alphabet of process Algol. It is convenient to class these structures together since in many connections they act alike.

The conversion from hardware Algol to process Algol includes the replacement of each identifier string, label string, number string, proper string, and truth value by a corresponding internal identifier. Numbers may be converted if necessary, and they, strings, and truth values are stored for later incorporation into the target program. The original external identifiers have no inherent meaning, so that, apart from diagnostic printouts, they are not further needed in translation or execution.

Since the internally generated entities may have a fixed format, the use of process Algol also circumvents the problem of handling identifiers of arbitrary length.

1.4. Representation of Process Algol

The choice of a representation for process Algol permits latitude which may be used so as to facilitate translation. Nearly all computers handle information most conveniently in units of fixed size. One common unit is the machine word. A natural choice for a representation is that in which each alphabetic symbol of process Algol, that is, each delimiter and each entity, is assigned a uniquely corresponding machine word. Some computers have other units which may conveniently be used instead. In that case, what is said about machine words below applies equally well to such units.

Since entities are handled in a manner different from that used for delimiters, the representation can make the distinction between the two clear. It is also necessary to distinguish between simple variables, numbers, arrays, truth values, strings, labels, switches, and procedures. The relative precedence of two operations affects the flow of control at certain points in the translation. It therefore is desirable to have as part of the representation the precedence level of all operations and relations.

The following representation takes all of these considerations into account. Each symbol of process Algol is represented by a machine word consisting of three syllables, P1, P2, and P3. P1 denotes the class of symbol; classes include entities, operations and relations (together a single class), and other classes with for the most part one or a few elements. P2 denotes a subclass when this is needed. The subclasses of the class of operations and relations are those with a common precedence level. The subclasses of entities are determined by type (integer, real, Boolean) and character (simple variables, arrays, strings, truth values, procedures, switches and labels). P3 may consist of a serial number; it may be convenient in the case of entities to let this also be a relative address. Where a class consists of a single subclass, P2 is not used, and where a subclass has a single element, P3 is ignored.

1.5. Number Representation

In the present report, there is an apparent use of a single arithmetic mode in the target program. In most machines, if desired, the use of a single mode can be made to suffice. This will usually coincide with the floating-point mode. Under it, the addition,

subtraction, or multiplication of integer-valued operands will yield integer-valued results. The use of a single mode avoids a question of implementation discussed below.

Some change in the translator is needed if real and integer types are implemented in the target program as floating-point and fixed-point numbers, respectively. The type of a variable is determined at the time type declarations are processed and can be made part of the representation of entities. In the processing of arithmetic expressions, it becomes necessary to test the type of each operand, provide for the conversion from fixed-point to floating-point whenever the types of the operands are mixed, determine the proper mode of arithmetic operations and tag intermediate results with the proper type. Similar provisions are needed to handle the application of the standard functions. The coding of these decision programs, while relatively extensive, is conventional.

1.6. Block Structure

An important feature of Algol is the block concept, which provides an obvious means to economize in the allocation of storage to variables and intermediate results. On machines where it is necessary to make much use of secondary storage, there is a possibility of using the block structure also for segmentation of programs and the storage of large arrays. A mechanism which may be used to allocate storage to variables in an economical manner is described in section 3.2.

1.7. Procedures

Algol 60 permits the use of recursive procedures. These are procedures which may directly or indirectly call themselves on a new level without exiting or loss of information on an old. Procedures which are so used are not distinguished from those which are not. Otherwise, procedures not used recursively could be handled by simpler linkage mechanisms in the target programs. The recursive use may also be of a sort that cannot be determined at translation time. Provision for the recursive use of procedures requires manipulations on entry and exit which when applied uniformly result in a considerably slower execution time for short procedures.

In numerical work, for which Algol was originally designed, recursion in a method is generally replaced by iteration in an algorithm. The replacement of recursive concepts by iterative algorithms is easily done by programmers, while the mental processes they use cannot at this time be fully simulated on a machine. For numerical work, recursive procedures are not seriously needed. They are therefore not considered in the present treatise.

On the other hand, recursive subroutines are useful in the construction of translators, and here they are not conveniently replaced by iterative ones. If the design of a "boot-strapping" translator, that is, one that can translate itself, is attempted, the implementation of recursive procedures is necessary.

1.8. Static and Dynamic Handling

It is important to distinguish between Algol features which may be processed at the time of translation and others which must be handled at the time of execution of the program. The former may be said to be

handled statically and the latter dynamically. A fully efficient object program is one in which all features which can be handled statically are so handled.

An example of the difference is furnished by the allocation of storage to simple variables and that to arrays with variable subscript bounds. The first may be done at translation time, i.e., statically, while the second is necessarily carried out during the execution of the program, i.e., dynamically.

The question of static versus dynamic handling arises from the treatment of number type. In most implementations of Algol, quantities declared to be of integer type are represented by fixed-point numbers and those of real type by floating-point numbers. The expression a ↑ i, where both a and i are of type integer, can be of either type depending on the value of i. To handle this strictly when numbers are implemented in this way requires the object program to keep track of type dynamically. For other than research purposes, this is quite undesirable, and a slight change in the definition of a ↑ i is generally made to permit the handling to be static. The possibility does exist of redesigning the representation to avoid this problem [17]. It must be pointed out that the definition of exponentiation in Algol is precisely the one that is used in mathematics. The difficulty here is due to hardware. The use of two disjoint classes of numbers in a computer is mathematically undesirable. Mathematically, integers are also real numbers. The design of some recent computers includes a number representation which obviates the difficulty.

There are cases where a choice between static and dynamic handling is not so critical. For example, in computing the value of a

Boolean expression, it is possible to build a truth table at translation time, leaving the object program simply to choose the correct entry when the values of the variables are known [5]. In effect the Boolean expression is then evaluated statically.

1.9. Machine Limitations

On practically any of the present generation of machines there is some feature which makes the implementation of some Algol device lead to a degree of inefficiency. It is not desirable to redefine the language because of this, since machines are still undergoing considerable modification from one generation to the next. Mechanisms exist for implementing fully almost all features of Algol on existing machines [10].

For the average present-day machine, practical considerations lead to (1) the exclusion of the dynamic handling of variable type, (2) the exclusion of the recursive use of procedures, and (3) some limitation on own arrays. These restrictions to Algol 60 do not seriously affect the power of the language.

2. Theory of Translation

2.1. Recursive Definitions

The concepts of Algol, as those of other algorithmic programing languages, include those of "variable," "arithmetic expression," "statement," and the like. The definitions of many of these as given in the report [29] are inductive, or recursive. By this is meant that while each definition lists strings of syntactic units which make up the type of structure defined, some of these consist entirely of syntactic constituents which have a separate definition, and others are in essence

rules which govern the construction of more complex examples of the structure from simpler ones.

For instance, "simple arithmetic expression" has the syntactic definition ([29], 3.3.1):

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

Here there are listed three types of strings which constitute simple arithmetic expressions. "Term" and "adding operator" are defined elsewhere in the report. The first two strings state that any term, or term preceded by an adding operator, constitutes a simple arithmetic expression. However the last string states that given any simple arithmetic expression another simple arithmetic expression is formed from it by appending an adding operator and a term. Simple arithmetic expressions may therefore contain as constituents other simple arithmetic expressions. It is this feature that makes the definition recursive.

In any algorithmic programing language, "arithmetic expression" must necessarily be defined recursively. In Algol, the definition of "statement" also is recursive ([29], 4.1). In this case, assignment statement, dummy statement, procedure statement, and go to statement are separately defined. Statements of the following types have statements as constituents: (1) compound statement, (2) block, (3) conditional statement, and (4) for statement.

The use of recursive definitions here is an extension of a similar use in mathematics, where it is quite common. For instance, the integer-valued function of integers, factorial n, is usually so defined:

If n = 0, then Fact (n) = 1;

If n > 0, then Fact (n) = n · Fact (n-1).

2.2. Effect on Translation

A basic problem of translation is the decomposition of an expression into the equivalent of parenthesis-free assignment statements. One can proceed in many ways to devise a routine to do this. The expression may be scanned for an "atomic" subexpression. This consists of a simple arithmetic expression with simple or at most simply-subscripted variables and a single operation or relation. The scan requires a means for isolating such constituents and replacing them by simple generated variables. At the same time, a corresponding elementary assignment statement is generated and listed. The process is repeated until the decomposition is complete.

The scan described requires in general many passes for the complete processing if the expression is at all complex. Instead of multiple scans, a single scan will suffice if in that scan information that cannot be processed immediately is systematically stored for subsequent use.

At this point the effect of the recursiveness of the concepts of Algol may be considered. However it may be organized, in effect there must exist within the translator subroutines which correspond to the various types of structures present in the language. One subset of the translator, for example, has as its function the processing of simple arithmetic expressions. In a sequential treatment, this subroutine provides for three alternatives at the outset corresponding to the

syntactical alternatives in the definition. First, a term may be encountered, in which case control is given to the corresponding term subroutine. Second, the expression may begin with an adding operator which means that this is followed by a term. The third possibility involves the processing of another simple arithmetic expression which is a subexpression of the given one before the handling of the full expression can be completed. Thus the subroutine must be able to call itself on another level, without discarding the information still needed for final processing on the current level. A subroutine which can call itself on recursively higher levels without loss of information on previous ones is generally called a recursive subroutine. The conclusion we draw at this point, that it is necessary in effect to have recursive subroutines in a translator, follows not only from the consideration of arithmetic expressions. In Algol the definitions of "variable" and "statement" require a similar recursive property of the translator. A translator is necessarily a set of recursive subroutines based on syntactic structures defining the language.

2.3. Recursion Versus Iteration

A digression to discuss a somewhat analogous numerical situation at this point is worthwhile. The factorial function is defined recursively, but it may be computed either recursively or iteratively. The computation of factorial n reduces immediately from the definition to the following recursive Algol procedure:

```
real procedure Fact (n); value n; integer n;
    if n < 0 then wrong use else if n = 0 then Fact := 1
                 else Fact := n × Fact (n-1)
```

Here "wrong use" is a procedure which monitors the fact that the parameter is not permissible.

In machine coding, programmers universally sense the difficulties that arise in efforts to implement the function in this way. In the programing of recursively defined mathematical functions the conversion to the iterative program is made with little difficulty and results generally in a much superior program. If a procedure calls only itself recursively, the execution may in fact always be made into an iteration. In this case, we may write instead the equivalent of the following procedure:

```
real procedure Fact (n); value n; integer n;
begin integer w,k;
    if n < 0 then wrong use;
    w := 1; for k := 1 step 1 until n do w := w × k;
    Fact := w
end
```

This iterative procedure is machine-wise preferable since it defines in advance all storage requirements.

The question arises whether the translation process, which appears to require recursive subroutines, might not also be handled better by iterative substitutes. In the past the opinion has often been expressed that the improvement that obtains in the mathematical case may also be made in this situation. However, the situation is definitely different in the case of translation. Three drawbacks to the attempted reduction to iterative techniques are that (1) the processing becomes necessarily nonsequential, (2) the bookkeeping involved in the analysis

may become involved enough to offset any possible gain, and (3) the recursion involves several different subroutines which may nest in almost arbitrary order. In our example of the simple arithmetic expression, a closer examination shows that "term" is defined in terms of "arithmetic expression." The probing for a true atomic entity becomes a complex scanning problem. The iterative handling will have at least the same order of complication as the recursive processing to be described. Besides being easier to grasp once the mechanism is understood, the latter has the added advantage that multiple passes through the original material may be avoided.

In an explicit recursive treatment there is essentially a recursive subroutine S to handle simple arithmetic expressions and a subroutine T to handle terms. S may, during translation, call either S or T, and in turn T may call S. At each stage information on one level required after the work on a second level is completed is stored before the call. This information includes a record of the place in the master to which control must be returned. Once suitable provision is made for the necessary storage of information, the various subroutines may be constructed almost immediately from an analysis of the syntactic skeletons defining the corresponding terms. The handling of the information is then sequential. From this point of view, a translator consists explicitly of a set of mutually recursive subroutines each of which takes its pattern from a syntactic skeleton defining the language.

2.4. Push-down Lists

The use of recursive subroutines in a translator requires the solution of the problem of providing for the systematic storage of information required by the subroutines in a nesting at a given time. Let subroutine R1 call at some point in the translation process subroutine R2. R2 may be R1 on a new level. When entry is made into R2 the information in R1 that will be needed by it after R2 finishes its work may be added to a list in which similar information has been stored by subroutines on previous levels of the nesting. If R2 itself calls new subroutines, similar information will be added by it to this list. On exit from R2, however, the information needed by it and the subroutines it called will have been retrieved. The last meaningful material in the list is precisely that stored before R2 was called.

A close relationship therefore exists between recursive subroutines and a type of list called a push-down list. In it information is stored in last-in first-out fashion. Between the storage and recall of any item in the list other information may likewise have been stored and recalled. Information is at the end of the list whenever it is needed. The contents of the push-down list at any time consist of all information stored by all subroutines currently in a nesting, arranged in the order in which they have called each other.

Theoretically only one push-down list is required for the intermediate storage requirements of a set of mutually recursive subroutines. Practically the information is usefully split into several. In our case we shall make use of two: (1) one containing the information associated with the point to which control will be returned in each of the subroutines

of the nesting, and (2) one containing the remaining information. The first we shall call the control push-down and the second the auxiliary push-down.

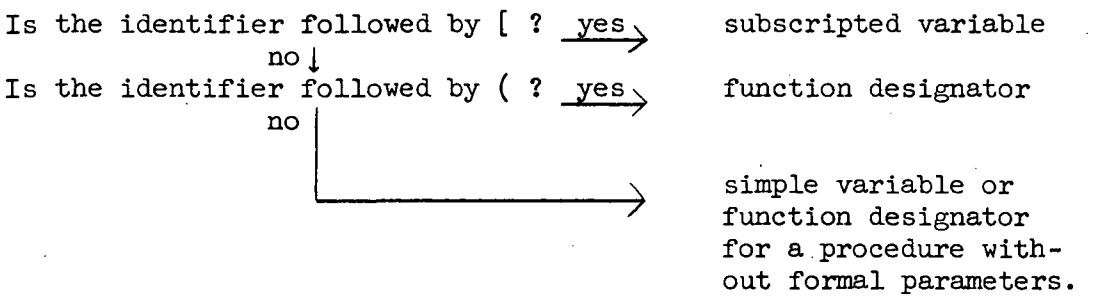
2.5. Translation and Syntax

The syntax of Algol indicates how valid statements and programs may be constructed from Algol symbols. In this the syntax emphasizes synthesis. The problem of translation, however, is that of decomposing a validly written program into its constituent parts. Here syntax is regarded from an analytical point of view. To have syntactic rules applicable to translation, it is necessary to derive rules of analysis from the rules of synthesis.

The design of a formal language such as Algol takes into account the necessity of analytical considerations, even though the report is written with synthesis in primary view. The latter, of course, alone concerns the user of the language. In natural languages the two aspects are also present, but the situation is more complicated. One difficulty encountered in automatic machine translation of natural languages into each other is the fact that the rules governing decomposition do not permit simple expression.

An example will make clear the contrast between the two aspects. Two important concepts of Algol are those of "variable" and "function designator" ([29], 3.1 and 3.2). Syntactic descriptions in the report show how valid strings of symbols to denote entities of these types may be built up. In a sequential translation process, the problem becomes one of recognizing the two types of entities: either may be present

when an identifier is encountered. Additional information is required to determine which is at hand. The rules of syntax in this case lead to an analytical scheme of the following kind:



In translation, syntax hinges on "identifier" while in writing programs for the machine in Algol, "variable" and "function designator" are the primary concepts.

One task met in designing a translator is the reorganization of the syntax for analytical purposes. The task may be mechanical to some extent and made part of the translator [25]. The mechanization of this task in such a way as to obtain desirable speed and efficiency remains to be worked out. Among the more important concepts which determine the course of processing statements are:

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 construct the equivalent of a closed recursive subroutine.

Some of the subroutines can make use of anticipating devices to both speed up and simplify the translation. This is true when a syntactic skeleton indicates before it is encountered what structure will follow in a validly written program. For example, when in certain contexts any of the symbols

(, [:= if step until

are encountered in a sequential examination, an expression (arithmetic or Boolean) must follow. The expression subroutine is constructed in such a way that it is entered on such anticipation. On the other hand, compound statements, blocks, assignment statements, and conditional statements cannot be anticipated. While most of these are recognized from the first symbol in the structure (apart from labels), the assignment statement is not fully determined until := is encountered because the presence of an identifier may also mean that a statement label is being processed.

2.6. Summary

Historically much attention has been focused on iterative methods of translation. The methods used in the original Fortran [1] and the techniques outlined by Rutishauser [33] were of this type. The true usefulness of recursive methods was overlooked, most likely, because for numerical processes more efficient iterative algorithms could usually be substituted for recursively defined processes, because the reduction of recursion to iteration generally is simply counting and forming an appropriate loop. In translation this is not so. The techniques developed by Samelson and Bauer [34] are based on recursion. The

relationship of the symbol cellar (which is related to our control push-down) to the nesting of recursive subroutines, while not stated, was implied. Once the relationship of the Samelson and Bauer techniques to the explicit use of recursive subroutines based on syntactic skeletons is recognized, it becomes apparent that for translation the important and useful methods are those based on recursion. It is also at that point possible to reformulate the methods of Samelson and Bauer to give more directly the relationship of the subroutines to the syntactic skeletons with a resulting simplification of the process.

3. Techniques of Implementation

3.1. General Structure of the Translator

An early consideration in the design of a translator is the choice of target language. This may be machine code or an intermediate language. In the latter category are the symbolic assembly languages as well as machine-independent languages (for an example of the latter see [16]).

The use of an intermediate form permits the postponement of certain functions to another program such as an assembler or a loader. In some cases this expedites such things as the use of a general control system and a library, the separate translation of procedures into a relocatable form, and mixing with other languages.

Translation directly to machine code is likely to be considerably faster. With a fast translator, it may be feasible to translate a program each time it is run. Programs are then checked out at the source language level.

If it is desirable to construct the translator so as to provide for an option of either machine code or assembly language output, little additional effort is necessary. The specifications for an Algol translator presented in this paper indicate the facilities required.

Another consideration in the structure of the translator is the number of "passes" to be made during the translation processes. By a pass is meant a transformation of a sequence of input symbols to a sequence of output symbols. Here available memory size is a controlling factor. If the memory is small, it may be necessary to segment the translator, and several passes may be required. With a large memory the number of passes can be kept small, though it is possible that some complexity may be avoided with a larger number of passes. Processing to improve target program efficiency may involve extra passes (see [22]).

3.2. Storage Allocation

The mechanism described here is for allocating storage to simple variables and information vectors (see section 3.4) during translation. For simplicity the case of simple variables only will be considered. An easy modification will also provide storage assignment to formal parameters, which is equivalent to generating local variables to correspond to the parameters.

Three push-down lists are required: (1) the variable push-down list V, (2) the block tracing push-down list T, and (3) the procedure maximum serial number push-down R. In addition a list S of sentinels (Boolean values) indicate which blocks of a nesting have procedures declared in them. This list may be made part, in an implementation, of the list R

or of T. If m is the current depth of nesting of blocks, the lists T, R, and S will have length m.

The variable push-down list V is the search area for the conversion of identifiers to symbolic or relative addresses. The position of the variable within the list, if a single entry list is used, can denote its relative address in the target program. However, the list will contain gaps, and actual deletions are necessary, whenever a procedure declaration has been processed, since new lists associated with higher level blocks must be added after the area which once contained the lists associated with a procedure declaration. The area is available for re-use after the end of the block in which the procedure was declared. The gaps may be avoided and the bookkeeping simplified if a double entry push-down list is used, as we shall assume here. For each variable, the double entry in V consists of the identifier and the associated serial number. The entries in R then determine with which serial number each new list must begin.

Each new variable list (including of course also information vectors for arrays) is added to V, which is then a list of all variables to which reference may be made in that block. The end serial number is added to R. This entry is compared to each entry in R which corresponds to that of a block in the nesting in which a procedure is being declared. If it is larger it replaces the corresponding element of R.

When procedure appears, the sentinel in S corresponding to the present block is set to indicate that the block is one in which there is currently a procedure declaration being processed.

When the end of a block is reached (whether an ordinary block or the block in a procedure declaration), the part of the variable list declared within this block is removed from V, and the corresponding entries from R, S and T. In addition the sentinel of the containing block is reset to indicate that a procedure declaration is not being processed within this block.

We may consider the lists R, S, T, and V to be one-dimensional arrays, whose current lengths are respectively m, m, m, and T[m]. Then (considering simple variables only) the mechanism may be reduced to operations at five points:

(1) when a begin is encountered:

```
m := m+1; T[m] := T[m-1]; R[m] := R[m-1]; S[m] := false
```

(2) when procedure is encountered:

```
S[m] := true
```

(3) when the declaration of a variable is encountered:

```
R[m] := R[m]+1; T[m] := T[m]+1; V[T[m]] := (var, R[m])
```

(4) at the end of the declarations for each block:

```
for i := 1 step 1 until m do
begin if S[i] then R[i] := max(R[i], R[m]) end
```

(5) when end is encountered:

```
m := m - 1; S[m] := false
```

Initialization requires m := R[0] := T[0] := 0; S[0] := false at the beginning of the program.

3.3. Temporary Storage for Expression Evaluation

In the evaluation of expressions, temporary storage is needed for the preservation of intermediate results. The assignment of this

storage may be performed during the execution of the target program by means of a simple push-down list. While this technique may be necessary if one is handling recursive procedures, it is likely to seriously affect target program speed if employed when recursive procedures are not to be handled. Considering the nonrecursive case, the problem is made interesting by function procedures and the desire to minimize the storage required.

Assuming one does not assign a different temporary to each request for storage (which would require an inordinate amount of memory), the compiler will make the assignments on a push-down basis. Because of the appearance of function designators in expressions, the temporary storage level will not necessarily be zero upon dynamic entry to a procedure. The following techniques may be considered:

(1) Temporaries may be assigned with aid of the same mechanism used for declared variables, as described in section 3.2.

(2) The temporaries required by each procedure may be allocated space within the coding for the procedure itself.

(3) The following algorithm may be used: Let h be the pointer for push-down storage and hmax another counter. When

h := h + 1 ,

execute the statement

if h > hmax then hmax := hmax + 1 .

And at the end of a procedure declaration,

h := hmax .

Initially,

h := hmax := 0 .

These operations are all performed by the translator. While technique (1) will yield the most economical use of storage, experience indicates that (3) is a quite reasonable method.

3.4. Arrays

3.4.1. Allocation of Storage. The storage allocation problem is quite different for own and non-own arrays. We shall not present plans for the full treatment of own arrays (see [35]) but shall discuss the case where the own arrays have subscript bounds which are either constant or, if variable, do not change after having once been given a value.

3.4.2. Non-own Arrays. The storage for non-own arrays is naturally handled in push-down fashion in accordance with the block structure of the program. A push-down list of counters BL[1], BL[2], ... is generated during execution. The value of BL[m] is the initial address of available storage upon entry to a block contained dynamically within a nest of m-1 other blocks.

The subscripting operation, computing the address of an array element, requires certain information about the array which is available in the declaration. Suppose the array A is declared by

$$\underline{\text{array}}\ A[a_1:b_1\ ,\ a_2:b_2\ ,\ \ldots,\ a_N:b_N]\ .$$

The a_i and b_i can be any arithmetic expressions. If any of them have non-integer values, the evaluation of those expressions must be followed by the invocation of the appropriate transfer function. Let a_i' and b_i', $i = 1, 2, \ldots, N$, denote the resulting integer values. Define

$$K_i = b_i' - a_i' + 1\ , \qquad i = 1, 2, \ldots, N.$$

The information necessary to the subscripting operation is conveniently stored in an information vector. The information should include the address of the array origin A[0, 0, ..., 0] and the numbers K_2, K_3, ..., K_N or, alternatively, K_1, K_2, ..., K_{N-1}.

The computations of the K_i and the array origin, the storing of these values into the information vector, and the adjustment of the proper BL[m] counter are performed in the target program by coding provided at the beginning of the block. Let the function φ_A be defined by $\varphi_A(j_1, j_2, \ldots, j_N) = (\ldots(j_1 \cdot K_2 + j_2) \cdot K_3 + \ldots) \cdot K_N + j_N$. Suppose the above declaration of A is the first in the block. Then the first sequence of operations upon entry to the block during execution will be:

(1) Aux := BL[m].

(2) Compute a_i, b_i, K_i, i = 1, 2, ..., N.

(3) loc (A[0, 0, ..., 0]) := Aux - $\varphi_A(a_1', a_2', \ldots, a_N')$.

(4) $L := K_1 \times K_2 \times \ldots \times K_N$.

(5) Aux := Aux + L.

When all array declarations for this block have been processed (during execution),

(6) m := m + 1.

(7) BL[m] := Aux.

3.4.3. Information Vector. The storage allocation mechanism can provide N consecutive locations for the information vector, the address of the first of these being that assigned to the array identifier itself. The information vector then has the form:

$$\text{loc (A)} : \text{loc (A[0, 0, ..., 0])}$$

$$\text{loc (A)} + 1 : K_2$$

$$\vdots$$

$$\text{loc (A)} + N - 1 : K_N \ .$$

In the one-dimensional case no K_i's are required, which corresponds to the case of the switch.

3.4.4. Own Arrays. In the case of an own array, another area of memory is made available for storage, and a single counter is sufficient for keeping a record of free space. The computation of the elements of the information vector are the same as in the non-own case. One additional device is necessary. If the block containing the declaration has been previously entered, no storage allocation is done, and the information vector is not disturbed. Some cell should be set aside to contain a Boolean value indicating whether the block has been previously entered. It should be initialized to false at the beginning of the program and set to true upon the initial entry to the block. It is tested upon each entry to the block to determine whether to process the declaration.

3.4.5. Subscripting Operation. To compute the address of $A[j_1, j_2, \ldots, j_N]$, assuming the j_i's are integer-valued, the operation is

$$\text{loc}(A[j_1, j_2, \ldots, j_N]) := \text{loc}(A[0, 0, \ldots, 0]) + \varphi_A(j_1, j_2, \ldots, j_N) \ .$$

3.5. Switches

A switch is essentially a one-dimensional non-own array. The subscripting operation in computing a switch designator should be carried

out in the same manner as the computation of the address of a subscripted variable. This is indicated by the following example:

procedure P(t) ; Q(t[i]) .

Here t may refer to either an array identifier or a switch identifier.

Consider a switch declaration such as

switch S := L, if B then M else N, T[j + k] .

To treat this as an array, we consider it as having three elements. There is a difficulty, however, in that the coding for the designational expressions will in general occupy more than one computer word (in the example the last two expressions). To solve this problem, these pieces of coding will be referenced indirectly; that is, there will be an array of three computer words each of which contains a jump instruction either to a simple label contained in the switch declaration (under certain conditions) or to a generated label which marks the beginning of the appropriate piece of coding. The simple label of each such piece of coding representing a designational expression will produce the address to which a jump must be made and perform this jump (possibly through a GO TO interpreter, see section 3.8). The address which results from the computation of the switch designator S[E] is the address of one of the jump instructions in the three-word array. The execution of the statement

go to S[E] ,

in general involves a multiple jump. As in the case of an array, the switch identifier S is initialized so as to contain the address of the "origin", which here is the location just preceding the first element of the three-word array. Assuming this initialization has been carried out, the coding generated by the above switch declaration has the following

skeletal form:

jump to L_u

L_{u+1}: jump to L

L_{u+2}: evaluate B, jump to M if B is true, otherwise jump to N

L_{u+3}: compute the address T[j+k] and jump to that address.

L_{u+4}: contents unimportant

jump to L_{u+1}

jump to L_{u+2}

jump to L_{u+3}

L_u :

The address L_{u+4} is that to which S is initialized. It can be made the last word occupied by the coding marked L_{u+3}.

A degree of optimization is possible in this example if L is the label of a statement in the current block. In that case the instruction "jump to L_{u+1}" can be replaced by "jump to L", and the rest of the skeletal form altered accordingly.

To make the effect of a statement

go to S[i]

that of a dummy statement if i lies outside the declared range requires obvious additions to the scheme. This introduces considerable inefficiency and is of doubtful value.

3.6. Procedures

The Algol procedure is a generalization of the familiar notion of the closed subroutine. The features which present translation problems are: recursiveness, call by name and call by value, and a wide variety of possible parameters. The problems of recursiveness and call of arrays by value are not discussed here, but the approach to implementation given below can be modified to include the complete procedure concept. See [23,24].

Included among the possibilities for parameters is that of the procedure identifier. This fact is of considerable significance in that it shows that the translator should in general make no reference to the procedure declaration in translating a procedure call. Consider the following example:

```
procedure P(a); integer a;
begin --- end;
procedure Q(b); value b; integer b;
begin --- end;
procedure R(c); procedure c;
begin integer K;
        ⋮
      c(K);
        ⋮
end.
```

It is the call c(K) that points up the difficulty. If R is called as R(P), then c(K) becomes P(K); if R is called as R(Q), then c(K) becomes

Q(K). Clearly the link from the body of R, at c(K), must be deferred until run time. Also, the translator cannot make any assumptions about whether K is called by name or by value. In this example, P will call K by name and Q will call it by value.

The call by name of expressions and subscripted variables presents the problem of repeated reference from the procedure body to the calling sequence. (See the Innerproduct procedure in section 5.4.2 of the Algol 60 report [29,30].) To accommodate this, the calling sequence can represent each parameter as a closed subroutine. This will henceforth be referred to as the parameter subroutine. The function of this subroutine will be to leave the appropriate address in some predetermined register.

As mentioned above, a parameter subroutine produces an address. The interpretation of this address is left to the calling subroutine, that is, the procedure body. In some cases a store is done into this address, in other cases the value in this address is brought into a register, or a transfer made to this address, etc. The next paragraph will describe how this linkage between procedure body and procedure statement is effected. First we shall consider the various types of actual parameters and see what sort of addresses must be produced by the parameter subroutines. We classify parameters as follows:

Class 1: simple variables, constants, labels, formal parameters called by value - the subroutine simply puts into some register the address of the variable, or constant, or label, or generated local identifier corresponding to the formal parameter;

Class 2: subscripted variables, switch designators - the parameter subroutine computes the address of the array element or the address to which transfer is to be made and leaves it in the register;

Class 3: array identifiers, switch identifiers - the subroutine places in the register the address of the table (information vector) where the information for the subscripting is to be found;

Class 4: procedure identifiers - the subroutine produces the address at which the coding for the procedure begins (the exception is where the procedure identifier is a function designator having no parameters; here the subroutine must carry out a subroutine-jump to the procedure);

Class 5: formal parameters called by name - the subroutine for such a parameter does a subroutine-jump to the parameter subroutine corresponding to the formal parameter (this kind of thing can proceed to a depth of several levels);

Class 6: expressions - the parameter subroutine evaluates the expression, stores the value in a temporary location, and produces the address of the temporary;

Class 7: strings - the subroutine produces the address at which the string begins.

The mechanism for linking a reference to a formal parameter to the appropriate parameter subroutine will to a high degree depend upon machine characteristics. We give below a brief sketch of an approach to the problem.

In addition to the parameter subroutines, the calling sequence generated for a procedure call will also have a list of pointers, each occupying one word. Each pointer contains the address of a parameter subroutine. To accommodate these pointers, the translator sets aside a "formal location" for each formal parameter when the procedure declaration is translated. At run time, then, the execution of a procedure call will involve the following steps:

1. A subroutine jump is made to the coding for the procedure declaration. The starting address of the pointers is somehow made available to the procedure declaration, perhaps through an index register.
2. The procedure declaration picks up each pointer and places it in the appropriate formal location.
3. Now when any reference is made to a formal parameter, the corresponding formal location contains the address of the necessary parameter subroutine. For a parameter called by value, this parameter subroutine is activated before the procedure body is entered; the value in the delivered address is assigned to a generated local identifier corresponding to the formal parameter, and all references to this parameter in the procedure body are treated as references to the generated identifier.

On many machines it would be convenient to have a subroutine jump code in the pointer as well as the address. It is clear that an execute instruction might be useful for activating the parameter subroutine through the pointer.

We have so far ignored the matter of specifications as well as the question of types. The correct matching of arithmetic types when one operand is a formal parameter can only be done through dynamic type handling, that is, in the running target program. A fairly satisfactory restriction, however, might be to have the translator assume a formal parameter which acts as an arithmetic quantity is of type real unless specified integer. The programmer must then keep the types of corresponding actual and formal parameters identical. As far as other specifications are concerned, the use of some of them, particularly array, might enable the translator to more easily achieve some optimization.

It should also be observed that the procedure linkage mechanism makes such calls as F(F(x)) recursive even if F calls its parameter by value.

3.7. The For Statement

Since the for statement is essentially a shorthand device for a complex of other Algol statements, it is a natural candidate for "boot strapping". As is described in these plans, it is simple to build the set of Algol statements which is equivalent to the for statement and process these through the other parts of the translator. It is desirable to modify this notion slightly in the case of a for list with more than one element. There the statement following the for clause should be made a closed subroutine to avoid requiring more than one copy of it. If object code efficiency may be ignored, the translation is simpler if this is done in every case.

The problem of optimization of loops can be attacked at several levels of difficulty. Because of this and the rather large extent to which hardware must be taken into account here, specific plans are not easily presented. We shall indicate a few of the possibilities.

The process in which new Algol statements are generated from the for statement and processed intact through the translator may generally be modified to yield improvement in the object code. For example, the multiplication in the if clause

$$\underline{\text{if}}\ (V - C) \times \text{sign}(B) > 0\ \underline{\text{then}}\ ,$$

constructed from a list element of the step - until type, is generally replaced by a faster logical operation to compare the signs of V - C and B.

The selection of certain forms of list elements for special processing that takes advantage of convenient machine instructions, will greatly improve object code efficiency. In particular, list elements of the type

$$x\ \underline{\text{step}}\ 1\ \underline{\text{until}}\ y$$

occur very frequently and deserve special treatment.

Finally, recursive address calculation using index registers [22,34] can greatly improve target program reference of arrays.

3.8. The GO TO Interpreter

On entry to a block at execution time, the block nesting index m is raised by one, storage is allocated to arrays declared in that block, and the beginning of free storage for any subordinate block is stored as the value of BL[m] (sec. 3.4.2). Except for procedures, the index m could be handled statically, that is, assigned on entry to the block

a definite value determined already at translation time. Procedure bodies, however, cannot be assigned a fixed number in advance, so it is preferable to handle this counter dynamically (i.e. as m := m + 1) at the beginning of each block. If this is done, a corresponding inverse operation must be performed on exit from the block. This involves among other things, therefore, bookkeeping in connection with any go to statement leading out of the block. To solve this problem it is sufficient to keep in the target program a push-down list of pairs of addressess, the last pair always being the beginning and ending addresses of the code for the innermost block currently being executed. When a go to statement is encountered, the address to which transfer is to be made is given to the GO TO interpreter, a routine which the translator places in the target program. This interpreter compares the address given it with the most recent address pair in the push-down list. If the given address lies within the limits, a simple jump is performed; otherwise the block nesting index is decreased by one and new block limits become effective.

Each pair of block limits is of course entered into the push-down list by coding which appears at the beginning of the block.

This description assumes that the coding for each block is stored sequentially in memory. For further comments on this and for an extension of the notion to include recursive procedures, see [26].

If the compiler is to have the facility for translating procedures separately from a calling program, provision must be made for linking the block limits push-down list in the calling program to that in the procedure.

3.9. Strings

In a strict interpretation of Algol 60, strings as such can only be used as actual parameters of procedures with formal parameters specified string. Therefore, strings may be treated much like constants. There need be no limit on their length. It may be desirable, anticipating the appearance of string variables in a future version of Algol, to refer to them indirectly, i.e., the address associated with the string points to the location where the storage of the string begins. As strings are manipulated by procedures written in non-Algol language, the programmer must know how the strings are stored and how they are delimited. While nested strings may be of limited value, no significantly additional effort is required to provide for them.

4. Specifications for a Translator

4.1. General Considerations

The translator, for which specifications are given below, has as source language process Algol. This means that a program is required to convert the hardware Algol of a given computer into process Algol before the language can be handled by the translator. Such a program depends on the actual hardware language used and may easily be designed. It need, consequently, not be discussed here.

Based on the theory outlined in chapter 2, the translator is organized as a collection of recursive subroutines based on the syntactic skeletons. The problem of designing the translator, therefore, becomes that of designing the subroutines. To promote translation efficiency, the organization of the subroutines is around the framework of two

push-down lists for the storage of intermediate information. The basic switching mechanism may be described by means of a table or matrix rather than a flow-chart. The various subprograms can be written in Algol itself, augmented by additional primitive elements; to obtain a working translator, these may easily be hand-translated into programs in a symbolic machine language, and a suitable device may be used to implement the switch [18].

The recursive subroutines representing the syntactic units call other subroutines, a few dozen in number. To avoid confusion in the terminology, the term subroutine is retained for the former and the term macro used for the latter. Macros are of three main types: (1) those that manipulate the control push-down and determine the sequence of operations performed by the translator, (2) those that generate target program, and (3) those that perform necessary bookkeeping, storage, and checking.

To permit easy reference to certain lists which play a leading role during translation, we introduce the following notation for them and their elements:

List	Name	Nature of List Element	Item	Counter
Source program	Γ	Symbol of process Algol	γ	g
Target program	$\prod$	Symbolic or machine instruction	π	p
Temporary push-down	H	Generated variable	η	h
Control push-down	$\sum$	State	σ	s
Auxiliary push-down	A	Miscellaneous information	α	a

The function of the translator is to produce from the source program Γ consisting of the elements $\gamma_1, \gamma_2, \ldots, \gamma_G$ the target program

$\prod$ with symbolic or machine instructions $\pi_1, \pi_2, \ldots, \pi_P$. Apart from the list H which furnishes generated variables needed in the target program, the remaining lists have a catalytic function in the process. For example, the control push-down at a given point during translation will contain elements $\sigma_1, \sigma_2, \ldots, \sigma_s$, where s is the current size of the list at this point. Initially and finally (in a syntactically correct program) the list is empty, except for an initial state.

The process to be described is sequential. This means that the list Γ is scanned only once from first element to last element during translation. At each point in translation, the last element σ_s of the control push-down $\sum_i$ and the p-Algol symbol γ_g under scan together determine a list of macros that operate on the lists $\sum_i$ and Γ. Thus, the control push-down is changed at times. The operations on Γ by the requirement of sequentiality are limited to retaining the current symbol under scan or proceeding to the next one (by the operation $g := g + 1$).

One way of describing the process is by a matrix whose columns are headed by all possible states which may occur as σ_s and whose rows are headed by all possible p-Algol symbols which occur as γ_g. In the field determined by each state and symbol (σ_s, γ_g) may be listed the macros to be executed when that combination occurs. This device was first used by the ALCOR group [3] [31]; it was retained in [19].

However, a serious attempt has been made to minimize the total number of entries in the present design by the suitable choice of states and macros. In consequence most of the fields in the matrix are blank; if the corresponding pairs do occur during translation, an error

in syntax is indicated. A simpler course, therefore, is to list in a table all valid combinations and the actions evoked by them. If in implementation, it is desired to place in memory the actual matrix first described in order to do the switching, the reconstruction may easily be made from the table.

The translator program is basically the following:

begin

Initialize; comment This procedure carries out all operations necessary to begin the process. This includes the initialization of the counters in the above table and the placing of the state PR into $\sum$.

next: g := g + 1;

process: Execute (σ[s], γ[g]); comment Execute is a procedure that executes in turn each of the macros listed in the table opposite the pair (σ_s, γ_g);

end

In addition to the above lists, two label tables, LABDEC and LABREQ, are used if the target language is machine language (their functions are generally delegated to an assembler if symbolic language is the output from the translator). The LABDEC table is a double entry table, with the first element a label name and the second element the address to which go to statements leading to this label must cause transfer. The LABREQ table is also double entry, with its first element a label name and its second element an address at which must be supplied the address associated with that label in the LABDEC table. At the time an entry is made into the LABREQ table, the address required in that entry may not be available due to a forward reference in the program. Of course no entry need be made in LABREQ if the address is available.

There are two sequences of generated labels: Lu, u = 1, 2, ... and Mq, q = 1, 2, ...

4.2. Control Operations

There are three macros which perform control operations. These add to, delete from, or replace the last entry of the control push-down. Letting λ denote a state, the operations are described by means of Algol statements as follows:

1. Ent(λ) - enter a recursive subroutine:

   s := s + 1; σ[s] := λ

   In the translation table, the activation of this macro is indicated by the appearance of a state in the Add column. It is also called from other macros.

2. Ch(λ) - establish a new state within a subroutine:

   σ[s] := λ

   In the translation table, the appearance of a state in the Change column denotes the activation of this macro.

3. Exit - exit from a recursive subroutine:

   s := s - 1

   This is indicated in the translation table by the appearance of an asterisk in the Delete column. Exit is also called from other macros.

4.3. Control States

1. Assignment statement

   A1 Added to control push-down upon encountering assignment symbol :=, assumes control when expression on right-hand side has been processed.

A2 Added to control push-down from state A1 when multiple assignment is discovered, assumes control when expression on right-hand side has been processed.

2. Conditional statement

C1 Added to control push-down when if is encountered in statement state, assumes control when Boolean expression following if has been processed.

C2 Added to control push-down when then is encountered in state C1, assumes control when unconditional statement following then has been processed.

C3 Added to control push-down when else is encountered in state C2, assumes control when statement following else has been processed.

3. Conditional arithmetic or Boolean expression

CE1 Added to control push-down when if is encountered in state E0, assumes control when Boolean expression following if has been processed.

CE2 Added to control push-down when then is encountered in state CE1, assumes control when expression following then has been processed.

CE3 Added to control push-down when else is encountered in state CE2, assumes control when expression following else has been processed.

4. Declaration

D Added to control push-down when declarator is encountered in state S0, and again if own is encountered in state D. It assumes control after each declaration is processed.

D1 Given control from state D after type declarator.

D2 Given control from state D1 after element of type list.

5. Array declaration

Da — Given control to process array identifier of array declaration.

Da1 — Given control from state Da. The left bracket of an array segment or a comma is expected.

Da2 — Assumes control when lower bound has been processed.

Da3 — Assumes control when upper bound has been processed, added to control push-down in state Da2.

Da4 — Given control from state Da3 following right bracket of array segment.

6. Procedure declaration

Dp — Given control from state D when procedure declarator is encountered.

Dp0 — Added to control push-down from state Dp, assumes control when the procedure declaration has been processed and a semicolon is expected.

Dp1 — Given control from state Dp to determine whether formal parameter part is empty.

Dp2 — Given control to process formal parameter.

Dp3 — Given control from state Dp2 to discriminate between right parenthesis and comma following formal parameter.

Dp4 — Given control from state Dp3 to process semicolon.

Dp5 — Given control from state Dp4 to determine whether value part exists.

Dp6 — Given control to process identifier of value part.

Dp7 — Discriminates between comma and semicolon following identifier of value part.

Dp8 — Determines whether specification part exists.

Dp9 Given control after type specifier is encountered in state Dp8.

Dp10 Discriminates between comma and semicolon following identifier in specification part.

Dp11 Given control to process identifier of specification part.

7. Switch declaration

Ds Given control to process identifier of switch declaration.

Ds1 Given control to process assignment symbol of switch declaration.

Ds2 Replaces state Ds1 in control push-down, uncovered when designational expression has been processed, discriminates between comma and semicolon.

8. Arithmetic or Boolean expression

E0 Whenever an arithmetic or Boolean expression is expected, this state is given control.

E1 Added to the control push-down in state E0. When state E1 assumes control, an operand has been processed, and its address directly or indirectly stored in the uppermost cell of the auxiliary push-down.

E2 Added to control push-down in state E1, together with the binary operation encountered there. When state E2 assumes control, an operand has been processed. It is in this state and E3 that precedence of operations is handled.

E3 Added to control push-down in state E0 when a unary operation is encountered. The unary operation is stored in the control push-down in the same cell as the state E3. This state assumes control when an operand has been processed.

UE0 Whenever an unconditional arithmetic or Boolean expression is expected, this state is given control from CE1.

9. For statement

F0 Given control when for is encountered. The controlled variable is copied into table V while in this state.

F1 Given control to copy the expression following the assignment symbol of for clause.

F2 Given control to copy the increment expression in step-until for list element.

F3 Given control to copy terminal expression in step-until for list element.

F4 Given control to copy Boolean expression in while for list element.

F5 Added to control push-down when do is encountered, assumes control to process end-of-statement indicator following for statement.

F6 Given control while copying expression in for clause when left parenthesis is encountered.

F7 Given control while copying expression in for clause when left bracket is encountered.

10. Go to statement and designational expression

G Added to control push-down when go to is encountered, assumes control to process end-of-statement indicator following go to statement.

G1 Given control to process designational expression of go to statement.

G2 Given control to determine whether preceding symbol was label or switch identifier.

G3 Added to control push-down when switch designator is determined, assumes control to process right bracket.

G4 Given control in state CG to process the unconditional designational experession following then.

G5 Added to control push-down when then is encountered in state CG, assumes control to process else of conditional designational expression.

G6 Added to control push-down in state G5, assumes control at end of designational expression.

CG Added to control push-down in state G1 when if is encountered, assumes control to process then of conditional designational expression.

11. Identifier, operand and parenthesis level

I1 Given control to distinguish subscripted variables and procedure calls from simple variables and to provide for the processing of procedures without parameters.

I2 Added to control push-down in state I1 upon discovery of subscripted variable, assumes control to distinguish between comma and right bracket following subscript expression.

I3 Added to control push-down in state I1 upon discovery of procedure call, assumes control to distinguish between comma and right parenthesis following actual parameter.

I4 Given control when an identifier is the first symbol in a statement, provides for adjustment of control push-down in case of label.

I5 Same as I4 for unconditional statement.

O When an operand is expected, this state is given control.

P Added to control push-down when left parenthesis is encountered in state O, assumes control to process corresponding right parenthesis.

12. Statement

PR Initial control state when processing begins, checks that program starts with begin.

S0 Given control when begin is encountered, determines whether a statement is a block.

S1 Added to control push-down when first declaration of block is found, provides for processing of end at end of block.

S2 Given control when statement is expected.

S3 Added to control push-down in states S2 and US2, assumes control to process end-of-statement indicator.

US2 Given control when unconditional statement is expected.

4.4. Translation Table

The translation table is given in the following pages. In the Symbol column is a list of all p-Algol symbols permissible in the indicated state. If the entry "other" occurs in the Symbol column, any symbol not listed explicitly for that state should cause the action that would be indicated if that symbol stood in place of "other". The actions associated with each state-symbol pair are given in that line of the table in which the symbol occurs. These actions are executed from left to right beginning with any macros in the Building Block column. The columns under the heading "Control Push-down" indicate changes in the $\sum$ push-down in accordance with section 4.2 on control operations. An asterisk in the Delete column indicates that the Exit operation should be performed. The entry in the Transfer column is the label in the translation program to which transfer is made after execution of all operations in that line of the table.

It is clear that a more extensive use of the "other" device can be made to reduce the number of table entries if a syntactical check is not desired. The table is derived from the metasyntactical formulas

appearing in the Algol Report. As has been pointed out [8], these formulas describe a "superlanguage" in which Algol 60 is imbedded. The table is intended to permit a complete syntactical check of this superlanguage except in the case of the for statement. The expressions in the for clause and the controlled variable may of course be checked while processing the strings constructed from the for statement. This may make it difficult, however, to pinpoint the error. Alternatively, a prepass may perform the check, and the modifications in states F0 through F7 are fairly evident: states F6 and F7 (needed in translation because of the problem of commas in the copying operation) are eliminated, state E0 is entered where an expression is expected, and a state is added to the table for checking the controlled variable (which may be subscripted). Still another possibility, one not requiring a prepass, is to provide special expression states which handle the copying operation. A complete syntactical check of Algol 60 programs must also determine whether the declarations are complete and consistent and whether identifiers are used in accordance with their declarations.

As indicated earlier in this section, the translator has as its source language process Algol. Consequently, it is assumed that the information contained in the declarations has previously been collected. The actions indicated in the table with regard to declarations are therefore restricted to those actually involving the generation of code.

The symbol ω is used in expression states to indicate a unary or binary operation. The letter I in the Symbol column denotes an identifier.

The form of the translation table here is somewhat different from that given in previous papers by one of the present writers [18,19]. The principles are, however, identical.

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
A1	end	EV1	*			process
	else	EV1	*			process
	;	EV1	*			process
	:=			A2	A1, E0	next
A2	end	EV2	*			process
	else	EV2	*			process
	;	EV2	*			process
C1	then	IF		C2	US2	next
C2	end	THEN	*			process
	else	ELSE		C3	S2	next
	;	THEN	*			process
C3	end	THEN	*			process
	;	THEN	*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
CE1	then	IF		CE2	UE0	next
CE2	else	CC,ELSE		CE3	E0	next
CE3	then	CC,THEN,CC1	*			process
	step	CC,THEN,CC1	*			process
	while	CC,THEN,CC1	*			process
	until	CC,THEN,CC1	*			process
	do	CC,THEN,CC1	*			process
	,	CC,THEN,CC1	*			process
	]	CC,THEN,CC1	*			process
	)	CC,THEN,CC1	*			process
	end	CC,THEN,CC1	*			process
	else	CC,THEN,CC1	*			process
	;	CC,THEN,CC1	*			process
D	real				D1	next
	integer				D1	next
	Boolean				D1	next
	array	ARRAY			Da	next
	switch				Ds	next
	procedure				Dp	next
	own			D		next
	other		*			process
D1	array	ARRAY		Da		next
	procedure			Dp		next
	I			D2		next
D2	,			D1		next
	;		*			next

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
Da	I	STID		Da1		next
Da1	[			Da2	EO	next
	,			Da		next
Da2	:	STV		Da3	EO	next
Da3	,	VECTOR		Da2	EO	next
	]	VECTOR,ORIGIN		Da4		next
Da4	,	ARRAY		Da		next
	;		*			next
Dp	I	PROCDEC		Dp0	Dp1	next
DP0	;	ENDPROC	*			next
Dp1	(			Dp2		next
	;			S2		next
Dp2	I	STORE PAR		Dp3		next
Dp3	,			Dp2		next
	)			Dp4		next
Dp4	;			Dp5		next
Dp5	value			Dp6		next
	other			Dp8		process
Dp6	I	VALUE		Dp7		next
Dp7	,			Dp6		next
	;			Dp8		next

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
Dp8	real				Dp9	next
	integer				Dp9	next
	Boolean				Dp9	next
	array				Dp11	next
	switch				Dp11	next
	procedure				Dp11	next
	string				Dp11	next
	label				Dp11	next
	other			S2		process
Dp9	I			Dp10		next
	array			Dp11		next
	procedure			Dp11		next
Dp10	,			Dp11		next
	;		*			next
Dp11	I			Dp10		next
Ds	I	STID,SWITCH		Ds1		next
Ds1	:=			Ds2	G1	next
Ds2	,	TRA,RAILROAD			G1	next
	;	TRA,JUMPLIST	*			next

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
E0	if			CE1	E0	next
	I			E1	0	process
	(			E1	0	process
	ω			E1	(E3,ω),0	next
E1	then		*			process
	step		*			process
	while		*			process
	until		*			process
	do		*			process
	,		*			process
	]		*			process
	)		*			process
	end		*			process
	else		*			process
	;		*			process
	:		*			process
	ω				(E2,ω),0	next
(E2,ω)	then	EXB	*			process
	step	EXB	*			process
	while	EXB	*			process
	until	EXB	*			process
	do	EXB	*			process
	,	EXB	*			process
	]	EXB	*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
(E2,ω) (cont.)	)	EXB	*			process
	end	EXB	*			process
	else	EXB	*			process
	;	EXB	*			process
	:	EXB	*			process
	ω	COMPEX				
(E3,ω)	then	EXU	*			process
	step	EXU	*			process
	while	EXU	*			process
	until	EXU	*			process
	do	EXU	*			process
	,	EXU	*			process
	]	EXU	*			process
	)	EXU	*			process
	end	EXU	*			process
	else	EXU	*			process
	;	EXU	*			process
	:	EXU	*			process
	ω	COMPUX				
UEO	I			E1	0	process
	(			E1	0	process
	ω			E1	(E3,ω),0	next

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
F0	:=			F1		next
	other	COPY(0)		F0		next
F1	step			F2		next
	while			F4		next
	,	A1,CL		F1		next
	do	A1,B		F5	S2	next
	[	COPY(1),list:=1			F7	next
	(	COPY(1),list:=1			F6	next
	other	COPY(1)		F1		next
F2	until			F3		next
	other	COPY(2)		F2		next
F3	,	A2,CL		F1		next
	do	A3,B		F5	S2	next
	[	COPY(3),list:=3			F7	next
	(	COPY(3),list:=3			F6	next
	other	COPY(3)		F3		next
F4	,	A3,CL		F1		next
	do	A3,B		F5	S2	next
	[	COPY(2),list:=2			F7	next
	(	COPY(2),list:=2			F6	next
	other	COPY(2)		F4		next
F5	;	C	*			process
	else	C	*			process
	end	C	*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
F6	(	COPY(list)			F6	next
	[	COPY(list)			F7	next
	)	COPY(list)	*			next
	other	COPY(list)				next
F7	[	COPY(list)			F7	next
	]	COPY(list)	*			next
	(	COPY(list)			F6	next
	other	COPY(list)				next
G	end	TRA	*			process
	else	TRA	*			process
	;	TRA	*			process
G1	if			CG	E0	next
	I			G2		next
	(			P	G1	next
G2	[			G3	E0	next
	;		*			process
	end		*			process
	else		*			process
	)		*			process
	,		*			process
G3	]		*			next
G4	I			G2		next
	(			P	G1	next

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
G5	else			G6	G1	next
G6	,		*			process
	;		*			process
	end		*			process
	else		*			process
	)		*			process
CG	then			G5	G4	next
I1	(	PROC		I3	E0	next
	[			I2	E0	next
	then	NOPAR	*			process
	step	NOPAR	*			process
	while	NOPAR	*			process
	until	NOPAR	*			process
	do	NOPAR	*			process
	,	NOPAR	*			process
	]	NOPAR	*			process
	)	NOPAR	*			process
	end	NOPAR	*			process
	else	NOPAR	*			process
	;	NOPAR	*			process
	ω	NOPAR	*			process
	:	NOPAR	*			process
	:=		*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
I2	,	STV			EO	next
	]	SUBS	*			next
I3	,	RETURN			EO	next
	)	RETURN,FUNC	*			next
I4	:		*	S2		next
	(			I1		process
	[				I1	process
	:=			A1	EO	next
	end		*			process
	else		*			process
	;		*			process
I5	:		*	US2		next
	(			I1		process
	[				I1	process
	:=			A1	EO	next
	end		*			process
	else		*			process
	;		*			process
0	I	STID		I1		next
	(			P	EO	next
	ω			(E3,ω)	0	process
P	)		*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
PR	begin			S0		process
S0	begin				S0	next
	for				S2	process
	goto				S2	process
	if				S2	process
	I				S2	process
	end	EOP	*			next
	;				S2	next
	declarator	BBL		S1	D	process
S1	begin				S2	process
	for				S2	process
	goto				S2	process
	if				S2	process
	I				S2	process
	end	EOB	*			next
	;				S2	next
S2	begin			S3	S0	next
	for	CLO		S3	F0	next
	goto			S3	G,G1	next
	if			S3	C1,E0	next
	I	STID		S3	I4	next
	end		*			process
	;		*			process

State	Symbol	Building Block	Control Push-down			Transfer
			Delete	Change	Add	
S3	;		*			process
	end		*			process
	else		*			process
US2	begin			S3	S0	next
	for			S3	F0	next
	goto			S3	G,G1	next
	I	STID		S3	I5	next
	end		*			process
	else		*			process
	;		*			process

4.5. Target Language

In order to describe the macros, the machine code of a fictitious single-address machine will be assumed as the target language. Only certain instructions are used explicitly:

CLA y Clear the accumulator and add the contents of location y.

STO y Store the contents of the accumulator in location y.

ENA t Clear the accumulator and enter the number t. This is equivalent to CLA y where t is in location y.

INA t Add the number t to the contents of the accumulator.

SUB y Subtract (fixed-point) the contents of location y from the accumulator.

TRA y Transfer control to the instruction in location y.

TIF y Transfer if value in accumulator is false to instruction in location y; otherwise proceed.

SJP y Perform subroutine jump to location y.

SSE y Set the subroutine exit address in the instruction in location y.

MPY y Multiply (fixed-point) the contents of the accumulator by the contents of y, leaving the result in the accumulator.

In several macros the operation codes SJP and SSE appear underlined. This occurs where strings of p-Algol symbols are built up in processing the for statement and means that the translator is to generate an instruction using the underlined operation code.

4.6. Notation in Macros

The macros are described in Algol, with the exception of some additional notational conveniences. The comment facility of Algol is used in instances where the details are obvious or highly machine-dependent. To augment Algol the following devices are used:

C(E) The target program address which is the value of E. E is a translator variable or itself a target program address (the case of indirect addressing).

< m > The representation in the translator of the target program address m. The assignment of this representation to a translator variable v is indicated by v := < m >.

Ac The representation in the translator of the accumulator, indicating its use in the target program as a temporary.

An apostrophe is used to indicate both left and right string quotes.

4.7. Macros

The macros appearing in the translation table are described below. With the exception of those in section I, they are grouped under headings corresponding to the subroutines in which they are used.

I. General procedures. These procedures are used by many of those that follow. Some are involved in target program generation and the others in bookkeeping.

A. Target program generation

1. procedure TARGET (instruction); string instruction;
comment Writes symbolic or machine instruction into the target program ∏ , increasing counter p as necessary;

2. procedure LOADA; comment Writes an instruction to load the accumulator;
begin
if α[a] = < η[h] > then h := h - 1;
if α[a] is sentinelled then
TARGET (' CLA C(C(α[a]))') else
TARGET (' CLA C(α[a])'); α[a] := Ac
end;

3. procedure ROUND; comment Writes subroutine jump to target program routine when real-to-integer transfer function must be invoked;
if α[a] is of type real then
TARGET ('SJP FIX');

B. Bookkeeping

1. procedure LABEL DECLARE (label); label label; comment If the target language is machine code, an entry of label together with the address at which it is used is made in the LABDEC table (any requests for label in LABREQ can now be filled). If the target language is assembly language, label is placed in the target program;
2. procedure LABEL REQUEST (label); label label; comment If the target language is machine code, a scan is made of LABDEC: if label is there the address is filled, if not, label is entered with the associated address into LABREQ. If the target language is assembly language, no action is taken;
3. integer procedure prec (operation);
   comment Each operation has a precedence number, those of higher precedence having higher numbers, those of equal precedence having equal numbers. The value of this function is this number;

II. Declarations

A. Array declarations

1. procedure ARRAY; comment Initializes computation of array storage;

```
begin
    TARGET ('ENA 1');
    TARGET ('STO ARRAYLENGTH')
end;
```

2. procedure VECTOR; comment Computes and stores away the multipliers for subscription;

```
begin
    LOADA; ROUND; a := a - 1; if α[a] = < η[h] >
    then h := h - 1;
    if α[a] is sentinelled then
    TARGET ('SUB C(C(α[a]))') else
    TARGET ('SUB C(α[a])');
    TARGET ('INA 1');
    comment The value now computed is stored into the
    information vector of each array identifier in the
    declaration; a := a - 1;
    TARGET ('MPY ARRAYLENGTH');
    TARGET ('STO ARRAYLENGTH')
end;
```

3. procedure ORIGIN; comment Writes code to compute the origin addresses for the arrays in the declaration and store them in the information vectors, removes array identifiers from auxiliary push-down;

B. Switch declarations

1. procedure SWITCH; comment Writes code to initialize a switch. This consists of storing the address of the switch origin into the location reserved for the switch identifier;
2. procedure RAILROAD; comment Inserts a label at beginning of coding of designational expression in switch list;

```
begin
    u := u + 1; a := a + 1; α[a] := < Lu > ;
    LABEL DECLARE (Lu)
end;
```

3. procedure JUMPLIST; comment Writes the array of jump instructions following the switch origin;

C. Procedure declarations

1. procedure PROCDEC; comment Writes entry line of procedure. A label is generated and associated with the procedure identifier, and LABEL DECLARE is called to handle the label;

2. procedure ENDPROC; comment Writes exit line of procedure. If any special locations are reserved for procedure linkage, this routine may allot the space;

3. procedure STORE PAR; comment Processes formal parameter. Any coding required to establish linkage with the actual parameter in the calling sequence may be written at this point;

4. procedure VALUE; comment Writes code to fetch value of actual parameter and assign it to the formal parameter;

III. Compound statement and block

1. procedure BBL; comment Writes code to carry out block entry operations. This includes adjustment of array storage push-down pointer and block limits push-down pointer;

2. procedure EOB; comment Writes code to carry out block exit operations. This includes adjustment of array storage push-down pointer and block limits push-down pointer. Calls EOP;

3. procedure EOP; comment Tests for end of program;

IV. Designational

1. procedure LABEL; comment Places label in target program or LABDEC table;

```
begin
    LABEL DECLARE (C(α[a]));
    a := a - 1
end;
```

2. procedure TRA; comment Writes code for transfer;

```
begin
    if α[a] is a label then
    begin
        TARGET ('ENA C(α[a])');
        LABEL REQUEST (C(α[a]))
    end else LOADA;
    TARGET ('TRA GO TO'); comment GO TO is the target
    program routine referred to previously as the GO TO
    interpreter and discussed in section 3.8;
    a := a - 1
end;
```

V. Assignment

1. procedure EV1; comment Writes code to perform the operation of assignment;

```
begin LOADA; a := a - 1; EV2 end;
```

2. `procedure EV2; comment` Writes code to perform a storing operation;

```
begin
    if α[a] is sentinelled then
    begin
        TARGET ('STO C(C(α[a]))');
        h := h - 1
    end else TARGET ('STO C(α[a])'); a := a - 1
end;
```

VI. Operand and identifier

1. `procedure` STID; `comment` if $\gamma[g]$ is a formal parameter called by name, this routine writes code to fetch address from calling sequence and store it into temporary location, puts temporary into next available position in auxiliary push-down, and sentinels it. Otherwise, it puts $\gamma[g]$ into next available position in auxiliary push-down;

2. `procedure` SUBS; `comment` Writes code to compute the address of a subscripted variable or switch designator. If K is the number of subscripts, the last K entries in the auxiliary push-down give the addresses of the K subscript values. Any of these which are temporaries are released. $\alpha[a - K]$ contains the address at which the information vector begins. The address computed here is stored in the next available temporary. The address of this temporary is sentinelled and placed in $\alpha[a - K]$.

3. procedure STV; comment Writes code to do any necessary rounding and store value into next available temporary;

```
begin
    if α[a] is of type real then
    begin LOADA; ROUND end;
    if α[a] = Ac then
    begin
        h := h + 1; TARGET ('STO η[h]')
        α[a] = < η[h] >
    end
end;
```

4. procedure PROC; comment Writes code to initiate procedure call. This includes the subroutine jump to the appropriate procedure body, the address of which may not be known at translation time (the case of a formal procedure identifier) and any instructions necessary to establish linkage with the calling sequence;

5. procedure NOPAR; comment If α[a] is a procedure identifier having no parameters, a subroutine jump is written to the procedure;

6. procedure RETURN; comment Writes code to place the proper address in some standard location and exit from a parameter subroutine. This will sometimes require code to store a computed result in a temporary location;

7. procedure FUNC; comment Writes any code or link words necessary to describe to the procedure the locations of the parameter subroutines;

VII. Expression

1. procedure COMPEX; comment Tests for precedence between the incoming binary operation and the one 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; Exit; go to next end else

   begin Ent(E2, γ[g]); Ent(0); go to next end;

2. procedure COMPUX; comment Tests for precedence between the incoming binary operation and the unary operation 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 EXU; Exit; go to next end else

   begin Ent(E2, γ[g]); Ent(0); go to next end;

3. procedure EXU; comment Writes code for the execution of a unary operation;

   begin

   if α[a] ≠ < η[h] > then h := h + 1;

   comment now code is written to perform the assignment

   η[h] := ω C(α[a]);

   α[a] := < η[h] >

   end;

4. procedure EXB; comment Writes code for the execution of a binary operation;

```
begin
    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;
    comment now code is written to perform the assignment
      η[h] := C(α[a - 1]) ω C(α[a]).
    This typically might involve the testing of ω to
    determine which binary operation it is and executing
    a subroutine which performs that particular operation.
    On many computers it will be necessary to test the types
    of the operands and possibly invoke a transfer function.
    Type information should be preserved in the auxiliary
    push-down entries;
    a := a - 1; α[a] := < η[h] >
end;
```

VIII. For statement

1. procedure COPY (list); comment Copies $\gamma[g]$ into the next available position in a table, which is one of the tables V, EX1, EX2 or EX3 according as list equals 0, 1, 2 or 3;

2. procedure CLO; comment Initializes pointers for tables V, EX1, EX2 and EX3 and also q := q + 1;

3. procedure CL; comment Initializes pointers for tables EX1, EX2 and EX3;

4. procedure A1; comment Processes a list element of the type EX1. The following string is constructed and processed as if part of the Algol program;

```
'V := EX1; SJP Mq;';
```

5. procedure A2; comment Processes a list element of the type EX1 step EX2 until EX3. The following string is constructed and processed as if part of the Algol program after performing u := u + 1;

```
'V := EX1; Lu:  if (V - EX3) * EX2 ≤ 0
then begin SJP Mq; V := V + EX2;
go to Lu end;';
```

6. procedure A3; comment Processes a list element of the type EX1 while EX2. The following string is constructed and processed as if part of the Algol program after performing u := u + 1:

```
'Lu: V:= EX1; if EX2 then
begin SJP Mq; go to Lu end;';
```

7. procedure B; comment Writes a transfer past the subroutine for the statement subject to the for clause and a subroutine entry to it;

```
begin
    u := u + 1; TARGET ('TRA Lu');
    LABEL REQUEST (Lu); a := a + 1;
    α[a] := < Lu > ; LABEL DECLARE (Mq);
    u := u + 1; TARGET ('SSE Lu');
    LABEL REQUEST (Lu); a := a + 1;
    α[a] := < Lu >
end;
```

8. procedure C; comment Writes the exit from the subroutine enclosing the statement subject to the for clause;

```
begin
    LABEL DECLARE (C(α[a])); a := a - 1;
    TARGET ('TRA _____');
    LABEL DECLARE (C(α[a])); a := a - 1
end;
```

IX. Conditional expressions and statements

1. procedure IF; comment Writes code for making a test on a Boolean value;

```
begin
    if α[a] is sentinelled then
    TARGET ('CLA C(C(α[a]))') else
    TARGET ('CLA C(α[a])');
    if α[a] = < η[h] > then h := h - 1;
    u := u + 1;
    TARGET ('TIF Lu');
```

```
        LABEL REQUEST (Lu);
        α[a] := < Lu >
    end;
2.  procedure THEN; comment Does label manipulation for
    processing of conditional;
    begin
        LABEL DECLARE (C(α[a]));
        a := a - 1
    end;
3.  procedure ELSE; comment A transfer is written following
    the first statement or expression of a conditional;
    begin
        u := u + 1;
        TARGET ('TRA Lu');
        LABEL REQUEST (Lu);
        LABEL DECLARE (C(α[a]));
        α[a] := < Lu >
    end;
4.  procedure CC; comment Writes code to load the accumulator
    properly in the case of a conditional expression;
    begin
        if α[a] is a label then
        begin
           TARGET ('ENA C(α[a])');
           LABEL REQUEST (C(α[a]))
        end else
```

```
        begin
        if α[a] is sentinelled then
        TARGET ('CLA C(C(α[a]))') else
        TARGET ('CLA C(α[a])');
        if α[a] = < η[h] > then h := h - 1
    end;
    a := a - 1
    end;
5.  procedure CC1; comment Adjusts the auxiliary push-down;
    begin
        h := h + 1; a := a + 1; α[a] := < η[h] >
    end;
```

5. Miscellaneous Features

5.1. Input and Output Problems

The Algol report does not specify facilities for input and output. There are provisions, however, for procedures written in non-Algol code. Such procedures may be designed for input and output and may conveniently be treated in the same manner as the standard functions, i.e., considered as available without declaration. Many implementations of Algol have involved the addition of new language elements to aid the design of input and output facilities. It has been shown, however, that a satisfactory system can be devised entirely within the language [12].

The procedures should handle type-matching problems automatically. On input, for example, this means that if, say, read is a

procedure which inputs a number without format requirements, then the statement

read (x)

should have the effect of

x := the number on the input medium.

Any necessary transfer function should be invoked automatically. Since there are no format requirements, the form of the number on the input medium should be required simply to be a proper Algol number.

The problems of format, which are more significant in output, may be solved through the use of strings for specifying number patterns.

5.2. Pretranslation of Procedures

It is a not uncommon programing practice to compile subroutines separately from the rest of the program, particularly when the program is quite large. Ordinarily the result of this is a relocatable code which will be linked to the calling program by means of some sort of loading routine. The details of how this is accomplished may vary greatly with the machine and operating system, so that the problems arising may not easily be anticipated by designers of a universal language. The Algol report specifies no facilities for handling this problem. If this feature is desired, it is necessary to make an extension of the language to facilitate the linkage of procedure to calling program. This extension should be designed so as also to satisfy any requirements of the library system.

The Algol report specifically allows that a procedure body may be in non-Algol language. The compiler may therefore need to be capable

of processing one or more other languages. It may be more practical, however, to handle such procedures separately, as described above. In this way the problem is handled at the level of the operating system, allowing the procedures to be written in any language for which the system has a processor.

APPENDIX

ALCOR Hardware Representation of Algol

In the table below is given a hardware representation of Algol symbols for which substitutions are required on 48-character set peripheral equipment. This is consistent with the ALCOR convention. On 80-column cards, ALCOR considers only columns 1 through 72 as relevant to Algol program texts, leaving the remaining columns to be used for identification purposes if desired.

Only capital letters are used. One character, supposed here to be the apostrophe, is reserved as an escape symbol. It is used to delineate word delimiters and truth values. The reference symbol begin, for example, appears in the hardware representation as 'BEGIN'.

Reference Symbol	Hardware Representation
$<$	'LS'
$\leqq$	'LQ'
$=$	'EQ'(1)
$\geqq$	'GQ'
$>$	'GR'
$\neq$	'NQ'
$\neg$	'NOT'
$\wedge$	'AND'
$\vee$	'OR'
$\supset$	'IMP'
$\equiv$	'EQV'
10	'
$\times$	*

Reference Symbol	Hardware Representation
↑	**
÷	//
:	..
;	$
:=	.=
[	(/
]	/)
‘	„(2)
’	"
␣	(space)

[1]The reason for making a replacement for the symbol = is that this symbol is also tolerated as a substitution for the symbol :=. If it is not desired to allow this, no replacement is made for the symbol =. In fact the keypunching rule on page 3 breaks down if this replacement is required.

[2]The representation of string quotes using apostrophes does not provide for nested strings. This difficulty is overcome by using '(' for ‘and ')' for ’.

References and Bibliography

1. J. W. Backus, et al., "The Fortran Automatic Coding System," Proc. Western Joint Comput. Conf., Los Angeles, Calif., 1957.

2. B. Balch and T. Gallie, Jr., "Algol at Duke," Datamation 8(1962), No. 6, 33-35.

3. F. L. Bauer, et al., The Structure of ALCOR, Institut fuer Angewandte Mathematik, Mainz, Germany, 1960. Multilith ALCOR Report.

4. H. H. Bottenbruch, "The Structure and Use of Algol 60," Journ. Assoc. Comput. Mach. 9(1962), 161-221, and ORNL-3148.

5. H. H. Bottenbruch and A. A. Grau, "On Translation of Boolean Expressions," Comm. ACM 5(July 1962), 384-386.

6. H. H. Bottenbruch, "Uebersetzung von algorithmische Formelsprachen in die Programmsprachen von Rechenmaschinen," Z. fuer math. Logik u. Grundl. d. Math. 4(1958), 180-221.

7. L. L. Bumgarner, The Oak Ridge Algol Compiler for the Control Data Corporation 1604 - Preliminary Programmer's Manual, ORNL-3460 (Jan. 30, 1964).

8. Caracciolo di Forino, "Some Remarks on the Syntax of Symbolic Programming Languages," Comm. ACM 6(Aug. 1963), p. 456.

9. E. W. Dijkstra, "Operating Experience with Algol 60," The Computer Journal 5(1962), 125-127.

10. E. W. Dijkstra, An Algol Translator for the X1, Amsterdam, 1961. Multilith report translated by P. Naur.

11. A. C. Downing, "Magnetic Tape Storage Files," ORBIT Memo No. 3, ORNL (1959).

12. F. G. Duncan, "Input and Output for ALGOL 60 on KDF9," The Computer Journal 5(1963), No. 4, 341-344.

13. F. G. Duncan, "Implementation of Algol 60 for The English Electric KDF9," The Computer Journal 5(July 1962), 130-132.

14. F. G. Duncan, "Algol Translation for KDF9," Automatic Programming Information Bulletin, May 1961, No. 7, 31-32.

15. M. Feliciano, Oracle Algol Translator 1 (Preliminary Report), Mathematics Panel, ORNL, Sept., 1960. Multilith ALCOR.

16. A. A. Grau, "A Translator-oriented Symbolic Programming Language," Journ. Assoc. Comput. Mach. 9(1962), 480-487.

17. A. A. Grau, "On a Floating Point Number Representation for Use with Algorithmic Languages," Comm. ACM 5(Oct. 1962), 160-161.

18. A. A. Grau, "Recursive Processes and Algol Translation," Comm. ACM 4(Jan. 1961), 10-15.

19. A. A. Grau, The Structure of an Algol Translator, ORNL-3054 (Jan. 23, 1961).

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

21. A. A. Grau, "Multi-Segment ORBIT Programs," ORBIT Memo No. 2, ORNL (1959).

22. E. N. Hawkins and D. H. R. Huxtable, "A Multi-pass Translation Scheme for Algol 60," Annual Review in Automatic Programming 3(1963), 163-205.

23. P. Z. Ingerman, "Thunks," Comm. ACM 4(Jan. 1961), 55-58.

24. E. T. Irons and W. Feurzeig, "Comments on the Implementation of Recursive Procedures and Blocks in Algol 60," Comm. ACM 4(Jan. 1961), 65-69.

25. E. T. Irons, "A Syntax Directed Compiler for Algol 60," Comm. ACM 4(Jan. 1961), 51-54.

26. D. E. Knuth and J. N. Merner, "Algol 60 Confidential," Comm. ACM 4(June 1961), 268-72.

27. P. Lucas, "The Structure of Formula Translators (Theoretical Investigation of Translation)," Algol Bulletin Supplement No. 16, Sept. 1961.

28. Math. Panel Ann. Progr. Rept. Dec. 31, 1961, ORNL-3264.

29. Peter Naur (editor), "Revised Report on the Algorithmic Language Algol 60," Comm. ACM 6(Jan. 1963), 1-17.

30. Peter Naur (editor), "Report on the Algorithmic Language Algol 60," Comm. ACM 3(May 1960), 290-314.

31. M. Paul, Structure of the first Algol Translator at the Institute fuer Angewandte Mathematik, Mainz, Germany, 1959. Unpublished notes.

32. A. J. Perlis and K. Samelson, editors, "Preliminary Report-International Algebraic Language," Comm. ACM 1(Dec. 1958), 8-22.

33. H. Rutishauser, "Ueber automatische Rechenplanfertigung bei programsteuerten Rechenanlagen," Z. Angew. Math. u. Mech. 31(1951), 255.

34. K. Samelson and F. L. Bauer, "Sequential Formula Translation," Comm. ACM 3(Feb. 1960), No. 2, 76-83.

35. K. Sattley, "Allocation of Storage for Arrays in Algol 60," Comm. ACM 4(Jan. 1961), 60-65.

THIS PAGE

WAS INTENTIONALLY

LEFT BLANK

ORNL-3592
UC-32 — Mathematics and Computers
TID-4500 (28th ed.)

INTERNAL DISTRIBUTION

1. Biology Library
2-4. Central Research Library
5. Reactor Division Library
6-7. ORNL — Y-12 Technical Library
Document Reference Section
8. CDPF Computer Library (K-25)
9-28. Laboratory Records Department
29. Laboratory Records, ORNL R.C.
30. Nancy B. Alexander
31. E. D. Arnold
32. Susie E. Atta
33. G. J. Atta
34. Nancy Betz
35. J. E. Bigelow
36-55. L. L. Bumgarner
56. H. P. Carter
57. D. K. Cavin
58. Arline Culkowski
59. M. H. Davis (K-25)
60. W. Davis, Jr.
61. H. J. deBruin
62. L. Edwards
63. Margaret Emmett
64. O. E. Esral
65. M. Feliciano
66. Barbara Flores
67. T. B. Fowler
68. D. A. Griffin
69. M. T. Harkrider
70. A. S. Housholder
71. W. H. Jordan
72. L. Jung
73. C. E. Larson
74. M. E. LaVerne
75. E. Leach
76-79. M. H. Lietzke
80. C. McCracken
81. Mary J. Mader
82. C. D. Martin
83. F. Miller
84. R. V. Miskell
85. C. E. Parker
86. S. K. Penny
87. D. C. Ramsey
88. R. M. Rush
89. C. D. Scott
90. M. J. Skinner
91. W. J. Stelzman
92. J. G. Sullivan
93. C. D. Susano
94. J. A. Swartout
95. J. S. Watson
96. A. M. Weinberg
97. R. E. Worsham
98. H. Wright
99. J. H. Zeigler (K-25)

EXTERNAL DISTRIBUTION

100. F. L. Bauer, Mathematisches Institut der Technischen Hochschule, Munchen, Germany
101. A. C. Downing, Control Data Corporation, Computer Division, 4201 North Lexington Avenue, St. Paul, Minnesota
102-121. A. A. Grau, Department of Mathematics, Northwestern University, Evanston, Illinois
122. B. H. Mount, Manager, Mathematics Section, Westinghouse Electric Corporation, Bettis Atomic Power Laboratory, Box 1468, Pittsburgh, Pennsylvania
123. Ralph Shively, Department of Mathematics, Swarthmore College, Swarthmore, Pennsylvania
124. R. G. Stueland, Control Data Corporation, 3330 Hillview Avenue, Palo Alto, California

125. K. A. Wolf, Control Data Corporation, Computer Division, 4201 North Lexington Avenue, St. Paul, Minnesota
126. Research and Development Division, AEC, ORO
127-655. Given distribution as shown in TID-4500 (28th ed.) under Mathematics and Computers category