1969

A Translation Grammar for ALGOL 68

Victor B. Schneider

Report Number:
69-042

Schneider, Victor B., "A Translation Grammar for ALGOL 68" (1969). *Department of Computer Science Technical Reports*. Paper 343.
<https://docs.lib.purdue.edu/cstech/343>

A Translation Grammar
for ALGOL 68

Victor B. Schneider

August, 1969

CSD TR ~~40~~ 42

TABLE OF CONTENTS

I.	Introduction	1.
II.	Notation for a Translation Grammer	11.
III.	Program Structure in ALGOL 68	1.
IV.	Declarations and Data Types	2.
V.	Variables and Subscripting in ALGOL 68	9.
VI.	Data Constants and Procedure Definitions in ALGOL 68	13.
VII.	Procedure Calls and Expression Primaries	18.
VIII.	Expressions and the Precedence of Operators	21.
IX.	Operator Definitions and Declarations	24.
X.	Mode Declarations	25.
XI.	Program Structure of ALGOL 68	26.
	Bibliography	30. & 31.
	Appendix 1: System Procedures Used	32.

I. Introduction

In this paper, a translation grammar is presented for a major subset of the ALGOL 68 programming language. This translation is from ALGOL 68 into an intermediate language that was originally designed for an EULER programming system 9,11. It appears that many of the ALGOL 68 programming facilities, especially the union declaration, the use of structures, and the manipulation of arrays with flexible dimension bounds can easily and naturally be expressed using EULER concepts along with some relatively straightforward EULER system procedures. Another advantage of using an EULER intermediate language is that implementations of the Euler system exist on the UNIVAC-1108, IBM-7094, Burroughs-6500, CDC-6500, and doubtless on the IBM-360 line of computers. Therefore, it would require a relatively modest effort for any organization having one of these computers to obtain a working "first version" of ALGOL 68.

The drawbacks of using EULER to implement ALGOL 68 should be mentioned: Since EULER performs run-time type checking, it might be argued that an EULER implementation of ALGOL 68 would run more slowly than necessary. However, it would not be difficult to extend the EULER intermediate language to add operators that do not perform any type checks. At any rate, ALGOL 68 itself demands run-time type-checking because of its use of variables that can store more than one data type and because of the existence of a so-called "conformity

operator" that presupposes the existence of a run-time mechanism for keeping track of data types stored within variables. The second objection concerning use of the EULER system is that EULER lists must certainly use computer memory space less efficiently than ALGOL 68 arrays and structures could. This objection seems quite valid. We can only answer by saying that our present translation grammar certainly offers hope of a quick implementation, and that the techniques used in this "toy" translator must certainly have counterparts in a more ambitious code-producing ALGOL 68 compiler. The fact that this smaller version of ALGOL 68 retains the full expressive power of the entire language should not be overlooked.

The version of ALGOL 68 to be presented here is simplified in the sense that, for example, not all versions of iterative statements are presented; the mode declaration is only given in its basic form; declarations are ALGOL 60 -style- the only initializations possible are in the priority declaration, the operator declaration, and the mode declaration. These restrictions certainly have no effect on what can be programmed in the language, and any program written in our subset of ALGOL 68 should be immediately transferrable to a standard ALGOL 68 system.

II. Notation for a Translation Grammar

In the translation grammar that follows, an Irons-style

notation $1,2,3$ is used to specify our subset of ALGOL 68.

The following simplified programming language syntax illustrates our notation and introduces some of the basic commands used by our intermediate language:

Grammar 1. A Simple Programming Language

<u>Syntactic Rule</u>		<u>Rule of Translation</u>
$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$	$\Rightarrow$	$\langle \text{var} \rangle \langle \text{expr} \rangle$ <u>assign</u>
$\quad \langle \text{sum} \rangle$	$\Rightarrow$	I
$\langle \text{sum} \rangle \rightarrow \langle \text{sum} \rangle + \langle \text{term} \rangle$	$\Rightarrow$	$\langle \text{sum} \rangle \langle \text{term} \rangle$ <u>add</u>
$\quad \langle \text{term} \rangle$	$\Rightarrow$	I
$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle$	$\Rightarrow$	$\langle \text{term} \rangle \langle \text{factor} \rangle$ <u>multiply</u>
$\quad \langle \text{factor} \rangle$	$\Rightarrow$	I
$\langle \text{factor} \rangle \rightarrow (\langle \text{sum} \rangle)$	$\Rightarrow$	$\langle \text{sum} \rangle$
$\quad \text{at } \langle \text{var} \rangle$	$\Rightarrow$	$\langle \text{var} \rangle$
$\quad \langle \text{var} \rangle$	$\Rightarrow$	$\langle \text{var} \rangle$ <u>in</u>
$\quad \langle \text{var} \rangle . (\langle \text{expr-sequence} \rangle) .$	$\Rightarrow$	$\langle \text{expr-sequence} \rangle \langle \text{var} \rangle$ <u>in</u>
$\langle \text{var} \rangle \rightarrow \langle \text{name} \rangle$	$\Rightarrow$	<u>variable</u> $\langle \text{name} \rangle$
$\langle \text{expr-sequence} \rangle \rightarrow \langle \text{expr} \rangle$	$\Rightarrow$	I
$\quad \langle \text{expr-sequence} \rangle , \langle \text{expr} \rangle$	$\Rightarrow$	$\langle \text{expr-sequence} \rangle \langle \text{expr} \rangle$

Note that the rules of translation above refer to sequences of symbols on the right parts of syntactic rules. In this example, we see that the rules of translation specify how symbols and sequences of symbols in the source language are rearranged and rewritten in the translated language. Where no change at all is indicated in the translation of a particular

rule, the symbol "I" appears as a translation rule. As an example of how sequences of symbols are rearranged for translation, the infix addition of

$$\langle \text{sum} \rangle + \langle \text{term} \rangle$$

is translated into the reverse polish sequence of symbols consisting of a "<sum>" followed by a "<term>" followed by the intermediate-language command for adding together the values resultin from evaluation of the previous two subexpressions.

As in good polish notation, parenthesis are removed from around expressions, and this process is specified by associating the translation rule "<sum>" with the syntactic rule

$$\langle \text{factor} \rangle \rightarrow (\langle \text{sum} \rangle).$$

The remaining rules having <factor> on the lefthand side are used for translating arithmetic operands into the intermediate language. For example, the syntactic rule

$$\langle \text{factor} \rangle \rightarrow \langle \text{var} \rangle$$

indicates that operands in arithmetic expressions are variable names, and the translation of a <var> into sequence

$$\langle \text{var} \rangle \text{ in}$$

indicates that the "in" command is used for fetching the value associated with <var> and for storing that value on top of the run-time operand stack of the intermediate-language interpreter.

The other syntactic rule

$$\langle \text{factor} \rangle \rightarrow \text{at } \langle \text{var} \rangle$$

reflects the fact the intermediate language permits use of program variables that are pointers to data named by other

program variables. Hence, the effect of the "at" command of the source language is to suppress the appearance of "in" in the translated program after the translated variable name. In this case, a pointer to the data stored in <var> is left on top of the interpreter operand stack at run time. Finally, the rule

$$\langle \text{var} \rangle \rightarrow \langle \text{name} \rangle$$

means that the names of program variables are translated into the sequence "variable <name>." Here, the effect of the "variable" command is to find a pointer to the data stored in the following name at run time and to place this pointer on top of the run-time operand stack.

The sequence "<var>.(<expr-sequence>)." on the right part of the remaining <factor> rule is the definition of a function call. Function calls are translated with the parameters preceding the function name in the translated program. In this way, the function call can be made to look like a reverse polish operator having n operands, with n the number of parameters. A parameterless function call is translated exactly the same way as a program variable. Thus, the sequence

"variable <name> in"

in a translated program serves both to fetch data and to initiate a call on a function, depending on the <name> involved. This calling sequence will be referred to in the full grammar that follows.

We have attempted to make our translation grammar for

ALGOL 68 as self-contained as possible. Thus, each translation rule is followed by an explanation of what effects the intermediate language commands are producing. It should be noted that the larger grammar uses, e.g., the symbol "=" in place of the "assign" command of our small example, and, in general translates as many source symbols as possible into similar commands of the intermediate language. A full description of the intermediate language can be found in Schneider^{9,10}.

III. Program Structure in ALGOL 68

Since many programming features in ALGOL 68 are defined in terms of ALGOL 68 constructs, ALGOL 68 programs must contain an outer block in which these features are defined. Thus, an ALGOL 68 program consists of an inner block which is the "particular program" of some programmer and an outer block containing library subroutines and standard definitions. For the moment, we will concentrate on particular programs and treat the outer block as though it were an implementation-dependent set of control cards necessary to the running of ALGOL 68 programs. Thus, we can write syntactic rule 1:

1. $\langle \text{program} \rangle \rightarrow (\langle \text{standard prelude} \rangle \langle \text{block} \rangle; \text{exit:} \langle \text{standard postlude} \rangle)$

Here, the sequence "exit:" is a label definition that is global to everything in the particular program $\langle \text{block} \rangle$. Thus, a particular program can always be terminated by a statement such as

go to exit;

The left and right parentheses used to surround the outer block of a program are essentially interchangeable with begin and end, which are also symbols in ALGOL 68:

2. $\langle \text{block} \rangle \rightarrow \begin{array}{l} \text{begin } \langle \text{clause} \rangle \text{ end} \\ | \\ \langle \text{clause} \rangle \end{array} \quad \begin{array}{l} \Rightarrow \langle \text{clause} \rangle \\ \Rightarrow \langle \text{clause} \rangle \end{array}$

Except in a few cases, ALGOL 68 treats blocks as though they were expressions that have values. Thus, the statement

a := bx(c+d);

means the same thing in ALGOL 60 as in ALGOL 68: The value obtained from adding c to d is multiplied by the value of b and stored in the location denoted by a. The ALGOL 68 statement

a := b x begin real y; y := 1; y+a end

means that the value obtained from adding y to a in the block will be multiplied by the value of b and stored in the location denoted by a. Note that here the sequence

"y + a"

is treated as a statement of the language that yields a value.

IV. Declarations and Data Types

Variables declared in an ALGOL 68 block are local to that block in essentially the same fashion as in ALGOL 60 - only, in ALGOL 68, these declared variables are said to be "protected" instead of local. As near as I can determine, protection of variables involves tagging them with a higher block number than the variables of their outer block:

3. <clause> → <statementsequence> ⇒ I
 |<declarations>;<statementsequence>

⇒ \$BEGIN <declarations>;<statementsequence> \$END

In this translation rule, the \$BEGIN command of the intermediate language increments the block number count, and \$END decrements that count. In addition, the \$END command initiates a "garbage collection" procedure that dismantles arrays and data structures constructed within its block. Since more than one declaration can appear in a block, we have the additional rule:

4. $\langle \text{declarations} \rangle \rightarrow \langle \text{declaration} \rangle \Rightarrow I$
 $|\langle \text{declarations} \rangle; \langle \text{declaration} \rangle \Rightarrow I$

There are four primitive data types used in ALGOL 68 declarations. In addition, data types can be constructed that consist of structures containing the primitive types together with other invented data types. Rules (5) list the primitive types, together with the $\langle \text{indicant} \rangle$ that has the same effect as a data type declaration:

5. $\langle \text{declarator} \rangle \rightarrow \underline{\text{real}} \Rightarrow \$\text{NUMBR } 0$
 $|\underline{\text{long real}} \Rightarrow \$\text{NUMBR } 0$
 $|\underline{\text{int}} \Rightarrow \$\text{NUMBR } 0$
 $|\underline{\text{long int}} \Rightarrow \$\text{NUMBR } 0$
 $|\underline{\text{bool}} \Rightarrow \text{FALSE}$
 $|\underline{\text{char}} \Rightarrow \text{.*_}$
 $|\langle \text{indicant} \rangle \Rightarrow \$\text{VARBL} \langle \text{indicant} \rangle \IN

Note that the "long" prefix indicates an extra multiple of arithmetic precision. A primitive data type has no translation, since it provides type information to the compiler. By contrast, an $\langle \text{indicant} \rangle$ is some data type designation invented by the programmer in the course of writing his program. Naturally, a given $\langle \text{indicant} \rangle$ can only have one definition attached to it at any given point in a program:

6. $\langle \text{indicant} \rangle \rightarrow \langle \text{name} \rangle \Rightarrow I$

The basic data types can be prefixed with what are essentially attributes and then concatenated together into regular expressions that yield information about, e.g., whether a variable is a

reference to the data stored in another variable or a procedure or some structural combination of references, procedures, invented data types, etc.

```

7. <indication> → <declarator>      = <declarator>
    | ref <indication>                = $VARBL $REF
    | ref [] <indication>              = $VARBL $REF
    | ref [<boundslist>]<indication>    = $VARBL $REF
    | proc                             = .$2$.
    | proc (<typelist>)                 = .$2$.
    | proc <indication>                 = .$2$.
    | proc [] <indication>              = .$2$.
    | proc [<boundslist>]<indication>    = .$2$.
    | proc (<typelist>) <indication>     = .$2$.
    | proc (<typelist>) [] <indication>  = .$2$.
    | proc (<typelist>)[<boundslist>]<indication> = .$2$.
    | struct(<typepack>)                 = .(<typepack>).
    | union (<declaratorpack>)         = $UNDEF

```

It can be seen in rules (7) that "ref" is a prefix attribute indicating that the variable being declared is a pointer. Thus, the declaration

```
ref [1:50] real a;
```

means that the variable "a" can contain a pointer to an array of fifty floating-point numbers. The declaration

```
proc (real, int) [1:5] int u;
```

means that the variable "u" is a procedure whose two parameters are respectively "real" and "integer", and that the procedure returns a value consisting of a five-element integer array.

In our translation scheme, declarations are used to initialize the data types of variables and to enter these

variables on the run-time name table. Thus, in rules 5, all numbers are initialized to zero by the "\$NUMBR 0" sequence; logical variables are initialized by the "\$FALSE" datum; characters are initialized to a blank by the ".\$*-" sequence; and invented data types are called onto the operand stack by the "\$VARBL <indicant> \$IN" sequence that fetches the data definition associated with the invented <indicant>. In rules (7), we find that all variables whose declarations are prefixed with ref are initialized to point to an undefined system variable "\$REF". The sequence "\$VARBL \$REF" brings the pointer to \$REF onto the run-time operand stack. All variables whose declarations are prefixed with "proc" are set equal to the empty procedure definition ".\$2\$.". Finally, the "struct" declaration constructs a list of the typed elements within it and sets the declared variables equal to a copy of that list. Since all variables in our system can actually store any data type or structure, any "union" declaration simply initializes the declared variables to the value "\$UNDEF" (meaning "undefined").

The mechanism for constructing a prototype list in a struct declaration is given in (8):

```
8. <typepack> → <type><name>
      =<type>
      <typepack>→ <typepack><type><name>
      ⇒ <typepack>,<type>
```

Because the translator must retain subscripting information

for declared structures, it is assumed that a copy of each structure declaration will be stored by the translator, and that the translator can discover situations in which one structure is an element of another structure.

Rules (9) tell us that a <type> can define arrays or simple variables:

```

9. <type> → <indication>      ⇒ I
    | []<indication>          ⇒ .(<indication>).
    | [<boundslist>]<indication>
    ⇒ .(<boundslist>).<indication> $VARBL $ARRAY $IN

```

In rules (9), simple variable declarations are left unchanged, but array declarations cause lists to be constructed. In the case of empty array brackets "[]", a one element list appears in the translation. To change the size of such an array, the programmer must assign sets of values to appropriate elements of the array. Array declarations having nonempty boundslists cause the system procedure "\$ARRAY" to be called, and this procedure constructs an appropriately dimensioned list of lists, each element of which contains a datum given by the translation of <indication>. In the typedeclaration of rules (10), the translations in rules (9) are copied by the system procedure "\$COPY" that initializes and declares variables at run time. Because \$COPY returns as its value a copy of its first parameter, a chain of calls on \$COPY allows the system to handle declarations such as

```
[1:10, 1:200] real x,y,z;
```

in which three separate two-dimensional arrays must be constructed.

10. $\langle \text{typedec1} \rangle \rightarrow \langle \text{type} \rangle \langle \text{name} \rangle$
 $\Rightarrow \langle \text{type} \rangle \$\text{NEW} \langle \text{name} \rangle \$\text{VARBL} \langle \text{name} \rangle \$\text{VARBL} \$\text{COPY} \IN
 $\langle \text{typedec1} \rangle \rightarrow \langle \text{typedec} \rangle, \langle \text{name} \rangle$
 $\Rightarrow \langle \text{typedec1} \rangle \$\text{NEW} \langle \text{name} \rangle \$\text{VARBL} \langle \text{name} \rangle \$\text{VARBL} \$\text{COPY} \IN

Procedures "\$COPY" and "\$ARRAY" are documented in Appendix 1.

In the translation scheme above, the system subroutine \$COPY is set into action by the calling sequence "\$VARBL \$COPY \$IN". The two parameters of \$COPY are stored in sequence on the run-time operand stack. Its first parameter is a list or a value. Following this parameter, the \$NEW command allocates a space on the run-time nametable for the declared <name>, and then a pointer to that <name> is called onto the operand stack by the sequence "\$VARBL <name>". Thus, both parameters are loaded onto the operand stack before the \$COPY routine is called. \$COPY is programmed to return a copy of its first parameter to the top of the operand stack as its value. As a side effect, storage is allocated for <name>. Thus, the second rule of (10) works in the same way as was just described, only here, the first parameter of \$COPY is supplied by the previous call on \$COPY that was made by a <typedec>.

To complete our description at this point, we give the translations of <boundslist> and <declaratorpack>:

11. $\langle \text{boundslist} \rangle \rightarrow \langle \text{bounds} \rangle \Rightarrow \text{I}$
 $\quad \quad \quad | \langle \text{boundslist} \rangle, \langle \text{bounds} \rangle \Rightarrow \text{I}$

12. $\langle \text{bounds} \rangle \rightarrow \langle \text{sum} \rangle^{(1)} : \langle \text{sum} \rangle^{(2)} \Rightarrow \langle \text{sum} \rangle^{(1)} ; \langle \text{sum} \rangle^{(2)}$
 $\quad | \langle \text{sum} \rangle : \text{flex} \quad \Rightarrow \langle \text{sum} \rangle$
 $\quad | \text{flex} : \langle \text{sum} \rangle \quad \Rightarrow \langle \text{sum} \rangle$
 $\quad | \text{flex} : \text{flex} \quad \Rightarrow \UNDEF
13. $\langle \text{declaratorpack} \rangle \rightarrow \langle \text{uniteddeclarator} \rangle \Rightarrow \text{I}$
 $\quad | \langle \text{declaratorpack} \rangle, \langle \text{uniteddeclarator} \rangle$
 $\quad \Rightarrow \langle \text{declaratorpack} \rangle \langle \text{uniteddeclarator} \rangle$
14. $\langle \text{uniteddeclarator} \rangle \rightarrow \langle \text{declarator} \rangle \quad \Rightarrow -$
 $\quad | [\langle \text{declarator} \rangle \quad \Rightarrow -$
 $\quad | \text{union}(\langle \text{declaratorpack} \rangle) \quad \Rightarrow -$

In the translation of array bounds in ALGOL 68, one has to take into account the possibility of undefined array limits indicated by the "[" sequence or by "flex" in place of a definite upper or lower bound. As handled in the translation scheme above, all arrays translate into lists. Hence, a lower index bound of 1 always exists for each dimension of an array, whether or not the programmer asks for that lower bound. In addition, arrays with no bounds specified for some dimension contain one element undefined sublists for that dimension.

Translation of variables declared to be of type union is simple and direct, since our run-time system treats all variables as though they were capable of storing any legal data type. The translator merely keeps a record of which variables are of particular data type.

V. Variables and Subscripting in ALGOL 68.

From the syntax in the preceding section, we see that the declaration

```
struct (real x, [1:10,1:5] int y)z;
```

is legal in ALGOL 68. This declaration causes `z` to refer to a two-element structure of values whose second element is a two-dimensional array that stores fifty integers. In order to store values into `z` or extract values from `z`, these elements of `z` are referred to by name; e.g.,

```
x of z := (y of z) [8,3];
```

In this statement, the forty-third integer in the second element of `z` is converted to a real number and stored into the first element of `z`. Thus, we have two methods for subscripting variables in ALGOL 68, and these methods can be used separately or in combination.

Another feature of variable usage in ALGOL 68 is that variables can be "selected" before being subscripted. Thus, the statement

```
a := if g1 then b else c fi [5,3];
```

is valid in ALGOL 68 if `a`, `b`, `c` and `g1` are appropriately declared, and the subscripts "[5,3]" are within the bounds of `b` and `c`. Another, essentially similar, usage is one in which a case statement (similar to the case statement in ALGOL W) is used for selecting a variable:

```
if g1 then d else e fi := case i in a,b,c esac [2,4];
```

This use of conditional expressions for selecting variables in ALGOL 68 leads to a translation difficulty in which the translator is unable to decide whether it is translating a chain of variable references or expressions yielding values because either interpretation is correct for a conditional expression. To get around this difficulty, we have decided to write our translation grammar so as to treat every expression as though it were a variable reference until the last possible minute. Thus, we obtain the following strange-looking syntax of subscripted variables:

```
15. <name> ~ <letter>      = I
      | <name><letter>      = I
      | <name><digit>       = I
```

In rules (15), a table lookup mechanism informs the translator that some <name> is actually a declared variable. Then, rules (16) are applied.

```
16. <subscriptvar> ~ <name>      = $VARBL<name>
      | <block>                  = I
      | (1)if<clause> (2)then<clause> (3)else<clause> (3)fi
      = (1)<clause> $IF (2)<clause> $THEN (3)<clause> ;
      | case <clause> in <clausesequenc> esac
      =><clause>.(.$ <clausesequenc>$.). $VARBL $CASE $IN
      | <subscriptvar>[<subscripts>]
      => <subscriptvar> <subscripts>}
```

In rules (16) above, it should be explained that the translator must have an operand stack that permits it to keep track of data types associated with variables and expressions.

With this operand stack, the translator will only allow subscripting to follow an expression whose value is a variable reference. In translating the conditional statement of rules (16), the translator also uses a stacking mechanism to supply program labels to the conditional branch command "\$IF" and the unconditional branch command "\$THEN". These labels are indicated schematically in the rules by the " " notation. Finally, a case statement is translated into a two-parameter procedure call on the system procedure "\$CASE". The first parameter is an expression that has a subscript as its value, and the second parameter is a run-time list of parameterless procedure definitions that is to be subscripted by the first parameter in the process of calling one of the list elements.

This list of procedure definitions is constructed by the following translation rules:

$$\begin{aligned}
 17. \quad & \langle \text{clausesequences} \rangle \rightarrow \langle \text{clause} \rangle \Rightarrow .\$ \downarrow \langle \text{clause} \rangle \$ \uparrow \\
 & | \langle \text{clausesequences} \rangle \rightarrow \langle \text{clausesequences} \rangle, \langle \text{clause} \rangle \\
 & \Rightarrow \langle \text{clausesequences} \rangle, .\$ \downarrow \langle \text{clause} \rangle \$ \uparrow
 \end{aligned}$$

A typical procedure definition in the translation of $\langle \text{clausesequences} \rangle$ would look like the sequence

".\\$ \downarrow \langle \text{clause} \rangle \\$ \uparrow "

Here, the "\$" command tells the run-time system that what follows is a procedure definition. It therefore leaves as its value the value of the translated program location counter that begins the translated $\langle \text{clause} \rangle$. Since the procedure

is not activated when the procedure definition is assigned to some variable, the "\$" command is followed by a jump to the code directly following the procedure definition. The "\$." command is part of the code in the procedure definition. The "\$." is executed as a return jump command that looks up a return label on a table of return jumps and transfers control back to that point in the program where the procedure was called.

Finally, for our particular choice of intermediate language, subscripting is accomplished by the ")" command. This ")" command assumes that the topmost operand of the run-time operand stack is an integer number, and that the next-to-top operand is a reference to a list cell. Thus, we have the translated sequence

<subscriptvar><subscripts>)

given in rules (16), and the following rules for translation of <subscripts>:

18. <subscripts> → <sum> = I
 !<subscripts>,<sum>=<subscripts>)<sum>

To complete this description of subscripting, we introduce the next layer of rules above rules (17) in the system of precedence:

19. <selection> → <subscriptvar> = I
 |<name> of <selection>
 ⇒ <selection> } [<name>]]

Thus, in our syntax, the numerical subscripting of (16) takes precedence over the logical subscripting of (19) because

of the ordering of rules. When a variable is logically subscripted as described in (19), the translator provides a numerical subscript to the translated program, and this numerical subscript corresponds to the position in its own structure of the logical subscript that is used. The notation

$$\mathcal{T}[\langle \text{name} \rangle])$$

in rules (20) represents the translator-supplied subscript number followed by the subscripting command. This subscripting capability of course implies that the translator must itself keep track of structures and pointers from one element of a structure to another structure. In this way, the translator must have a list processing capability for tracing lists and sublists to any desired depth of nesting.

VI. Data Constants and Procedure Definitions in ALGOL 68

At this point, it is convenient to introduce the data types used in the language. Data of type char (character) is denoted by the following syntax:

21. $\langle \text{charprim} \rangle \rightarrow \langle \text{alphameric} \rangle = . * \langle \text{alphameric} \rangle$

The sequence $. * \langle \text{alphameric} \rangle$ stores that symbol on top of the run-time operand stack. Here, $\langle \text{alphameric} \rangle$ is whatever set of symbols are available for a particular computer.

By convention, a quote symbol (") is represented by a pair of quotes (") in the language. Thus, the assignment

$$v := "";$$

stores a single quote in character variable v.

Data of type bool (logical) is denoted by the following syntax:

22. <logicalprim> → true ⇒\$TRUE
 ! false ⇒\$FALSE

The \$TRUE (\$FALSE) command stores the internal representation of logical truth (or falsity) on top of the run-time operand stack.

Although ALGOL 68 allows real and integer numbers, as well as multiple precision versions of numbers, we have for simplicity translated all numbers into single-precision floating point:

23. <number> → <integer> ⇒ \$NUMBR] [<integer>]
 !<integer>.<integer> ⇒\$NUMBR] [<integer>.<integer>]

In both cases of rules (23), the command "\$NUMBR" is followed by the internal floating point representation of the appropriate character strings. "\$NUMBR" serves to place the following translated word on top of the run-time operand stack.

For convenience in initializing small arrays, ALGOL 68 allows structures similar in appearance to lists. So, we will call them lists, instead of using the ALGOL 68 name for them:

24. <listprim> → (<clause>,<listend>
 ⇒.<clause>,<listend>
 <listend> → <clause>) ⇒ <clause>).
 !<clause>,<listend> ⇒ <clause>,<listend>

Here, if we use the declaration "[1:2] real x" and follow this by the assignment "x:=(1.0,3.5);" we can see how the lists are used. Thus, these list primaries translate directly into our own notation, with the exception that lists of zero or one elements cannot be represented using list notation. This is to avoid an ambiguity, e.g., in which the sequence "(1)" is interpreted as being both a <block> and a <listprim>.

In an analogous vein, ALGOL 68 has provisions for representing strings of symbols:

```
25. <stringprim>->"<alphameric><stringend>
    =.(.*<alphameric>,<stringend>
    <stringend>-><alphameric>"=.*<alphameric>).
    |<alphameric><stringend>=.*<alphameric>,<stringend>
```

Thus, a string translates into an array of characters in the same fashion as in ALGOL 68. Here again, strings of length zero and one cannot be directly represented to prevent ambiguity. As we will see, it is possible to invent a data type called string whose variables all consist of character arrays with flexible upper bounds.

Finally, we include procedure definitions in this section because procedure definitions can be assigned to variables in the same way as other data constants. We have chosen to use an older version of the ALGOL 68 procedure definitions, because the keyword "expr" used in these older versions is subject to fewer usages in the language than the ":" that replaces expr in the syntax. We have also chosen

to force the programmer to be explicit about his choice of procedure definitions in our version. Thus, the assignment statement

`v1:= if g1 then expr v2 else expr v3 fi;`

(Assign procedure definition v2 to v1 if g1 is true, otherwise assign procedure definition v3.) is legal in our version of ALGOL 68 as well as the "full" version of the language.

But we do not allow the alternate assignment statement:

`v1:= if g1 then v2 else v3 fi;`

even though this is also legal in ALGOL 68 and is equivalent to the first statement. We thus have the syntax of procedure definitions in rules (26):

26. `<procdef> → expr<assignment>~.$_<assignment>$.1
 |<indication>expr<assignment>~.$_<assignment>$.1
 |(<p. list>)expr<assignment>~.$_<p. list><assignment>$.1
 |(<p. list>)<indication>expr<assignment>
 ~.$_<p. list><assignment>$.1`

These procedure definitions are of the same form as described in rules (18). However, here the procedure definition symbols "\$" and "\$" surround what will be seen to be single statements. In order for a procedure to be permitted to return a value in ALGOL 68, the type of its returned value must be indicated in the procedure definition. Thus, all the procedure definitions having an <indication> before the expr keyword behave like ALGOL 60 or FORTRAN functions, while the remaining definitions are similar to procedures in those languages. Finally, we have the

translation syntax of the parameter list:

```

27.  <p.list> → <type><name>      ⇒ $FORMA<name>
      |<p.list>,<name>           ⇒ $FORMA<name><p.list>
      |<p.list>,<type><name>      ⇒ $FORMA<name><p.list>

```

Here, the formal parameters are translated into a reverse sequence. The command "\$FORMA" has the effect of entering the following <name> of a parameter onto the run-time name table. \$FORMA then fetches the topmost operand on the run-time operand stack, and stores its value and type into the newly declared parameter <name>. Because of this method for passing parameters to procedures, the old call-by-name and call-by-value conventions in ALGOL 60 procedures have been abandoned in favor of allowing the programmer to pass procedure definitions, pointers to variables, and actual values into his procedure calls. As an illustration, if v1 is a reference variable, v2 is an integer variable, and v3 is an appropriate procedure name, then the procedure call

```
v3(v1, int expr v2,5)
```

passes as parameters to the procedure definition of v3 a pointer stored in v1, a procedure definition whose body "calls" v2, and an integer value. As we will see, the procedure calling mechanism puts these three parameters onto the operand stack in the right sequence for allowing the \$FORMA commands to pick off their values for the formal parameters.

VII. Procedure Calls and Expression Primaries

In a number of preceding rules, we implicitly assumed the existence of a procedure calling mechanism in our intermediate language. This mechanism works as follows: A typical procedure call is of the form

`<value1><value2> ...<value n>$VARBL<name>$IN.`

The n values are stored in sequence on the run-time operand stack, and then the command sequence "`$VARBL <name> $IN`" passes control to the procedure. This procedure is assumed to have n formal parameters, and therefore n "`$FORMA<name>`" sequences at the beginning. These "`$FORMA<name>`" sequences pick off the values stored on the operand stack in the reverse of the sequence in which they were stored. Hence, any procedure calling system must have a mechanism for matching actual and formal parameters. When we discuss the extendible operations feature of ALGOL 68, we will justify the necessity for our use of this procedure calling command sequence.

The mechanism needed for matching actual and formal parameters is further complicated by the legality of ALGOL 68 procedure calls such as in the following example:

`if g1 then p1 else p2 fi [3,9](4, real expr p3);`

Here, one of two procedure arrays is chosen by a conditional statement. Then, the chosen procedure array is subscripted, and finally, the actual parameters are supplied when that procedure element is called. To explain our method for translating such ALGOL 68 procedure calls, we give the following

syntax of procedure calls:

28. $\langle \text{procall} \rangle \rightarrow \langle \text{subscriptvar} \rangle (\langle \text{a.p. list} \rangle)$
 $\Rightarrow \langle \text{subscriptvar} \rangle . (\langle \text{a.p. list} \rangle) . \$\text{VARBL } \$\text{XEQUT } \IN

We see that a program procedure call with parameters is translated into a call on a system procedure named "\$XEQUT". This is accomplished by transforming the program procedure call into two parameters. The first parameter is a (possibly subscripted or selected) pointer to where the procedure call is stored on the run-time name table. The second parameter is a link to a list constructed from the actual parameter list of the program by rules (28) and (29):

29. $\langle \text{a.p. list} \rangle \rightarrow \langle \text{assignment} \rangle \Rightarrow \text{I}$
 $\quad \quad \quad | \langle \text{a.p. list} \rangle , \langle \text{assignment} \rangle \Rightarrow \text{I}$

The listing of the \$XEQUT procedure can be found in Appendix 1.

A procedure call without parameters can be treated syntactically like an ordinary value primary of the language. This is because the sequence

$\langle \text{selection} \rangle \IN

is sufficient to execute procedures as well as to bring values referenced by $\langle \text{selection} \rangle$ pointers to the operand stack.

We can thus begin to give a syntax for expression primaries in ALGOL 68:

```

30. <prim> → <procall>      ⇒ I
      | <procdef>           ⇒ I
      | <stringprim>        ⇒ I
      | <listprim>          ⇒ I
      | <number>            ⇒ I
      | <logicalprim>       ⇒ I
      | <charprim>          ⇒ I
      | <selection>         ⇒ <selection> $IN
      | <referenceprim>     ⇒ I
      | val <prim>         ⇒ <prim> $IN

```

In rules (30) above, the \$IN command is supplied by the translator to fetch values in two instances. For the first instance to apply, the translator program must determine by inspection of its operand stack that <selection> appears after the left part of an assignment statement such that the left part is not of type reference. This is because the assignment statement,

$$a := b;$$

where "a" is a variable of type reference, is legal in ALGOL 68. Thus, "b" must be treated as a <referenceprim>, and the translator must determine by context (using its own operand stack) whether or not some <selection> is a <referenceprim> as in rule (31) or a value-bearing <prim> as in rules (30):

```

31. <referenceprim> → <selection> ⇒ I

```

Of course, the sequence

val <prim>

means that the programmer explicitly desires to fetch a value to which a <referenceprim> refers. So as to spare the translator from the necessity of trying to discover how many layers of pointers must be traced through in order to fetch a value, we will require the use of val whenever a <referenceprim> is to be "depressed" to a value in the middle of an expression.

VIII. Expressions and the Precedence of Operators

In ALGOL 68, there are ten levels of operator precedence. Corresponding to each level is a set of standard operators for that level. These operators can be redefined by the programmer, who may change their precedences, introduce new procedures that describe their actions, and introduce new operators to suit his convenience. The syntax for such redefinable operations must take into account this new facility:

```

32. <unary> → <op 10><prim> ⇒ <prim>$VARBL<op 10> $IN
      | go to <prim>      ⇒ <prim> $GOTO
<complex> → <unary>      ⇒ I
      | <complex><op 9><unary>
      ⇒ <complex><unary>$VARBL<op 9>$IN
<exponent> → <complex>   ⇒ I
      | <exponent><op 8><complex>
      ⇒ <exponent><complex>$VARBL<op 8>$IN

```

cont.

```

<product> → <exponent>      ⇒ I
          | <product><op 7><exponent>
⇒ <product><exponent>$VARBL<op 7>$IN

<sum> → <product>            ⇒ I
       | <sum><op 6><product>
⇒ <sum><product>$VARBL<op 6> $IN

<inequality> → <sum> ⇒ I
              | <sum>(1) <op 5> <sum>(2)
              ⇒ <sum>(1) <sum>(2) $VARBL<op 5>$IN

<confrontation> → <inequality> ⇒ I
                 | <inequality>(1) <op 4> <inequality>(2)
                 ⇒ <inequality>(1) <inequality>(2) $VARBL<op 4>$IN

<conjunction> → <confrontation> ⇒ I
               | <conjunction><op 3><confrontation>
               ⇒ <conjunction><confrontation>$VARBL<op 3>$IN

<disjunction> → <conjunction> ⇒ I
               | <disjunction><op 2><conjunction>
               ⇒ <disjunction><conjunction>$VARBL<op 2>$IN

<assignment> → <disjunction> ⇒ I
              | <selection><op 1><assignment>
              ⇒ <selection><assignment>$VARBL<op 1>$IN

```

The complete table of standard ALGOL 68 operators is given in section 8.4.2 of (11). In rules (32) above, we use operator categories <op 1>, ..., <op 10> to replace the usual arithmetic, logical, and relational operators that appear in similar grammars. For the translator to know

which category an operator belongs to, it must have a table of legal operators similar to its nametable, and with each operator will be an associated level number. To each of these operators there corresponds either a standard intermediate-language operation (in which case, the intermediate language operation is written into the translated program) or a procedure definition (in which case the procedure call "\$VARBL <op_n> \$IN" is written into the translated program). Procedures defining the standard operations and their effects when executed are given in section 10.2 of (II).

It should be mentioned that we would include several operations that are not in the ALGOL 68 table. For example, the standard operations of <confrontation> are the relational "=" and "≠". In addition to these standard operators at that level, the ALGOL 68 conformity symbol "::" (which checks whether two expressions are of the same mode) and the ALGOL 68 identity symbol "==" (which asks whether two expressions yield references to the same <name>) are included because they are used in essentially the same way as "=" and "≠".

Note also that the definition of a jump instruction in ALGOL 68 is put into the <unary> rule of (32) because its precedence in the language is compatible with that level of the grammar. However, the "go to" operation is most emphatically not redefinable, and so is listed separately.

IX. Operator Definitions and Declarations

Now that we have seen a syntax for expressions, we can discuss the syntax of operator declarations and priority declarations in ALGOL 68. A priority declaration has the form

33. $\langle \text{priority decl.} \rangle \rightarrow \underline{\text{priority}} \langle \text{operator} \rangle = \langle \text{priority} \rangle$
 $\quad \quad \quad ! \langle \text{priority decl.} \rangle, \langle \text{operator} \rangle = \langle \text{priority} \rangle$

A priority declaration is not translated, since its role is to provide information to the operator table of the translator. Naturally, the $\langle \text{priority} \rangle$ is some $\langle \text{integer} \rangle$ from 1 to 10:

34. $\langle \text{priority} \rangle \rightarrow 1 | 2 | \dots | 10$

An operator declaration takes the form

35. $\langle \text{operator decl.} \rangle \rightarrow$
 $\quad \quad \quad \underline{\text{op}} \langle \text{operator} \rangle = (\langle \text{p.list} \rangle) \langle \text{indication} \rangle \underline{\text{expr}} \langle \text{assignment} \rangle$
 $\quad \quad \quad \Rightarrow \$\text{NEW} \langle \text{operator} \rangle \$\text{VARBL} \langle \text{operator} \rangle . \$\text{L} \langle \text{p.list} \rangle \langle \text{assignment} \rangle \$\text{I} =$

In (35), the translator enters the new $\langle \text{operator} \rangle$ name onto its operator table and translates the operator declaration as a new procedure definition. Although the $\langle \text{p.list} \rangle$ mechanism is used for simplicity in the translation process, an operator declaration is meaningless unless the $\langle \text{p.list} \rangle$ consists of only one or two parameters. Moreover, as in ALGOL 68, it is assumed that all unary operators have a non-redefinable priority of 10. This is because of the ambiguity that would result if a programmer attempted to

redefine the precedence of unary "+" or "-", where the binary addition and subtraction have the same denotations.

As an example of an operator declaration, we give here our version of subtraction with real operands in ALGOL 68:

```

op-=(ref real a,b) real expr val a minus val b;
op-=(ref real a, real b) real expr val a minus b;
op-=(real a, ref real b) real expr a minus val b;
op-=(real a,b) real expr a minus b;

```

Here, the operator "-" is translated into the intermediate-language command for subtracting the two topmost values on the run-time operand stack. This definition should be compared with definition 10.2.4(g) of the ALGOL 68 report (11), where the definition of subtraction is given in words and then addition and negation are programmed in terms of this subtraction operator.

X. Mode Declarations

As a complement to the facility for defining new expression operators, ALGOL 68 allows the definition of new data types and structures of data types. This is accomplished by the mode declaration:

```

36. <mode.decl.> -mode <indicant>=<indication>
    =$NEW<indicant>$VARBL<indicant><indication>=

```

Here, the translation of <indication> produces some initial value, array, or structure. The <indicant> is translated as though it were a newly declared program variable, and the translation of <indication> becomes its value. This means that, along with its name table and operator table, the translator must have a table of indicants that stores links at translation to any structures that may be assigned to indicants. Thus, when a variable is logically subscripted whose mode is a structure, the translator can find its own prototype copy of that structure and supply appropriate numerical subscripts in the translation.

Because the <indicant> is treated as a program variable by the run-time system (and as a mode declarator by the translator), the translation of <indicant> given in rules (5) presents to the \$COPY procedure a list which is copied and assigned to all variables later declared to be of the mode given by the <indicant>.

XI. Program Structure of ALGOL 68.

At this point, we can complete our description of ALGOL 68 program structure and fill in some remaining details concerning implementation of the language: First, we can draw together the different declarations outlined so far:

37. <declaration> → <mode decl.> ⇒ I
 |<operator decl.> ⇒ I
 |<priority decl.> ⇒ I
 |<type decl.> ⇒ I

We can then complete our definition of a program <statement>:

38. <statementsequence> → <labelstat.> ⇒ I
 |<statementsequence>;<labelstat.> ⇒ I

The effect of the semicolon in rules (38) should not be ignored. Its effect as an intermediate-language command is to unstack the topmost operand of the run-time operand stack. Because of the semicolon, the <block>

(x:=x+1;2+x)

causes the intermediate value "x+1" to be unstacked, but leaves the value of "2+x" on the operand stack as the value of the <block>. This same mechanism is used for returning function values, since the last value on the operand stack before exit from a procedure is its value.

Following the scheme of precedence in our syntax, we next define labelled statements:

39. <labelstat> → <statement> ⇒ I
 |<name>:<labelstat>
 ⇒ \$LABEL <name> <labelstat>

Here, the sequence "<name>:" invokes a label declaration in the translated program. The notation " " means that the value of the translated program location counter plus one

is inserted following "\$LABEL<name>" in the translated program. Strictly speaking, the translator will also have a mechanism for writing out

\$LABEL<name>

whenever a go to statement is translated before the appropriate label is encountered. In addition, it should be noted that program labels are treated as variable names by the syntax of (15), (16), (19), (30) and (32).

Next, we have a syntax for <statement>:

40. <statement> → <assignment> = I

<statement> → for<selection>from<clause>⁽¹⁾by<clause>⁽²⁾to<clause>⁽³⁾
do<statement>

⇒ <selection><clause>⁽¹⁾<clause>⁽²⁾<clause>⁽³⁾ . \$_L <statement> \$_L
\$VARBL \$FORSL \$IN

<statement> → from<clause>⁽¹⁾by<clause>⁽²⁾to<clause>⁽³⁾ do<statement>
⇒ <clause>⁽¹⁾<clause>⁽²⁾<clause>⁽³⁾ . \$_L <statement> \$_L
\$VARBL \$FOR \$IN

<statement> → while <clause> do <statement>

⇒ . \$_L <clause> \$_L . \$_L <statement> \$_L \$VARBL \$WHILE \$IN

Listings of the system procedures "\$FORSL", "\$FOR", and "\$WHILE" can be found in Appendix 1.

With these forty sets of rules, we have completed a description of the essential features of ALGOL 68. Missing from the syntax is any built-in procedure for input-output, as well as any description of formatting. As the language

is described in this report, formatless input and output procedures for this language can be written that are quite similar to the procedures given in section 10.5.2 of the ALGOL 68 report (11). Such input-output routines have been programmed for the CDC-6500 computer at Purdue University, and are currently being tested together with the remaining components of the ALGOL 68 translator.

Bibliography

1. Floyd, R. W. A Descriptive Language for Symbol Manipulation.
J. ACM 8 (Oct. 1961), 579-584.
2. Irons, E. T. A Syntax-Directed Compiler for ALGOL 60.
C. ACM 4 (Jan. 1961), 51-55.
3. Lewis, P. M. Stearns, R. E. Syntax-Directed Transduction.
J. ACM 15 (July 1968), 465-488.
4. Mills, David L. The Syntactic Structure of MAD/1. Defense
Documentation Center Report Number AD-671-683, June,
1968.
5. Neur, P. (Ed.) Revised Report on the Algorithmic Language
ALGOL 60. C. ACM 6 (Jan. 1963), 1-17.
6. Schneider, V. B. The Design of Processors for Context-Free
Languages. National Science Foundation Memorandum,
Department of Industrial Engineering and Management
Science, Northwestern University, Evanston, Illinois,
August, 1965.
7. ———. Pushdown-Store Processors of Context-Free
Languages. Doctoral Dissertation, Northwestern
University, Evanston, Illinois, 1966.
8. ———. A System for Designing Fast Programming
Language Translators. Technical Report 68-76, Computer
Science Center, The University of Maryland, College
Park, Md., July, 1968. Also in Proc. Spring Joint
Comp. Conf. (1969).
9. ———. A Translator System for the EULER Programming
Language Technical Report 69-85, Computer Science
Center, The University of Maryland, College Park,
Md., January, 1969.
10. ———. Some Syntactic Methods for Specifying
Extendible Programming Languages. Proc. Fall Joint
Comp. Conf. (1969).

11. Van Wijngaarden (Ed.). Report on the Algorithmic Language ALGOL 68. Mathematisch Centrum Report MR-101, Amsterdam, February, 1969.
 12. Weizenbaum, J. A Symmetric List Processor. C. ACM 6 (Sept. 1963), 524
 13. Wirth, N. A Generalization of ALGOL. C. ACM 6 (Sept. 1963), 547-554
 14. Wirth, N. and Weber, H. EULER: A Generalization of ALGOL and Its Formal Definition: Parts I and II. C. ACM 3 (Jan. - Feb. 1966), 13-25 and 89-99.
-

Appendix 1. System Procedures Used

The following "system procedures", with the exception of the \$XEQT routine, are written in Wirth and Weber EULER¹⁴. It is understood that their translated versions are supplied by the ALGOL 68 translator to the run-time interpreter system as part of the <standard prelude> of a translated <program>. Hence, these procedures are globally defined in every translated ALGOL 68 program.

```
$WHILE + 'formal clause; formal stat; if clause then
          (stat; $WHILE('clause','stat')) else  $\Omega$ ';
```

```
$FORSL + 'formal var; formal from;
          formal by; formal to; formal stat;
          begin new sign; label cycle;
          sign + if by < 0 then -1 else +1;
          var + from;
          cycle: if (to - var) x sign > 0
          then begin stat; var + var + by;
          go to cycle end else  $\Omega$  end;
```

```
$FOR + 'formal from; formal by;
        formal to; formal stat;
        begin new index;
        $FORSL(@ index, from, by, to, 'stat') end';
```

```
$CASE + 'formal subscript; formal statementlist;
        statementlist[subscript]';
```

```
$COPY + 'formal structure; formal var;
```

```

var- + if islist structure then
begin new dimension; new index;
  dimension + list (length structure);
$FORSL (@index, 1, 1, length structure,
  '$COPY(structure[index],@ dimension[index])');
  dimension end
  else structure';

```

```

$ARRAY + 'formal boundslist; formal value;
begin new dimension; new index;
  dimension + list boundslist [1];
  if length (boundslist) = 1
  then $FORSL (@ index, 1, 1, boundslist[1],
    '$COPY(value, @ dimension[index])')
  else $FORSL (@ index, 1,1, boundslist[1],
    'dimension [index] + $ARRAY(tail boundslist,value)');
  dimension end';

```

If it were legal in EULER to omit the semicolon between statements, thus leaving the values of preceding statements on the operand stack without erasing them, the \$XEQT procedure could be written in EULER as follows:

```

$XEQT + 'formal var; formal paramlist;
  (paramlist [1] if length (paramlist)>1
  then $XEQT (var, tail paramlist)
  else var-)' ;

```

The effect of the procedure above is to place all the parameters of the procedure call onto the run-time operand stack, and then call the procedure using the "var-" statement. Since the semicolon is missing between

"paramlist [1]" and "if", the effect of the procedure call is to recursively place the first element of paramlist onto the operand stack, and successively "pop off" the top of the paramlist in each recursive use of paramlist in the call "\$XEQT(var, tail paramlist)". In our intermediate language, the \$XEQT procedure can be (correctly) written as follows:

```
$VARBL $XEQT . $37 $FORIA PARLST
$FORIA VAR $VARBL PARLST $NUMBR 1) $IN
$VARBL PARLST $IN $LENGT $NUMBR 1 $GT
$IF 13 $VARBL VAR $IN $VARBL PARLST $IN $TAIL
$VARBL $XEQT $IN $THEN 5 $VARBL VAR $IN $IN $.=;
```