COMPUTER SCIENCES CENTER

PURDUE UNIVERSITY

User's Manual

for the

ALCOR-University of Illinois

ALGOL-60 Translater

April 5, 1965

CONTENTS

1. Introduction

The purpose of this manual is to explain the differences which exist between ALGOL-60 as it is defined in "Revised Report on the Algorithmic Language ALGOL-60" (Ref. 1) and as it actually has been implemented by the ALCOR-University of Illinois programming group. Relatively few features of ALGOL-60 have not been implemented, so this manual consists mostly of an explanation of the "hardware" representation of ALGOL rather than deviations from it.

This manual is specifically addressed to those individuals who have a working knowledge of ALGOL and merely want information to enable them to transform their programs into the "hardware" form and get them onto the computer. This manual is in no way to be regarded as a primer in programming, in ALGOL or any other programming language.

2. Hardware Representation

2.1. Introduction

A large part of the basic symbols of ALGOL are not available at present as the symbols usable by computer systems. Therefore, in order to use ALGOL at all, certain alterations have had to be made to the ALGOL symbol set.

Computer word size imposes practical limitations on the magnitude of numbers, number of significant digits, etc., used in ALGOL programs. Size of core storage effectively limits the size of arrays. Some of the more important restrictions are discussed in the following sections. Omitted from discussion are those too obvious to merit special, detailed attention (e.g., a computer core having 32,768 words cannot store an array of 2,000,000 elements) and those which exist, but will cause trouble only in the rarest of instances (e.g., only 4096 different identifier names are actually permitted).

In the tables which follow, two hardware representations of some symbols are shown under the headings of "hardware representation" and "tolerated representation." These optional representations are called "tolerated," and in every case where such a representation occurs in a source program, the user will be so advised on his output. Not all symbols have both representations and a blank in the tables under "tolerated hardware representation" implies that no such representation exists.

2.2. Conventions and Restrictions

The most obvious restrictions are that subscripts and superscripts cannot be used. But there are others, and each symbol group will be discussed in turn. For the moment, we will consider a more important class of unavailable symbols, that of the word symbols, begin, end, if, go to, etc. Clearly, we cannot use bold-face print or underline certain words if we expect to enter the information into a computer. Another convention must be adopted to identify these word symbols and distinguish them from the apparently identical English words "begin," "end," "if," "go to," etc., to which they bear no other relation. The convention which has been adopted here is that of enclosing each of the word symbols in a pair of "escape symbols", apostrophes; that is, in hardware representation the word symbol begin becomes 'BEGIN', end becomes 'END', go to becomes 'GO TO'. It should be noted that except in strings, blanks are ignored in hardware representation, just as in publication ALGOL, so that 'GO TO' is identical to 'GOTO' or 'GO   TO' and 'BE   GIN' is as acceptable as 'BEGIN'.

Minor restrictions not mentioned in the ALGOL Report have been imposed by the nature of the 7094 computer. The word size of the 7094 is 35 binary bits plus sign bit; in floating point arithmetic, 8 of these 35 bits are used for the exponent part. This Translator does not distinguish internally between integer arithmetic and real arithmetic, but does both in floating point mode. Hence, the following restrictions exist on number size:

$$-38 < \text{exponent}_{10} < +38$$

$$|\text{integer}| < 134{,}217{,}727_{10}$$

2.3. Alphabet and Numerals

Note that begin has become 'BEGIN' and not 'begin' or 'Begin'. Standard character sets for computers do not normally include lower case as well as upper case alphabetic characters, so lower case letters, though permitted in publication ALGOL, will not be used in hardware representation.

Numerals appear in publication ALGOL in only one form; hence no alteration of numerals is necessary in hardware representation, since this form is identical to that in computer symbol sets.

2.4. Word Symbols

A large part of the ALGOL symbols are word symbols and a list of the hardware representation is given below.

The careful reader will note that two non-ALGOL word symbols, 'CODE' and 'FINIS', appear at the end of the list below. The introduction of these word symbols was made necessary by implementation requirements, and neither interferes in any way with the action of any other word symbol. They are discussed in Sections 2.9. and 2.10. respectively.

Publication Language Symbol	Hardware Representation
go to	'GO TO'
if	'IF'
then	'THEN'
else	'ELSE'
for	'FOR'
do	'DO'
step	'STEP'
until	'UNTIL'
comment	'COMMENT'
begin	'BEGIN'
end	'END'
own	'OWN' (Not Implemented)
Boolean	'BOOLEAN'
integer	'INTEGER'
real	'REAL'
array	'ARRAY'
switch	'SWITCH'
procedure	'PROCEDURE'
string	'STRING'
label	'LABEL'
value	'VALUE'
code	'CODE'
finis	'FINIS'

2.5. Boolean Values

The two values of Boolean variables and expressions, true and false, are ALGOL word symbols. Their hardware representation is analogous to that of other word symbols: true is represented as 'TRUE' and false as 'FALSE'. The actual internal representation of true and false is

true	777777777777
false	000000000000

The last statement should not be misinterpreted as meaning that true and false values of Boolean variables can be read as data in the forms shown above. This is not true at all; there does not exist a procedure by which Boolean values may be read directly as data. It is suggested that if a user wishes to input Boolean values, he use the real values 0 and 1 for false and true respectively and use these real values to create Boolean values by means of, for example, an if statement. There are, of course, other means of accomplishing the same end.

2.6. Arithmetic Operators

Some, but not all, of the ALGOL arithmetic operators are available in the 7094 and 1401 character sets. The hardware representations of all the arithmetic operators are tabulated below.

ALGOL Symbol	Description	Hardware Representation	Tolerated Representation
+	plus	+	
-	minus	-	
X	multiplication	*	
/	division	/	
÷	integer division	//	
↑	exponentiation	'POWER'	**

2.7. Logical and Relational Operators

The logical and Boolean operators are not available in the symbol set, and all have been transliterated into word symbols enclosed by escape symbols. For the convenience of the skilled ALGOL programmer, an alternate, "tolerated," set of abbreviated word symbols for these operators is provided. The beginning ALGOL programmer will doubtlessly wish to use the long form, since there is less chance for human misinterpretation.

ALGOL Symbol	Description	Hardware Representation	Tolerated Representation
$<$	less than	'LESS'	'LS'
$\leq$	less than or equal to	'NOT GREATER'	'LQ'
$=$	equal to	'EQUAL'	'EQ'
$\geq$	greater than or equal to	'NOT LESS'	'GQ'
$>$	greater than	'GREATER'	'GR'
$\neq$	not equal to	'NOT EQUAL'	'NQ'
$\equiv$	logical equivalent	'EQUIV'	'EQV'
$\supset$	logical implies	'IMPL'	'IMP'
$\vee$	logical or	'OR'	
$\wedge$	logical and	'AND'	
$\neg$	logical negation	'NOT'	

2.8. Separators, Brackets and Others

This category of symbols includes all those not previously discussed. The following list shows each publication symbol, its hardware representation and its tolerated hardware representation, if any.

OK

Publication Language	Description	Hardware Representation	Tolerated Hardware Representation
,	comma	,	
.	decimal point	.	
$_{10}$	base 10	'	
:	colon	..	
;	semicolon	.,	$
:=	assignment symbol	.=	=
# or b	blank space		
(	left parenthesis	(	
)	right parenthesis	)	
[	left bracket	(/	(
]	right bracket	/)	)
‘	left string quote	'('	"
’	right string quote	')'	"

2.9. The 'CODE' Word Symbol

The 'CODE' word symbol is not a part of the set of word symbols in the ALGOL Report. The word "code" is used in the report to indicate that a procedure's body does not appear with its declaration, but appears instead outside the program in which it is declared, as a procedure written either in ALGOL or some other source language. Its use with this Translator has that same purpose, but it will be considered as a true word symbol with the form 'CODE'.

For discussion of the code procedure see Section 4.3.

2.10. The 'FINIS' Word Symbol

There is no provision in ALGOL for notifying a translating program that the end of a source program has been reached. It has been found desirable, from the standpoint of efficient translation of ALGOL source programs, to have some means of signaling the end of a source program, and the word symbol 'FINIS' is used for that purpose with this Translator. Hence 'FINIS' must be the last word symbol of every ALGOL source program, following the final 'END' of the program or comments after this final 'END'.

3. Input/Output Operations

3.1. Introduction

There is no specification in the ALGOL Report for input/output operations in ALGOL. This was not an oversight on the part of the designing committee, but a result of its realization that input/output operations vary so much from one installation to another and from one computer to another that specifications for input/output were better left to each installation. Hence the Translator uses code procedures for input/output. The use of these procedures is described in detail in the following sections.

3.2. Code Procedures for Input/Output

There are several ALGOL code procedures which are associated with the input/output operations presently available through the Translator. These basic I/O procedures are viewed in the same light as standard functions; that is, they are considered to have such importance and universal applicability that they are global to all ALGOL programs compiled by the Translator. For the user this means that there is no need to declare the input/-output procedures. It further implies that the identifiers used for these procedures must have the same restricted use as those set aside for the standard functions, sin, cos, exp, etc. To use the identifiers for any other purpose can cause an error condition. However, one can "submerge" any of these procedure names by declaring a procedure or variable with the same name, as one can do with ordinary identifiers in nested blocks.

For example,

```
begin real a, b, c;
      read (a, b);
      c := a + b;
      print (a, b, c)
end
```

shows the use of the read and print code procedures. Neither has been declared in the example, since this is unnecessary.

On the other hand,

```
begin real a, b, c, d;
      read (b); begin
                     procedure read (e, f);
                     e := f↑2; read (a, b)
                end;
      read (d); c := a + b;
      print (a, b, c, d)
end
```

shows an entirely different use of a declared procedure with the same name as read. This procedure is declared in an inner block, used there, and is no longer defined after exit from that block. Hence the statement "read (b)" causes the real number b to be read; "read (a, b)" causes the calculation a := b↑2 to be made; and "read (d)" causes the real number d to be read.

3.3. Simplified Input/Output

Since ALGOL is a language designed for expressing algorithms in numerical analysis, input and output operations are concerned mainly with the transmission of numerical data.

There are two input procedures and two output procedures designed especially for the user who does not have specific format requirements.

The two simplified input procedures are read and readmatrix, and both accept data in a free form. The form of the read procedure call is

```
read (a, b, c,...)
```

where a, b, c,... are previously declared variables, either simple or subscripted. The procedure reads one variable at a time, so if subscripted variables appear in the list, then subscripts must be specified. For example, let a be an array of dimension 3 x 4 and b and c be simple variables. Then

```
read (a, b, c)
```

is incorrect, while

```
read (a (1,2) b, c)
```

is acceptable. Of course, in the last case, only element a (1,2) will be read, and not the entire array.

If the user has an entire array to be read, a second easy-to-use procedure is available, readmatrix. The form of its call is

```
readmatrix (a, b, c,...)
```

where a, b, c,... are array identifiers. The procedure reads elements of an array in such a way that the last subscript changes first, then the preceding one, etc.

The input data in both cases is assumed to be in a free form. The data can be any ALGOL number (see section 2.5., ALGOL Report) and placed anywhere on a card. The numbers are separated by three blanks, a comma, or the end of the card (column 72). Successive calls for either of the procedures does not initiate reading from a new card; reading proceeds continuously from one number to the next on a card and when that card is exhausted (column 72) it proceeds to the next.

The two output procedures for simplified use are print and printmatrix. The form of the print call is

print (E_1, E_2, ...)

where E_1, E_2, ... represent arithmetic expressions. Of course, an arithmetic expression may consist of simply a variable name, and in most cases it probably will, so

print (area, depth, velocity * weight)

is an acceptable print procedure call. All the output from such a call will be printed on the off-line printer according to the standard format list

'1X,5E14.7'.

That is, the numbers will be printed in lines of 5, each with 7 digits to the right of the decimal point, in what is commonly called "scientific notation". The number -3765.831 would appear in this notation as

-.3765831E␣␣4

and the number .00376 becomes

.3760000E-␣02

The printmatrix procedure call is

printmatrix (a, b, c, ...)

where a, b, c, ... are array identifiers. Output is by rows in exactly the same format as that of print, 5 elements per line. The 3 x 4 array b would be printed as

$$b_{11}\ b_{12}\ b_{13}\ b_{14}\ b_{21}$$
$$b_{22}\ b_{23}\ b_{24}\ b_{31}\ b_{32}$$
$$b_{33}\ b_{34}$$

No alphabetic data can be input or output with any of the four simplified procedures.

For more control over the format of the input and output, other procedures are available and are described in the following sections.

At this point, it appears desirable to begin using certain unfamiliar terms and notation, such as "syntax" and "semantics" and unconventional brackets < > and vertical lines |. These conventions have been borrowed from the field of linguistics and are highly useful in describing precisely how parts of a language (and ALGOL is a language, however limited it may be) can be put together to mean something to someone or something. The reason for including these conventions here is mainly to be precise in describing certain things omitted by the ALGOL Report, but also to initiate the ALGOL beginner in the terminology of the ALGOL Report. Ability to read and understand the Report will be indispensable to the active ALGOL user, so an attempt to entirely avoid the notation problem would be false economy. If the reader keeps in mind these interpretations of the symbols, he should progress well.

::=	means "is"
|	means "or"
<>	are simply brackets that mean that the terms enclosed by them go together to form a single unit.

For example,

<unsigned integer> ::= <digit> | <unsigned integer>< digit>

can be read "An unsigned integer is either a digit or an entity composed of an unsigned integer followed by a digit." This is simple enough, but the definition is strange in that it uses "unsigned integer" to define "unsigned integer." This is a recursive definition and is quite simple to explain: an unsigned integer is either a single digit (0,1,2,...,9) or an entity composed of a digit following one or more digits. With these conventions in mind, we proceed to an exposition of the more comprehensive input and output procedures.

3.4. The Format Procedure

3.4.1. Introduction

The format procedure provides the basic information to the input/output procedures for the placement and scaling of information, whether it is on a card image as input or on a printed page as output.

In the following section, the complete syntax of the format procedure is given in the same notation used for the ALGOL Report; a discussion of the meanings and uses of the various constructions completes the coverage of the format procedure.

3.4.2. Syntax

```
<format call>       ::= FORMAT (<integer expression>, <format list>)
<format list>       ::= <format string> | <format list>, <format string>
<format string>     ::= <left string quote> <secondary list> <right string
                        quote>
<secondary list>    ::= <secondary> | <secondary list>, <secondary>
<secondary>         ::= <field specifier> | (<format primary>) | <unsigned
                        integer> (<format primary>)
<format primary>    ::= <field specifier> | <format primary>, <field
                        specifier>
<field specifier>   ::= <F-conversion> | <E-conversion> | <X-field>
                        | <H-field> | <void-specification> | <record
                        separator>
```

3.4.3. Semantics

<format call>: The form of the format procedure call is

FORMAT (E, "A, B, C, ...") ,

where the E represents an integer expression and the list of indefinite length, A, B, C, ..., represents units of information concerning the form of data. The integer expression denoted by E above identifies a logical tape unit available to the user. It is the responsibility of the user to satisfy this requirement.

The tape numbers designated by the integer expression E correspond to the tape units as follows:

E	Unit	Use
1	SYSIN1 (input) or SYSOU1 (output)	regular input (output) tape
2	SYSUT1 (A3)	scratch tape
3	SYSUT2 (B3)	scratch tape
4	SYSUT4 (B4)	scratch tape
5	SYSPP1 (B2)	regular punch tape
6	SYSOU1 (B1)	regular print tape
7	SYSIN1 (A2)	regular input tape
9	SYSCK1 (A5)	scratch tape

The term "scratch tape" in the table means that during execution those tapes are available to the user for whatever use he wishes.

The number E=1 is a special all-purpose parameter which, when used, automatically causes designation of the regular input tape (SYSIN1) if the call is readf or readmatrixf or the regular print tape (SYSOU1) if the call is printf or printmatrixf. Using E=5 in connection with an output procedure causes information to be written on the output tape with an indication that it is to be punched rather than printed.

<format list>: This is a list of ALGOL strings separated by commas. No fixed number of such strings is required in a format call, in contrast to the normal procedure call. That is, the format procedure is considered to have an arbitrary number of formal parameters.

Each of the strings must be enclosed in string quotes, and might appear as "A, B, C,...", where A, B, C, ... represents a list (of arbitrary length) of units of information concerning the form of data. These units of information are field-specifiers, which prescribe a form for data, or collections of field-specifiers enclosed in parentheses. The field-specifiers provide for input or output of (1) numerical data in the familiar decimal notation (as 123.76) or in "scientific notation" or exponential form (as $.12376 \times 10^3$), (2) blank fields, and (3) alphabetic-numeric information, such as titles, headings, notes to the user, etc., or act as record separators.

<secondary>: The secondary exists for two important reasons. Both are concerned with the use of a portion of a format list more than once for a given input or output procedure call. To be realistic here, we must assume that the secondary consists of several field-specifiers enclosed by parentheses, and perhaps preceded by an unsigned integer. Such a secondary might appear as

$$3(P_1, P_2, P_3)$$

where the P_i are field specifiers. This has the same effect as the format list

$$(P_1, P_2, P_3), (P_1, P_2, P_3), (P_1, P_2, P_3)$$

and, except in the case mentioned below, the same effect as

$$P_1, P_2, P_3, P_1, P_2, P_3, P_1, P_2, P_3$$

The other use for the secondary enclosed by parentheses occurs when an input or output procedure call lists more variables than are listed in the controlling format procedure call. When the format list has been exhausted but the input or output list has not, then control of format goes back to the last left parenthesis before the end of the format list, and input or output proceeds according to the field specifiers to the right of this left parenthesis.

<primary>: The primary consists of a single field specifier or several field specifiers separated by commas. It should be apparent that in many cases a primary is also a secondary (e.g., when it consists of a single field specifier).

<record separator>: The record separator is a slash or a series of slashes. Since input is in the form of card images on magnetic tape, each slash in the format list causes reading of a new card image; for output, each slash causes a new line of printing or punching to be started. The first field of the new record is that specified by the first field specifier following the record separator. In general, n successive slashes will cause n - 1 blank lines on the printed output or n - 1 successive cards not read.

The format procedure call must account for every column in the unit record with which it is concerned. With input, the originating medium is a card, so every column on the card must be accounted for, beginning with column 1 and continuing through the last column containing information of interest. The Translator assumes that unaccounted for columns remaining to the right in a card image are of no interest. For example,

```
FORMAT (7, "F 10.4, 3 F 15.6, 5 X, F 10.4, 10 X")
```

accounts for all 80 columns on the card, even though the last 10 (71-80) columns are to be skipped and not read. We can accomplish exactly the same thing by

```
FORMAT (7, "F 10.4, 3 F 15.6, 5 X, F 10.4")
```

On the other hand, we cannot ignore leading blank fields (or X-fields, generally). Thus,

```
FORMAT (7, "F 10.4, 5 X, F 10.4")
```

and

FORMAT (7, "10 X, F 10.4, 5 X, F 10.4")

are not equivalent.

The same general idea is true for output, the essential difference being the fact that instead of reading card images, we are printing lines of characters, 132* characters per line, or punching cards, 80 columns per card, and every space must be accounted for. Again all unspecified spaces to the right of specified fields are left blank.

3.5. Field Specifiers

3.5.1. Syntax

<F-conversion>	::= F <unsigned integer> · <digit> | <unsigned integer> F <unsigned integer> · <digit>
<E-conversion>	::= E <unsigned integer> · <digit> | <unsigned integer> E <unsigned integer> · <digit>
<X-field>	::= <unsigned integer> X
<H-field	::= <unsigned integer> H <proper string>
<record separator	::= /| <record separator>/

3.5.2. Semantics

<F-conversion> ·

The F-conversion field specifier is of the form nFw.d, where n , w , and d are unsigned integers. If n = 1, it may be omitted.

- - - - - - - - - - - - - -

*the first character of every output record to be printed is used as carriage control by the 1401. So at most 131 characters per line can actually be printed.

The n in this field specifier denotes the number of such consecutive fields; hence 3F10.3 is equivalent to

F10.3,F10.3,F10.3,

and 1F10.3 is equivalent to simply F10.3.

The w in this field specifier indicates the total width of the field in number of characters. The appearance of numbers in the F-conversion is the familiar form of a sequence of decimal digits in which there appears one, and only one, decimal point. Hence, the total characters in the field must include the decimal point. A number in this conversion may be either plus or minus, so w must also include one column count for the sign.

For input the plus sign may or may not be punched at the discretion of the user; the minus sign must be punched and must precede the most significant digit in the field.

For output, the plus sign will not be printed; the minus sign will be printed in the first column to the left of the most significant digit in the field. Leading zeroes will not be printed.

The d in the field specifier denotes the number of digits to the right of the decimal point. This number does not include space for the decimal point itself. d must not be greater than 20.

For example,

format (6, ‘F 8.4, F 6.2, F 10.3’)

specifies a set of three fields, of 8, 6, and 10 columns respectively. In the first, 4 digits lie to the right of the decimal point (which takes up one column itself). This leaves, of the original 8 columns, one more for the sign and 2 for digits to the left of the decimal point. In the second, 2 digits lie to the right of the decimal point and 2 to the left, leaving, of the original 6 columns, one for the sign and one for the decimal point. In the third, 3 digits lie to the right of the decimal point and 5 to the left.

Suppose we wish to print -12.1372, 21.63, and + 17238.312 according to the above format specification. With b representing blank spaces, our printed line would look like this:

-12.1372	b21.63	b17238.312
field 1	field 2	field 3

<E-conversion>.

The E-conversion field specifier is of the form nEw.d, where n , w and d are unsigned integers. As with the F-conversion, if n = 1, it may be omitted.

The n in this field specifier denotes the number of such consecutive fields; hence 3 E 13.7 is equivalent to

E 13.7, E 13.7, E 13.7,

and 1 E 13.7 is equivalent to E 13.7.

Again paralleling the F-conversion, the w denotes the entire width of the field in number of characters. The appearance of numbers in the E-conversion resembles the form widely known as "scientific notation," a decimal fraction followed by an exponent of 10, as, for example,

$$.78325 \times 10^3.$$

The exact form of numbers in the E-conversion is

$\pm$.dd... d E$\pm$ee,

where d's represent decimal digits, the E implies "exponent follows" and the ee represents a two digit exponent of 10. The two sign positions, one for the number itself and one for the exponent, are indicated by ±. Note that every number in this conversion has at least six columns of its field used for "bookkeeping" symbols:

± . E ±ee

Hence, if a field were specified as E 13.7, the field would be 13 columns wide, only 7 of which can contain digits of the number put into this conversion. Similarly, E 14.9 is an invalid field specification, since only 14 - 6 = 8 columns are available for digits of the number. A specification of E 14.3 does not use all 8 of the columns available to it for placement of significant digits of the number.

For example, if we want to place -138,714.31 into E-conversion form in a field 14 columns wide, we specify E 14.8, and we have

-.13871431E+06.

A field specification of E 12.6 results in

-.138714E+06,

and one of E 14.6 results in

-.138714bbE+06.

In both these last cases, information has been lost in the conversion (the last two digits, 31, of the original number).

The F-conversion and E-conversion are the only conversions presently provided with ALGOL for input/output of numerical information, and integers as data have not been mentioned. There is no special integer conversion, but integers can be handled through either the F-conversion or the ~~E-conversion~~.
For example, the integer 317 becomes, in F 5.0 conversion

+317 or b317

It is important to note that the sign and decimal point must be accounted for. The same number in E 9.3 becomes

+.317E+03 or b.317Eb03;

in this case, we have had to provide for the 6 character spaces always present in the E-conversion. In the E-conversion, d must not be greater than 20.

<X-field>.

The X-field specifier is of the form nX, where n is an unsigned integer. The X-field is a field of n blank spaces. The n cannot be omitted, even if it equals 1.

The X-field makes it easy to space printed output as desired, and permits skipping of unwanted information on input cards. For example, suppose we have cards with six 10-column fields (beginning in column 1) and we wish to read only from the second, third and fifth fields. Assume the data in these fields are in F 10.4 conversion. The format call will look like this:

```
format (6, '10X, 2 F 10.4, 10X, F 10.4').
```

A readf call of

```
readf (A, B, C)
```

will cause the data in fields 2, 3 and 5 to be stored as variables A, B and C, respectively. Note that in the format call above, the sixth field has not been accounted for, and need not be.

<H-field>.

The H-field specifier is of the form

nHss...s,

where the n is an integer and the ss...s is a proper string; i.e., the ss...s is a list consisting of any n characters available in the character set, except the escape symbols.

The use of the H-field is primarily to print labels, titles, variable names, etc., so as to make interpretation of printed output easier. For example,

```
        FORMAT (7, "23 HbCOMPUTEDbAVERAGESbb=bb, F 12.4"),
        PRINTF  (AVG)
```

will cause the 23 characters, including blanks, following H to be printed, followed by the current value of the variable AVG in F 12.4 conversion. If AVG = 138.7642, we would have

```
                    bCOMPUTEDbAVERAGESbb=bbbbbb138.7642
```

as the printed output.

The user is responsible for assuring that n is precisely the number of characters he intends to be in the H-field.

WARNING: The first character of each output record is used by the 1401 as a carriage control character.

3.6. The Input and Output Procedures

3.6.1. Introduction

The input and output procedures must each be preceded by a format procedure call in order for the computer to be able to correctly position and scale the input or output information, as the case may be. The set of simplified input/output procedures assumes a standard format, so that the user need not concern himself with providing formats for them. Indeed, he cannot, since the simplified procedures ignore all formats. Complete information on all input/output procedures follows.

3.6.2. Syntax

```
<read call>               ::= READ (<input list>)
<readf call>              ::= READF (<input list>)
<readmatrix call>         ::= READMATRIX (<array identifier list>)
<readmatrixf call>        ::= READMATRIXF (<array identifier list>)
<input list>              ::= <variable> | <input list, variable>
<array identifier list>   ::= <array identifier> / <array identifier list>,
                              <array identifier>
```

3.6.3. Semantics

<read call>: The form of the read procedure call is

read (a, b, c, ...)

where a, b, c, ... represents a list of variables, simple or subscripted, separated by commas. The variables must have been previously declared. They are read from card images on the input tape (number 7) ignoring any format calls which may appear in the program. The procedure does not start reading automatically from a new card, but accepts ALGOL numbers in any defined form (see the ALGOL Report, section 2.5, Numbers), separated by a comma, three blanks, or the end of a card (column 72), continuously until the input list is exhausted. Further calls for the read procedure cause continuation of reading the same card, not for a new card.

<readf call>: The form of the readf procedure is identical to that of the read call. The difference between the two is that the readf procedure reads input according to the last executed format procedure call.

<readmatrix call>: The form of the readmatrix procedure call is

readmatrix (a,b,c,....)

where a,b,c,.... are array identifiers. The procedure reads elements of an array in such a way that the last index changes first, then the preceding one, etc. The elements are acceptable in any ALGOL number form, separated by three blanks, a comma, or the end of a card (column 72).

<readmatrixf call>: The form of the readmatrixf procedure call is identical to that of the readmatrix. The difference between the two is that the readmatrixf procedure reads input according to the last executed format procedure call.

<input list>: The form of the input list is

A, B, C, ...

where A, B, C, ... represents a series of previously declared identifiers. They may be simple variables or elements of an array; in the latter case, the subscripts must be present, as for example a (2,3) and b (7,6). The array identifier above, without the subscripts, is not acceptable.

The user should keep in mind that the format procedure call not only controls the form of the data but also prescribes the logical tape number from which the data is read (see section 3.4.3.).

3.6.4. Syntax

<print call>	::= PRINT (<output list>)
<printf call>	::= PRINTF (<output list>)
<printmatrix call>	::= PRINTMATRIX (<array identifier list>)
<printmatrixf call>	::= PRINTMATRIXF (<array identifier list>)
<output list>	::= <arithmetic expression> | <output list>, <arithmetic expression>
<array identifier list>	::= <array identifier> / <array identifier list>, <array identifier>

3.6.5. Semantics

<print call>: The form of the print procedure call is

print (E_1, E_2, ...)

where E_1, E_2, ... represents arithmetic expression. The procedure evaluates the arithmetic expressions at execution time and places the results on the output tape for the off-line printer according to the standard format list

'1X, 5E14.7'

<printf call>: The form of the printf procedure call is

printf (E_1, E_2, ...)

where the E_1, E_2, ... represent arithmetic expressions. Despite its name, the procedure can be used for various output tasks, such as placing intermediate results on scratch (utility) tapes, placing card images on the punch output tape for punching into cards, or printing output on the off-line printer, depending upon the logical tape unit prescribed by the last executed format procedure call preceding the printf procedure call (see section 34.3.), which also controls the data transmitted.

<printmatrix>: The form of the printmatrix procedure call is

printmatrix (a,b,c,....)

where a,b,c,.... are array identifiers. The elements of the array are printed by rows on the off-line printer according to the standard format list

'5F14.7'

Hence, the 2 x 3 matrix a will be printed as

a_{11} a_{12} a_{13} a_{21} a_{22}
a_{23}

<printmatrixf>: The form of the printmatrixf procedure call is

printmatrixf (a,b,c,....)

where a,b,c,.... are array identifiers. The elements of the array are output to the tape unit specified by the *last executed* format procedure call preceding the printmatrixf procedure call, which also controls the format of the data thus transmitted.

<output list>: The output list consists of arithmetic expressions of any kind, separated by commas, but *cannot* be void. That is, an output procedure such as

printf( )

is not valid, even though the controlling format may consist entirely of an H-field. Carriage control characters for the 1401 may be inserted into output records by using an F1.0 field specifier. For example,

```
FORMAT (1, "F1.0, HPAGE HEADING")
PRINTF(1)
```

will produce the following output line

```
1 PAGE HEADING
```

The numeral 1 in the first character position will be used by the 1401 for carriage control and will cause a page to be ejected before the heading is printed.

3.7. Data

The actual introduction of data into the computer is handled by subroutines of the Operating System (see section 5.), and requires the use of one basic control instruction

```
$ DATA
```

which must appear on a card, with the $ in column 1, and must immediately follow the last card of the ALGOL source program. The data cards then follow the $ DATA card. Form of data has been discussed in Sections 3.3., 3.4., and 3.5. The $ DATA card is not required if the program has no associated data.

4. Procedures

4.1. Introduction

One of the most useful features of ALGOL is the ease with which subprograms or pieces of programs that are frequently used can be included in a given ALGOL program by means of the ALGOL procedure. There are generally two types of procedures which will be considered for use by ALGOL users:

1) those written in ALGOL and
2) those written in some other source language.

We consider each in turn.

4.2. Procedures written in ALGOL

These procedures present no unusual problems. They should be written in accordance with the description of procedures in the ALGOL Report and transliterated for machine use as described in the section of this manual concerned with hardware representation.

4.3. Procedures written in other source languages

This type of procedure can be incorporated into an object program produced by the ALGO Translator. However, the user must write such a procedure so as to be compatible with the translator-produced object program. The information necessary to do this may be obtained from the ALCOR-ALGOL reference manual which is on file in the programming systems library in Computer Sciences Center.

5. The IBSYS Operating System and the ALGOL Translator

The Translator runs under the control of the IBSYS Operating System and many of the "housekeeping" chores for the Translator such as the manipulation of magnetic tapes, loading the translated program for execution, processing error conditions, and input/output conversion are performed by the common routines of the system.

The 7094/1401 computer system at the Computer Sciences Center, is operated on an open-shop programming and closed-shop operating basis. This means that the user must submit his program, punched into cards, to the Computer Sciences Center for running and cannot have direct access to the computers. The programs submitted are arranged in "batches" of many jobs, placed on magnetic tape by the 1401 computer, and processed one after the other by the 7094. Every job must, then, be appropriately identified with a program ($ID) card. This card is described in Appendix A. Since the system must be informed what processing is to be carried out on the job following the $ID card, $ control cards must be placed before that job. The control cards necessary for translation and execution of an ALGOL program are described in Section 6.3.

6. Procedure for Submitting an ALGOL Job

6.1. Introduction

This section concerns the details of actually preparing an ALGOL source program for entry into the computer. Of necessity, the source program must be punched into cards, appropriately identified, accompanied by certain IBSYS control cards, and properly submitted at the dispatching room.

The sole object of this section is to make this ALGOL Manual self-contained, in that if a potential user knows how to program in ALGOL, then all the information he needs to put an ALGOL job on the computer is available to him in this manual. If he requires more than this minimum body of information concerning non-ALGOL portions of the operating system, subroutine library, etc., then he will have to look to other publications. The procedures outlined in this chapter may be changed in time, so the reader is cautioned to assure that he has the latest version.

6.2. Coding and Keypunching

Assuming that the user has completed the planning of his program, the next step is to place the ALGOL statements on a coding sheet so as to facilitate keypunching. We assume that BCD cards are used as the primary input medium of the source program. Hardware ALGOL is a one-dimensional continuum, so placement of the statements on cards has no meaning of any kind. That is,

```
'BEGIN' 'REAL' VOLUME, PRESSURE, TEMPERATURE., 'INT
EGER' NUMBER, EXPERIMENT, DATE., 'REAL' 'PROCEDURE'
INTEGRATION (ALPHA, BETA, GAMMA).,
```

is as acceptable and meaningful as

```
'BEGIN'
     'REAL' VOLUME, PRESSURE, TEMPERATURE.,
     'INTEGER' NUMBER, EXPERIMENT, DATE.,
     'REAL' 'PROCEDURE'
          INTEGRATION (ALPHA, BETA, GAMMA).,
```

even though the second version is somewhat more readable for humans. Except in strings, blanks have no meaning in ALGOL, so blanks can be inserted at will, to make it easier to human readers to read and understand an ALGOL program. Since placement of the ALGOL statements on cards is strictly a matter of personal preference, there are no special ALGOL coding forms. The only restriction to placement on cards is that columns 73-80 should not contain ALGOL symbols; these columns are used strictly for identification, and are not read by the Translator. Use of this field for identification is optional with the user; they may be left blank if so desired. It is, however, suggested that some numbering system be used in this field to assure that the cards are in their proper order.

For examples of ALGOL programs that have been translated and executed, and whose structure is apparent to the human reader, see Appendix D of this manual. In order to make the various parts of the programs more easily understood by human readers, liberal use has been made of the <u>comment</u> word symbol and comments after the final <u>end</u> of the programs.

6.3. System Control Cards for ALGOL

6.3.1. The $ID card

This card is same $ID card that is used for all Computer Sciences Center systems. It must be the first card of every job. The $ID card is described in detail in Appendix A.

6.3.2. The $EXECUTE ALGOL card

This card is required on all ALGOL jobs. The form of the card is:

```
1              16
$EXECUTE        ALGOL
```

When this card is read by IBSYS the first record of the ALGOL subsystem is read into core storage and control is relinquished to it. Any $-control card which appears after the $EXECUTE card is processed by the ALGOL system itself.

6.3.3. ALGOL Monitor Control Cards

a) The $COMPILE ALGOL Card.

This card is required for every ALGOL source program. It causes translation of the source program that follows this card.

b) The $NOGO Card.

This card causes the source program to be translated if there is a $COMPILE ALGOL card present but execution will be deleted. The option $GO may be used to indicate that execution is desired after translation. GO is redundant since the monitor always assumes that execution is desired unless $NOGO appears in the deck.

c) The $DECK Card.

This card requests that a binary object deck be punched for the program being translated.

d) The $DUMP Card.

This control card option permits one area of memory to be dumped if the program is terminated for any reason other than a normal exit. The form of the card is:

```
1
$DUMP(OCT1,OCT2,M)
```

OCT1 and OCT2 are two octal addresses, and the area between OCT1 and OCT2 will be dumped in mode M which is interpreted as follows:

M = 1	OCTAL
M = 2	BCD
M = 3	SQUEZY - mnemonics with address & tag
M = 4	SQUEZY and OCTAL
M = 5	OCTAL and mnemonics
M = 6	OCTAL, mnemonics, and BCD

e) The $DATA Card.

If the program contains data, then the data must be preceded by a $DATA card.

Any of the above described ALGOL Monitor options with the exception of $DATA may be combined on one $ control card. The options are separated by commas. For example:

```
$COMPILE ALGOL, DECK,DUMP(43274,77777,6)
```

has the same effect as three separate cards

```
$COMPILE ALGOL

$DECK

$DUMP(43274,77777,6)
```

6.4. Data Cards

The control card $DATA need be present only if data cards for the ALGOL program are present. Of itself, the $DATA card does not cause any activity on the part of the monitor or the Translator but simply states that the cards which follow contain data intended to be read by the immediately preceding ALGOL program.

The data cards themselves should be punched in accordance with the format procedure call which describes them (see section 3.4.).

6.5. Binary Cards and ALGOL

Generally speaking, there are two distinct uses to which binary cards are put: for entire compiled (translated) programs and for subroutines for a source program written in some language other than binary.

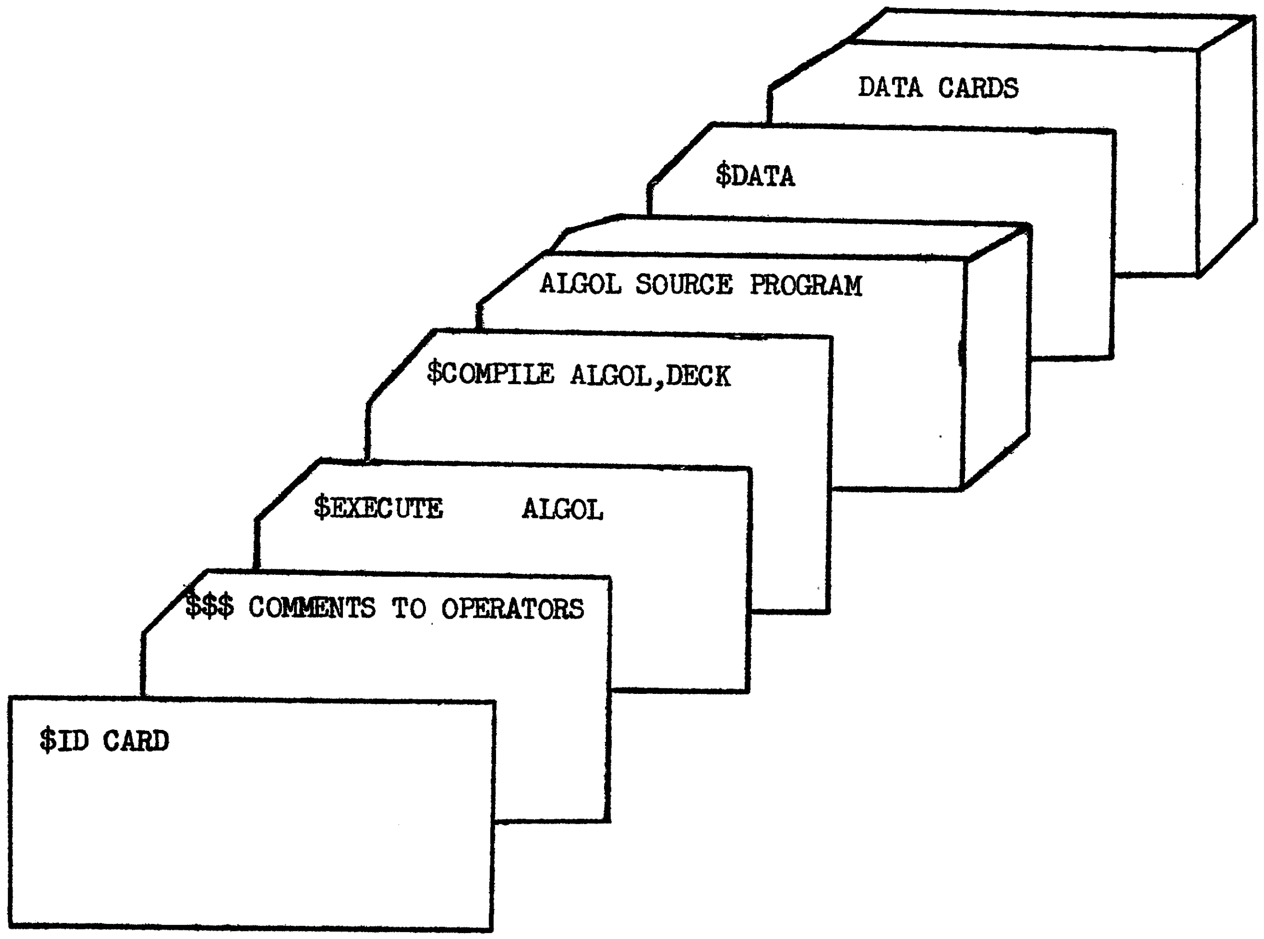
If a user gets, by use of the system control card $DECK, a binary deck of his translated object program, then he may use this binary deck whenever he chooses without further reference to ALGOL.

If, on the other hand, a user has a subroutine on binary cards, no matter where he got it, he must satisfy the requirements for the use of the contents of the binary deck as a code procedure. These requirements are described in detail in section 4.3.

6.6. Library Routines on Tape or Cards

The subroutine library is at the disposal of the ALGOL user through the code procedure declaration. This facility is not yet available, but will be in the near future; at that time, specific instructions for its use will be available as a replacement for this section.

6.7. Typical Deck Makeup



The control cards shown here will cause the ALGOL source program to be translated and produce a binary object deck. Also the absence of the NOGO option specifies that execution is desired if there are no fatal translation errors.

7. Errors and Error Messages

7.1. Introduction

The most simple-minded translating programs simply reject a source program when an error in syntax has been found. Such a summary dismissal of an erroneous source program can be a frustrating and time-consuming experience for the user, so more sophisticated translators now perform extensive syntax analysis during translation of the source program, and attempt to "fix" the source program where errors are detected and continue translation. Obviously, with a language such as ALGOL wherein redundancy is purposely minimized, only minor errors can be repaired, and a truly gross error can only result in abortion of the translating attempt.

7.2. Classes of Errors and Their Detection

7.2.1. Syntax Error

Suppose

<u>if</u> (Boolean Expression) <u>then</u> Expression E_1; <u>else</u> Expression E_2

appears in a source program; the semicolon after E_1 obviously is out of place, and a well-designed translator can recognize this, disregard the semi-colon, and translate the statement correctly. On the other hand, if the programmer omits every <u>end</u> in a multiple-block program, the translator normally has insufficient information on which to base an attempt at inserting the missing <u>end</u>'s, and so must abandon its attempt at translation. That is, a sequence of <u>begin</u>'s

```
        begin
        begin
        begin
        begin
        begin
```

might be interpreted in several ways as far as block structure is concerned. Consider

```
    begin                             begin                 begin
    begin   end                       begin                 begin
    begin   end                       begin   end           begin   end
    begin   end                       begin   end           begin   end
    begin   end                       begin   end           end
    end                               end                   end
                                      end                   begin   end.
```

These are just a few of the several structures five begin's can imply; no computer program can be expected to pick the correct one in a given case.

Perhaps the foregoing example is far-fetched. Consider the case where 'BEGIN' is written BEGIN. BEGIN is then nothing more than another identifier, and has lost its identity as a basic ALGOL symbol. Presumably, under such circumstances an 'END' will appear, and have no 'BEGIN' to be matched with. It is not reasonable to expect such errors to be repairable, since the identifier BEGIN could legitimately be used in the same program in which the mistake might appear.

7.2.2. Semantic Error

A second type of error likely to occur, apart from syntactical or "grammatical" errors, is the case in which the program simply does not do what the user had intended. The source statements may very well constitute a valid ALGOL program, the Translator may translate it correctly, the data may be perfectly acceptable—but the results are not those expected.

Such an error condition can never be detected by anyone or any machine unless he or it knows what the programmer meant to tell the computer to do.

At present there is no alternative to a careful examination of the structure of the source program by the user or programmer. In some programming systems, a trace mode is provided to aid the user in following through the action of the program. There is no such special device available for ALGOL users, but a liberal use of print calls at appropriate points in the program will accomplish the same end result.

7.2.3. Errors Detectable Only During Execution

A third type of error which from time to time appears is a pathological condition during execution after a successful translation. This can result from two somewhat related occurrences. The first cause could be that the data introduced into the machine were not within the domain of the data for which the program was written. The second is that, although the syntax of the ALGOL source program is correct, the semantics of the program is not; that is, the statements are grammatically correct but under certain circumstances the program can enter a loop and never come out of it. An example:

```
begin real a; 0;
              a := 0
        again: a := a + 1;
        go to again; print (a)
end
```

Syntactically, this is a perfectly good ALGOL program, but it will run indefinitely; it can never reach the print procedure call because the go to statement will always send control back to statement "again". Hence, this type of error will show up only during actual operation of the object program. This type of error cannot, then, be detected at translation time, and must be dealt with by a "post mortem" on the "dead" object program.

Note that there is only a fine distinction between this last error type and that discussed in Section 7.2.2. The essential difference lies in the fact that the first need not stop execution of the program, whereas the second will either stop execution or make the computer perform in such a peculiar manner (as an indefinite loop) as to require operator intervention.

7.2.4. Machine and/or Translator Errors

Certain error conditions are not necessarily due to negligence on the part of the programmer, but rather to limitations inherent in the Translator. The most obvious limitation is that on the length of program, or portion thereof, imposed by the size of the computer's core memory, which is, of course, not infinitely large. Such error conditions can frequently be corrected by minor alterations to the source program. If changes are impossible or impractical, then a talk with the programmer charged with maintenance of the Translator is in order. If no such individual exists, then the thwarted user is referred to the Programmer's Manual, which describes the structure of the Translator in some detail and contains information for dealing with these error conditions.

Finally, it must be pointed out that computers are only machines and are not infallible. They make mistakes. To be sure, the incidence of computer errors is low when compared to that of humans, and the design of the machine is usually such that errors of this nature are detected, but nevertheless, the user should expect, from time to time, to find that his program did not run and that the failure was actually due to machine trouble. The remedy is simple: report a suspected machine error to the consultant on duty and try to run the program again after the computer has been checked and verified as accurate. A tolerant attitude toward machine errors will make life more pleasant for the computer user.

7.3. Error Messages

The various error types discussed in the preceding sections which are detectable by this Translator are reported to the user by means of printed error messages, along with whatever other printed information he has requested (See Section 5.).

The Translator numbers consecutively every card image of an ALGOL source program, and the output from every program submission contains these numbered card images, otherwise exactly as they were punched into the cards. The numbered card images are particularly helpful in analyzing programming errors when they occur.

Figures 7.1 and 7.2 show typical output for erroneous programs.

ALCOR-ILLINOIS 7090 ALGOL COMPILER

```
 1      'BEGIN'   'REAL'  'PROCEDURE' B(X).,  'INTEGER' X.,   'CODE' .,
 2            'REAL'  C.,
 3     •'PROCEDURE'  M(X).,  'CODE'.,
 4      M(3).,   M(LABEL).,
 5      LABEL..  C.= B(3).,
 6
 7      'BEGIN'   'REAL'  AZ.,
 8               'READ' 'ARRAY'  A(/1.0.10,SIN(3.0,6.3).. B, AZ..
                                                     B(LABEL)  .,
 9
10      'END'
11      'END'
12      'FINIS'
```

```
     CARD NO.          SYNTACTICAL ERRORS IN ALGOL PROGRAM

8009     8        ILLEGAL OCCURRENCE OF CHAR. UNDEFINED DELIMITER IN OR AFTER
                  STATEMENT FOLLOWING A BLOCK BEGIN , IT HAS BEEN DROPPED FROM
                  THE SOURCE PROGRAM.

   1     8        ILLEGAL CONSTANT.  TWO PERIODS.  THE CHARACTER FOLLOWING IS

  21     8        ARRAY DECLARATION.  BOUND PAIR SEPARATORS NOT RIGHT

2026     8        CALL OF FUNCTION OR PROCEDURE.  THE NUMBER OF PARAMETERS IN
                  SIN WITH KIND REAL CODE FUNCTION WITH PARAMETERS DIFFERS
                  FROM PREVIOUS USE

1012     8        IDENTIFIER  B  IS NOT SIMPLE OR FUNCTION WITHOUT PARAMETERS

1030     8        IDENTIFIER  AZ  IS NOT DECLARED OR IS UNDEFINED AT THIS
                                                                 POINT

2003     8        CALL OF FUNCTION OR PROCEDURE.  ACT. PAR. LABEL (  LABEL   )
                  NOT COMPATIBLE WITH FORMAL PAR.  (  INTEGER    SIMPLE  )

8012     8        A ARRAY DECLARATION IS ENDED BY A  .,  BUT IS NOT COMPLETED.
```

FIGURE 7.1.

```
                    ALCOR-ILLINOIS 7090 ALGOL COMPILER

   1       'BEGIN'  'REAL' A,B,C.,
   2       A=A(1).,
   3         B= A(2).,
   4       'GOTO' A.,  C=A(3).,
   5       'GOTO' A.,  A(4).,
   6       'GOTO' A.,    A(4).,
   7       'END'   'FINIS'

    CARD NO.           SYNTACTICAL ERRORS IN ALGOL PROGRAM

1031     2        IDENTIFIER  A  IS NOT ARRAY OR FUNCTION WITH PARAMETERS

4016     3        IDENTIFIER  A  WAS USED TWICE WITH THE SAME WRONG TYPE OR
                  KIND.  TO SAVE SPACE SUCH ERRORS ARE ONLY PRINTED THE FIRST
                  TIME THEY OCCUR.  CHECK PROGRAM

1025     4        IDENTIFIER  A  IS NOT LABEL

1009     5        IDENTIFIER  A  IS NOT ARRAY OR PROCEDURE WITH PARAMETERS

9001              THE FOLLOWING TOLERATED CHAR. WERE USED IN PROGRAM., =
```

FIGURE 7.2.

8. A Feature of ALGOL Not Implemented, own

The type declaration own has not been implemented in the Translator and will, therefore, be ignored and passed over as if it were not present if it appears in a source program.

No plans exist at present for the implementation of this ALGOL feature. However, a judicious use of global variables should make it possible for the careful programmer to accomplish the same thing as would be accomplished by the use of own. That is, by keeping "own variables" global, instead of local to certain blocks, they remain defined throughout execution of the program.

A P P E N D I C E S

Appendix A.

Deck Identification Card Format

Columns 1-3	$ID
Columns 4-6	must be blank
Columns 7-72	Accounting information arranged in five fields and separated by asterisks as described below.

Example 1:

```
$ID   70777*2*20*15*James Arnold--C.S. 200
```

The first parameter is an account number assigned by Computer Sciences Center to each project or course. This is either a four or five digit number. The first digit must be punched in column seven.

The second parameter is an estimate of the maximum 7094 processing time (in minutes) required by the job. This estimate includes all compilation time and program execution time. If the actual run time is less than the estimate, the user is charged only for the time used. If the run time exceeds the estimate, the job is automatically terminated but only the estimated time is charged to the users account.

The third and fourth fields contain estimates of the maximum output that will be produced by the object program. The first is an estimate of the number of pages to be printed (60 lines per page) and the second an estimate of the number of cards to be punched. These estimates are only for execution-time output. Program listings, binary object decks, and any other system outputs are excluded from this estimate. If either of these estimates is exceeded the job will be automatically terminated with the message "OUTPUT ESTIMATE EXCEEDED".

The last field contains the name of the person submitting the job followed by the name of his department and course number (if any).

The above example indicates that the job is to be charged to account number 70777, will require no more than 2 minutes of computer time, with output a maximum of 20 pages and 15 punched cards, and is to be returned to James Arnold. Notice that leading zeros are not required. In fact a zero parameter may be indicated by two consecutive asterisks. For example:

```
$ID   1106*20*1200**Raymond Crawford--E.E.
```

This card specifies a 20 minute time estimate, a maximum of 1200 pages of output and no punched cards.

Every job deck submitted to the Computer Sciences Center for processing must include an $ID card. This card must be the first card of the deck and it is very important that it be punched exactly as described above. Several routines within the system scan this card prior to processing and, if inconsistencies are detected, reject the job. If this occurs, the message "ILLEGAL ID CARD" is output for the offending job.

Appendix B.

Bibliography on ALGOL-60

1. "Introduction to ALGOL," Baumann, Samelson, Bauer and Feliciano, Oak Ridge National Laboratories.

This is the revised and extended version of the ALGOL Manual of the ALCOR group, translated from the original German at Oak Ridge National Laboratories. It is a tutorial paper of some 100 typewritten pages and is by far the best presently available, in the writer's opinion, for individual study by persons not previously familiar with ALGOL or any other similar automatic programming language. It is well-written, and on an elementary level.

2. "Structure and Use of ALGOL-60," H. Bottenbruch, Journal of the Association for Computing Machinery 9 (April, 1962), p. 161-221.

This is a well-written tutorial paper by a staff member of Oak Ridge National Laboratories. It is somewhat more advanced in presentation than [1], and for this reason is not recommended for persons who have no previous knowledge of programming. However, it is highly recommended for programmers desiring a complete exposition of the subject of the structure and use of the ALGOL language. This publication is available from the Digital Computer Laboratory as a reprint to qualified users.

Appendix B.

Bibliography on ALGOL-60 (cont'd.)

3. "An Introduction to ALGOL-60," H. R. Schwarz, Communications of the Association for Computing Machinery 5 (February, 1962) p. 82-95.

The object of this paper is to explain the ALGOL Report with descriptions of the syntactic structures and examples. It is not intended to be a complete introduction to programming via ALGOL, and is not, therefore, recommended as a first paper on the subject. It should prove valuable, particularly in understanding the ALGOL Report, to those who have already read [1], [2], [6], or [7]. This paper is available (September, 1963) as a reprint from the ACM.

4. "Introduction to ALGOL and its Application," H. Rutishauser, File No. 452, DCL, University of Illinois, May 4, 1962.

This is a paper describing the structure of the ALGOL language and is exceptionally rich in non-trivial expository examples. In view of the ready availability of [1] and [2], this paper can only be recommended as supplementary reading material. Limited copies are available at the DCL to qualified users.

5. "An Introduction to ALGOL-60," M. Woodger, Computer Journal 3 (1960), p. 67-75.

This paper is an effort to make understandable the ALGOL Report and as such, it succeeds. It can be recommended only as supplementary reading material.

Appendix B.

Bibliography on ALGOL-60 (cont'd.)

6. "A Primer of ALGOL-60 Programming," E. W. Dijkstra, Academic Press, 1962.

This is a text explaining the structure and use of ALGOL and includes the version of the ALGOL Report published in May, 1960 in Communications of ACM. It is rather expensive ($6.00).

7. "A Guide to ALGOL Programming," by D. D. McCracken, John Wiley and Sons, Inc., 1962.

This is a text, including examples and problems (solutions for which are provided), and is one of the best commercially available for the beginning programmer. Rather than an exposition of ALGOL in great detail, this publication is a text on programming computers which uses ALGOL as the programming language.

Appendix C.

Hardware Representation of ALGOL-60 Elements

ALGOL Symbol	Symbol Name	Hardware Rep.	Tol. Hrdwe Rep.
A|B|...|Z	upper case alphabet		
a|b|...|z	lower case alphabet	A|B|C|...|Z	
0|1|2|...|9	numerals	0|1|2|...|9	
<u>true</u>	Boolean true	'TRUE'	
<u>false</u>	Boolean false	'FALSE'	
+	plus sign	+	
-	minus sign	-	
X	multiplication sign	*	
/	division sign	/	
÷	integer division sign	//	
↑	exponentiation	'POWER'	**
<	less than	'LESS'	'LS'
≤	less than or equal to	'NOT GREATER'	'LQ'
=	equal to	'EQUAL'	'EQ'
≥	greater than or equal to	'NOT LESS'	'GQ'
>	greater than	'GREATER'	'GR'
≠	not equal to	'NOT EQUAL'	'NQ'
≡	logical equivalent	'EQUIV'	'EQV'
⊃	logical implies	'IMPL'	'IMP'
∨	logical or	'OR'	
∧	logical and	'AND'	
¬	logical negation	'NOT'	

Appendix C.

Hardware Representation of ALGOL-60 Elements (cont'd.)

ALGOL Symbol	Symbol Name	Hardware Rep.	Tol. Hrdwe Rep.
go to		'GO TO'	
if		'IF'	
then		'THEN'	
else		'ELSE'	
for		'FOR'	
do		'DO'	
,	comma	,	
.	decimal point	.	
$_{10}$	base 10	' (apostrophe)	
:	colon	..	
;	semi-colon	.,	
:=	assignment sign	.=	
# or b	blank space		
step		'STEP'	
until		'UNTIL'	
while		'WHILE'	
comment		'COMMENT'	
(	left parenthesis	(	
)	right parenthesis	)	
[	left bracket	(/	
]	right bracket	/)	

Appendix C.

Hardware Representation of ALGOL-60 Elements (cont'd.)

ALGOL Symbol	Symbol Name	Hardware Rep.	Tol. Hrdwe Rep.
‘	left string quote	'('	
’	right string quote	')'	
<u>begin</u>		'BEGIN'	
<u>end</u>		'END'	
<u>own</u>		'OWN'	
<u>boolean</u>		'BOOLEAN'	
<u>integer</u>		'INTEGER'	
<u>real</u>		'REAL'	
<u>array</u>		'ARRAY'	
<u>switch</u>		'SWITCH'	
<u>procedure</u>		'PROCEDURE'	
<u>string</u>		'STRING'	
<u>label</u>		'LABEL'	
<u>value</u>		'VALUE'	
<u>code</u>		'CODE'	
<u>finis</u>		'FINIS'	

Note: See Section 2. for discussion of hardware representation and tolerated hardware representation.

Appendix D.

<u>Examples</u>

There follow several examples of ALGOL procedures and complete ALGOL programs. Their purpose in appearing here is to illustrate the transliteration from publication ALGOL to hardware ALGOL.

Example 1. Example from Section 5.4.2, ALGOL Report.

```
procedure Spur (a) Order:  (n) Result:  (s);
     value n; array a; integer n; real s;
     begin integer  k;
          s := 0;
          for k := 1 step 1 until n do
          s := s + a [k, k]
     end
```

```
'PROCEDURE' SPUR (A) ORDER..(N) RESULT..(S).,
     'VALUE' N., 'ARRAY' A., 'INTEGER' N., 'REAL' S.,
     'BEGIN' 'INTEGER' K.,
          S .= 0.,
          'FOR' K .= 1 'STEP' 1 'UNTIL' N 'DO'
          S .= S + A (/K, K/)
     'END'
'FINIS'
```

Example 2. Example from Section 5.4.2, ALGOL Report.

```
procedure Transpose (a) Order: (n); value n;
     array a; integer n;
          begin real w; integer i, k;
               for i := 1 step 1 until n do
                     for k := 1 + i step 1 until n do
                           begin w := a [i, k];
                                a  [i, k] := a [k, i];
                                a  [k, i] := w
                           end
          end Transpose


'PROCEDURE' TRANSPOSE (A) ORDER .. (N)., 'VALUE' N.,
     'ARRAY' A., 'INTEGER' N.,
          'BEGIN' 'REAL' W., 'INTEGER' I, K.,
               'FOR' I .= 1 'STEP' 1 'UNTIL' N 'DO'
                    'FOR' K .= 1 + I 'STEP' 1 'UNTIL' N 'DO'
                         'BEGIN' W .= A (/ I, K/).,
                              A (/I, K/) .= A(/K, I/).,
                              A (/K, I/) .= W
                         'END'

          'END' TRANSPOSE
'FINIS'
```

Example 3. Example 1, ALGOL Report.

```
procedure euler (fct, sum, eps, tim); value eps, tim;
integer tim; real procedure fct; real sum, eps;
comment euler computes the sum of fct (i) for
i from zero up to infinity by means of a
suitably refined euler transformation;
begin integer i, k, n, t; real array m [0: 15];
     real mn, mp, ds; i := n := t := 0;
     m [0] := fct (0); sum := m [0] /2;
     next term: i := i + 1; mn:= fct (i);
          for k := 0 step 1 until n do
               begin mp := (mn + m [k] )/2; m [k] := mn;
                  mn := mp
               end means;
          if (abs (mn) <abs (m [n] ) ) ∧(n < 15) then
              begin ds := mn/2; n := n + 1;
                 m [n] := mn
              end accept                         else
              ds := mn;
              sum := sum + ds;
          if abs (ds) <eps then t := t + 1 else t := 0;
          if t < tim then go to next term
end euler
```

Example 4., continued.

```
$ ALGOL
$ GO
'BEGIN' 'COMMENT' COMPLEX DIVISION USING ALGORITHM 116,
COMMUNICATIONS OF ACM, AUG. 1962.,
     'REAL' R, P, Q, S, T, U., 'INTEGER' N.,
     'PROCEDURE' COMPLEXDIV (A, B, C, D) RESULTS., (E, F).,
     'VALUE' A, B, C, D., 'REAL' A, B, C, D.,
     'COMMENT' COMPLEXDIV YIELDS THE COMPLEX QUOTIENT OF
     A + IB DIVIDED BY C + ID.,
     'BEGIN' 'REAL' R, DEN.,
          'IF' ABS (C) 'NOT LESS' ABS (D) 'THEN'
          'BEGIN' R .= D/C.,
               DEN .= C + R * D.,
               E .= (A + B * R) / DEN.,
               F .= (B - A * R) / DEN
          'END'
                                        'ELSE'
          'BEGIN' R .= C/D.,
               DEN .= D + R * C.,
               E .= (A * R + B) / DEN.,
               F .= (B * R - B) / DEN
          'END'
     'END' COMPLEX DIV.,
     READ (R, P, Q, S).,
     COMPLEX DIV (R, P, Q, S, T, U).,
     PRINT (R, P, Q, S, T, U)
'END'
'FINIS'
```